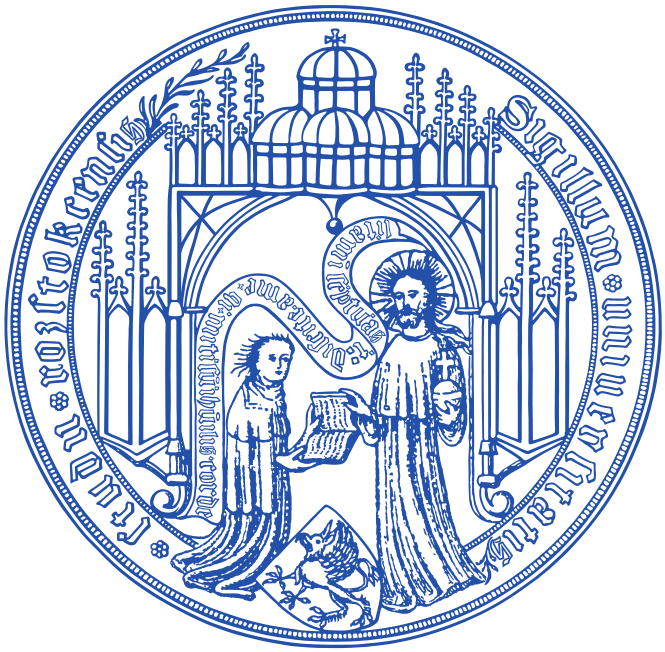
Universität Rostock
Institut für Informatik
Lehrstuhl für Softwaretechnik



Diplomarbeit

Thema	**Erarbeitung und prototypische Anwendung von Konzepten zur dynamischen Textgestaltung**
Eingereicht von	Johannes Rössel, Matrikel-Nr. 6100116 Geboren am 29. April 1985
Erstgutachter	Herr Prof. Dr.-Ing. habil. Peter Forbrig
Zweitgutachter	Herr Prof. Dr. rer. nat. Clemens H. Cap
Betreuerin	Frau Dr.-Ing. Anke Dittmar
Eingereicht am	17. Oktober 2011

Danksagung

An dieser Stelle möchte ich einigen Personen danken, die mich beim Verfassen dieser Arbeit unterstützt haben und dafür gesorgt haben, dass der Text nicht völlig konfus oder fehlerhaft ist.

Mein besonderer Dank gilt meiner Betreuerin Dr. Anke Dittmar, die sich durch meine Konzepte, Ideen und Entwürfe arbeiten musste bevor daraus lesbare Kapitel wurden. Ebenso stand sie mir unermüdlich und mit hilfreichen Anmerkungen zur Seite.

Ebenso möchte ich Stefan Piehler, Andreas Dähn, Jonas Ritze und Alexander Sturm danken, die in zahlreichen Diskussionen wichtige Gedanken und Ideen lieferten, die in diese Arbeit einflossen. Alexander und Andreas haben diese Arbeit auch in verschiedenen Stadien der Vollständigkeit korrekturgelesen und damit einen wesentlichen Beitrag dazu geleistet, dass der resultierende Text verständlich und lesbar ist.

Meinem Vater bin ich insbesondere dankbar für inhaltliche Diskussionen, Korrekturlesen dieser Arbeit sowie Ratschläge und Hilfe bezüglich Schriftsatz in der verwendeten Software.

Zu guter Letzt gilt auch meiner Freundin Dank dafür, dass sie während des Verfassens dieser Arbeit stets verständnisvoll war, obwohl ich wahrlich mehr Zeit an dieser Arbeit als mit ihr verbrachte.

Zusammenfassung

Traditionell besitzen Texte eine lineare Struktur; sie werden von Anfang bis Ende gelesen. In den letzten Jahrzehnten kam mit Hypertext eine weitere Art Text dazu. Hierbei bildet der Text ein Netzwerk miteinander verknüpfter Textabschnitte. Dies eröffnet die Möglichkeit, mehrere verschiedene Lesepfade durch einen Text zu haben, was bei linearen Texten nicht gegeben ist. Hypertexte stellen aber oft Herausforderungen dar, da ein Leser selbst entscheiden muss, wo er weiterliest. Dies kann dafür sorgen, dass der gelesene Text inhaltliche Brüche aufweist, die das Verständnis beeinträchtigen.

In dieser Arbeit soll eine Art Mittelweg zwischen den beiden Formen gesucht werden, sogenannte dynamische Texte. Diese sollen zwar wie Hypertext verschiedene Lesepfade ermöglichen, aber dennoch den inhaltlichen Zusammenhang linearer Texte bieten. Hierzu werden an die jeweiligen Textvarianten Anforderungen und Ziele modelliert, welche dann vom Leser vor dem Lesen ausgewählt werden können. Die Sicherstellung guter Lesbarkeit obliegt im Wesentlichen dem Autor, da immer noch mehrere Textvarianten modelliert werden. Deswegen soll in dieser Arbeit ebenfalls ein Vorschlag unterbreitet werden, wie ein Autor beim Verfassen dynamischer Texte vorgehen kann.

Solche Texte könnten beispielsweise in der Lehre oder in interdisziplinären Arbeitsgruppen eingesetzt werden, wo das Vorwissen jeweils verschiedener Personen sehr unterschiedlich sein kann. Dynamische Texte könnten es ermöglichen, verschiedenen Zielgruppen das gleiche Thema auf unterschiedliche Weise nahezubringen – angepasst an den jeweiligen Leser.

Abstract

Traditionally text is linear; one reads them from beginning to end. However, in recent decades hypertext introduced itself as another major type of text. Hypertext is structured as a number of interlinked text passages. This provides several distinct “reading paths” through a hypertext possible while linear text provides only one. However, hypertext creates its own challenges since a reader has to decide which links to follow. In extreme cases, this can lead to disruptions in reading because some linked text segments may not be coherent. This may affect the reader’s understanding of a text.

This work presents a hybrid between these two text types: dynamic text. Dynamic text allows, like hypertext, for different reading paths, or variants, but without compromising coherence and understanding of the text. Requirements and goals for these text variants are modeled by the author and can then be selected by the reader prior to reading the generated variant. Providing coherence and good understanding of the text is essentially the author’s task in this concept; thus a recommendation for an author’s approach to writing dynamic text is also part of this work.

Such texts can, for example, be used in teaching or interdisciplinary work groups where prior knowledge can vary greatly between individuals. Dynamic text can convey a single topic to readers of different backgrounds in the different ways necessary for their understanding.

Inhalt

Kapitel 1. Einleitung

1.1 Motivation und Thema

Seit Jahrhunderten verfassen Menschen schon Texte. Traditionell sind die meisten davon geradliniger Natur: Stein- und Tontafeln, Schriftrollen und Bücher beinhalten sogenannte lineare Texte, seien es Schauspiele oder Dichtungen, wissenschaftliche Artikel, Diplomarbeiten oder Romane. Sie werden von Anfang bis Ende gelesen und sind üblicherweise nicht dazu da, dass sie in beliebiger Reihenfolge oder unter Auslassung ganzer Abschnitte gelesen werden. Einem Autor erlaubt dies, den Text entsprechend zu strukturieren, da er gewisse Annahmen darüber treffen kann, was der Leser schon gelesen hat und was nicht.

> ⓘ Als **Text** wird jede Art von vorwiegend textueller Information bezeichnet. Dies kann auch andere Medien beinhalten, beispielsweise Bilder, die im Text eingebettet sind. Generell ist ein Text etwas, was dazu gedacht ist, gelesen zu werden und Informationen und Wissen zu vermitteln, üblicherweise aber in linearer Form.[1]

In jüngerer Zeit gibt es immer mehr Texte, die nicht einer rein linearen Struktur entsprechen, üblicherweise unter dem Begriff Hypertext zusammengefasst. Hypertext ist im Gegensatz zu linearen Texten kein geradliniger Textblock, der in einer Richtung gelesen wird, sondern stattdessen ein Netz von einzelnen Abschnitten, die untereinander durch sogenannte Hyperlinks verknüpft sind. Durch diese Hyperlinks kann der Leser selbst wählen, in welcher Reihenfolge er welche Abschnitte liest und es kann auch sein, dass einige Abschnitte dabei gar nicht gelesen werden.

Beide Textformen eignen sich für unterschiedliche Dinge. Es gibt sicher wenige Romane, die als Hypertext verfasst wurden, da der Leser gerade bei erzählten Geschichten den Ausführungen des Autors sehr genau folgen muss. Auf der anderen Seite sind Lexika und Nachschlagewerke selten in Form eines Buches, welches von Anfang bis Ende gelesen werden sollte. Beide Textformen eignen sich jeweils für unterschiedliche Anwendungen, was insbesondere in Abschnitt 2.1.2 noch genauer erläutert werden soll.

---

[1] Begriffe, die in der gesamten Arbeit konsistent und immer mit der gleichen Bedeutung verwendet werden, sind in einem solchen Rahmen als Definition kenntlich gemacht. Begriffe werden üblicherweise bei ihrer ersten Verwendung in dieser Form kurz definiert. Zusätzlich zu diesen Definitionen, die an geeigneter Stelle im Dokument erscheinen, findet sich in Anhang D ein Glossar mit allen im Dokument verwendeten Definitionen.

Bei linearen Texten hat der Leser wenig Kontrolle darüber, was er liest – dies obliegt allein dem Autor. Ein Leser, der den Großteil der Erläuterungen im Text schon kennt, wird sich eher langweilen, hat jedoch keine gute Möglichkeit, genau das zu überspringen, was er schon weiß. Auf der anderen Seite hat der Autor bei Hypertext nur noch geringen Einfluss auf den Weg des Lesers durch einen größeren Text. Aber gerade Leser, die nur wenig Hintergrundwissen eines Themas haben, können oft keine sinnvolle oder gute Entscheidung über die nächste zu folgende Verknüpfung treffen.

Gerade Lern- und Lehrmaterialien sowie Fachtexte sind aber oft darauf angewiesen, dass der Leser die Inhalte in einer sinnvollen und kohärenten Form präsentiert bekommt – anderenfalls leidet das Verständnis. Ebenso kann es nützlich sein, Leser mit verschiedenen Vorkenntnissen jeweils unterschiedlich (aber gezielt) an ein Thema heranzuführen, oder auch dem interessierten Leser zu erlauben, weitergehende Themen anzuschneiden.

> ⓘ **Inhalt** bezeichnet relativ abstrakt eine Menge an Informationen, die vermittelt werden soll. Dies kann in Form eines *Textes* geschehen, ist aber nicht darauf beschränkt. Im Rahmen dieser Arbeit bezeichnen Inhalte jedoch grundsätzlich Inhalte in Textform.

Bei interdisziplinären Inhalten ist es häufig der Fall, dass der Leser, je nachdem aus welchem Fachgebiet er kommt, mit jeweils anderen Teilen des Themas besser vertraut ist. Hier wäre es beispielsweise sinnvoll, einem Bioinformatiker mit Informatikhintergrund Teile der Biologie detaillierter zu erklären und umgekehrt.

> ⓘ Ein **Lesepfad** bezeichnet eine Sequenz von *Fragmenten* eines *Textes*, die der Reihenfolge nach gelesen werden. In Hypertexten, die mehrere Verzweigungsmöglichkeiten durch Hyperlinks haben, ergibt sich ein solcher Lesepfad durch die jeweils gewählten Verzweigungen.

Weder lineare noch Hypertexte sind in einem solchen Fall für sich genommen wirklich gut geeignet. Ideal wäre es, wenn die Vorteile von linearen Texten mit den Vorteilen von Hypertext verknüpft werden könnten; einerseits den „roten Faden“, der durch den Autor vorgegeben wird sowie andererseits die Möglichkeit verschiedener Lesepfade, um Leser mit verschiedenen Hintergründen für einen Text gewinnen zu können. Denkbar ist ein Hybrid zwischen beiden Textformen, der genau das erlaubt. Ein solcher Mittelweg ist Thema dieser Arbeit.

> ⓘ Eine **Textvariante** ist ein *Text*, der das gleiche Thema wie ein anderer Text behandelt, aber andere *Voraussetzungen* bzw. Annahmen hat. Beispielsweise kann ein Kinderbuch grob das gleiche Thema behandeln wie ein Schulbuch oder ein Artikel in einem Lexikon. In diesen Fällen sind jedoch die Voraussetzungen, die an den Leser gestellt werden, deutlich andere.

In einem solchen System können drei größere Kernprobleme aufgezeigt werden:

1. Es stellt besondere Anforderungen an einen Autor, da nicht mehr nur ein Text verfasst wird, sondern mehrere Varianten. Alle diese Varianten sollen immer noch gut lesbar und verständlich sein – ohne inhaltliche Brüche (siehe hierzu auch Kapitel 2.1.3).
2. Ein weiterer Bestandteil sind Softwaresysteme, um diese Texte anhand der Kriterien aus den einzelnen „Bausteinen“ zusammenzusetzen und um den Autor beim Erstellen solcher Texte zu unterstützen.
3. Zuletzt ist eine Evaluierung nötig, ob solche individuellen Texte tatsächlich besser zur Wissensvermittlung geeignet sind oder andere Vorteile haben gegenüber äquivalenten üblichen linearen oder Hypertexten.

In dieser Arbeit soll es insbesondere darum gehen, die Anforderungen an den Autoren näher zu untersuchen und methodische Hilfsmittel für die Strukturierung und Erstellung von Texten zu entwickeln.

1.2 Erste Ideen

2009 gab es unter den Autoren der deutschsprachigen Wikipedia umfangreiche und hitzige Debatten darüber, welche Themen, Personen oder sonstige Dinge eigene Artikel verdienen und was nicht innerhalb der Grenzen dessen ist, was Wikipedia erfassen soll (1). Während sich solche Debatten bis heute regelmäßig wiederholen, gäbe es im Grunde eine (relativ) einfache Lösung, die solche Fragen gar nicht erst aufkommen lässt. Wikipedia-Autoren, die diese Diskussionen führen, argumentieren im Wesentlichen in zwei Kategorien für neue Artikel: „behalten“ oder „nicht behalten“. Anstelle dieser sehr groben Einordnung wäre aber auch die Einstufung auf einer Skala von „enzyklopädisch“ bis „Trivialität“ denkbar. Dies würde einem Besucher der Seite ermöglichen, selbst zu wählen, was er wirklich sehen und lesen will. Die vorgeschlagene Skala wäre nur ein Beispiel gewesen, welches sich ungefähr mit den Argumentationslinien der Diskussionen deckt. Neue Artikel erreichen oft nicht den geforderten Grad an „Relevanz“, welche sich ungefähr an der Anzahl der vertrauenswürdigen Quellen für die Inhalte misst. Tatsächlich wären aber mit entsprechenden Filter- oder Sortiermechanismen zu viele Artikel prinzipiell kein Problem. Speicherplatz stellt heutzutage kein nennenswertes Problem mehr dar und „tote“ Artikel reduzieren, solange sie nicht gerade gut sichtbar verlinkt werden, nicht die wahrgenommene Qualität des Projektes[2]. Wird jedoch zu viel ausgelassen, so wird es wahrscheinlicher, dass einige Nutzer für sie relevante Informationen nicht finden.

Eine Weiterführung dieses Gedankens ist die Anwendung dessen auf Lehr- und Sachbücher. Beispielsweise lassen sich Themen finden, zu denen es sowohl sehr einfach gehalte-

[2] Beispielsweise könnten Artikel, die nicht lange genug sind oder wenige bis keine Quellenangaben haben, standardmäßig versteckt sein und nur für erfahrene Nutzer zugänglich sein.

ne und verständliche Kinderbücher, Lehrbücher für die Schule sowie tiefergehende Literatur für Studenten und Forschung gibt, wobei das grundlegende Thema in allen Fällen identisch ist und lediglich der Anspruch an den Leser variiert. Würden also solche Werke konsolidiert und in einem (digitalen) „Buch“ gebündelt werden, so könnte das Thema mit dem gleichen Ausgangsmaterial in verschiedenen Detailstufen angeboten werden um unterschiedliche Grade an Vorwissen anzusprechen. Ebenso würde es ermöglichen, dass interessierte Leser mehr lesen können als sie eigentlich sollten und somit ihren eigenen Interessen über die eigentlich zu vermittelnden Inhalte hinaus nachgehen können.

Gerade für diese Anwendung haben sich bei den ersten Ideen zu dieser Arbeit drei größere Punkte herausgebildet, die betrachtenswert sind. Diese sind am Ende des vorherigen Abschnittes schon erwähnt und erläutert worden.

1.3 Gliederung

In Kapitel 2 werden die zum Verständnis der Arbeit notwendigen grundlegenden Konzepte eingeführt und erläutert. Dies schließt die Betrachtung unterschiedlicher Textformen mit ein, sowie die schrittweise Entwicklung des in dieser Arbeit vorgestellten Themas. Weiterhin wird kurz auf textlinguistische Begriffe eingegangen.

Kapitel 3 geht auf bereits vorhandene, ähnliche Ansätze ein und stellt den Bezug sowie die Abgrenzung zu dieser Arbeit her.

Kapitel 4 befasst sich mit der Modellierung der in dieser Arbeit angesprochenen Textformen. Zunächst wird dafür ein Überblick über die im Rahmen dieser Arbeit erstellten Prototypen gegeben. Danach folgt die Betrachtung von Konzepten zur Modellierung des Textes in Form einer Datenstruktur, sowie zur Modellierung der Anforderungen an den Leser auf einer abstrakteren „Metaebene“.

In Kapitel 5 wird ein Vorschlag dargestellt, wie Autoren vorgehen könnten, solche dynamischen Texte zu verfassen. Dies geschieht unter Berücksichtigung der Modellierungskonzepte, die in Kapitel 4 eingeführt wurden.

Kapitel 6 befasst sich mit einer prototypischen Implementierung der in Kapitel 4 vorgestellten Modellierung.

In Kapitel 7 wird ein ausführlicheres Fallbeispiel zur Anwendung der Konzepte, die in Kapitel 4 vorgestellt wurden, gegeben. Anders als die anfänglichen Prototypen ist dies *nach* der Ausarbeitung der Konzepte entstanden und folgt demzufolge auch dem in Kapitel 5 vorgeschlagenen Arbeitsablauf.

Ein abschließender Ausblick auf Möglichkeiten der Weiterführung dieses Themas, sowie Gedanken und Konzepte, die in dieser Arbeit noch nicht betrachtet wurden, findet sich in Kapitel 8.

Kapitel 2. Hintergrund

2.1 Lineare Texte und Hypertext

Text ist im allgemeinsten Falle eine üblicherweise schriftliche Aneinanderreihung von Symbolen (Grapheme), denen in ihrem Zusammenhang eine Bedeutung zugeschrieben werden kann. Text ist das erste historische Archivmedium gewesen; mündliche Weitergabe von Wissen und Erzählungen ist mit dem Problem behaftet, dass mit jeder Weitergabe ein wenig Information verlorengehen kann, wie an dem Kinderspiel „Stille Post“ erkennbar ist. Dies muss zwar nicht zwangsläufig der Fall sein, jedoch gibt es für den Zuhörer keinerlei Möglichkeit der Überprüfung so dass geschriebener Text allgemein robuster in dieser Hinsicht ist.

2.1.1 Struktur der Textformen

Wie in Kapitel 1 schon kurz angesprochen, können zwei größere „Klassen“ von Texten unterschieden werden: lineare Texte und Hypertext. Während lineare Texte durch eine strikte Sequenz von Textabschnitten charakterisiert sind, ist Hypertext anders strukturiert. Der Begriff wurde 1965 von Ted Nelson geprägt (2):

> *“Let me introduce the word ‘hypertext’ to mean a body of written or pictorial material interconnected in such a complex way that it could not conveniently be presented or represented on paper. It may contain summaries, or maps of its contents and their interrelations; it may contain annotations, additions and footnotes from scholars who have examined it.”*

Hypertext bezeichnet also eine Struktur, in der aus einzelnen Textabschnitten ein semantisches Netz von Inhalten – ein Graph – konstruiert wird (3) – dies geschieht durch sogenannte Hyperlinks. Das bekannteste Beispiel heutiger Zeit ist zweifellos das World Wide Web (4), obwohl die grundsätzliche Idee schon deutlich älter ist. Die ersten Anfänge einer großen intern verknüpften Wissensbasis finden sich schon in den 30er und 40er Jahren des 20. Jahrhunderts (5). Und natürlich sind Lexika und Wörterbücher wenig anderes – Querverweise zwischen verwandten Einträgen und Artikeln sind im Grunde genommen Hyperlinks; sie lassen sich lediglich nicht ganz so einfach verfolgen wie heutzutage im WWW.

Das Lesen von Hypertexten gestaltet sich oft etwas anders als bei ihren linearen Gegenstücken. Einzelne Textabschnitte werden im Allgemeinen als solche gelesen, aber Hyper-

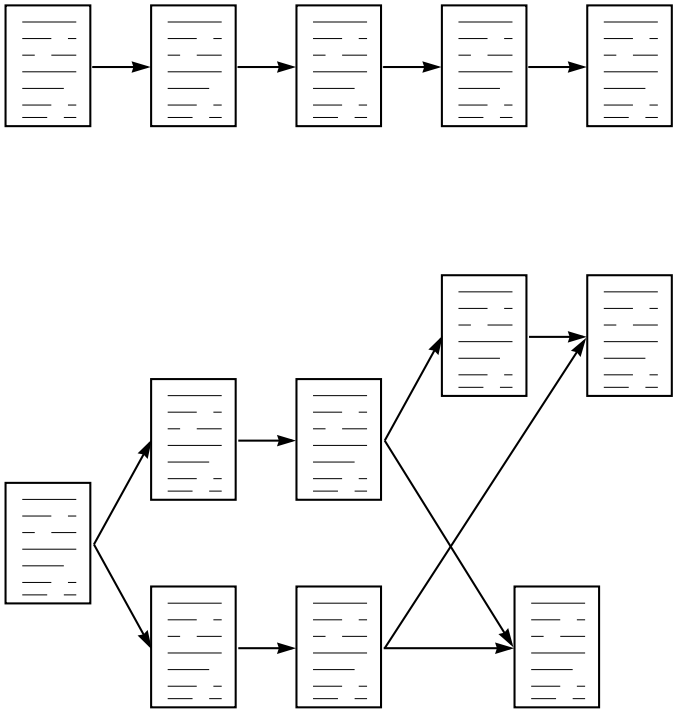


Abbildung 2.1 *Strukturelle Unterschiede zwischen linearem (oben) und Hypertext (unten). Entgegen dieser Darstellung ist Hypertext nicht zwangsweise ein azyklischer Graph. Abbildung nach einer Grafik aus (6).*

links ermöglichen dem Leser, sich selbst einen Lesepfad zu wählen. In der Theorie klingt dies sinnvoll, da auf diese Weise Hypertexte einfach mehr Möglichkeiten bieten als ihr lineares Gegenstück. Das Problem ist aber, dass ein *sinnvoller* Pfad für einen Leser nicht immer sichtbar ist – insbesondere, wenn der Leser mit dem Thema des Textes nicht ausreichend vertraut ist, um gute Entscheidungen zu treffen. In gewisser Weise mag argumentiert werden, dass der Leser geradezu dazu gezwungen wird, eine sinnvolle Entscheidung treffen zu müssen, auch wenn diese nicht immer ersichtlich ist (3).

Wie ein linearer Text und ein Hypertext im Sinne von aneinandergereihten Abschnitten aussehen können, zeigt Abbildung 2.1 exemplarisch. Während der lineare Text geradewegs von Anfang bis Ende gelesen wird, hat der Hypertext zwar auch nur einen „Einstiegspunkt“, aber von dort aus gibt es verschiedene Pfade durch die einzelnen Abschnitte und potentiell sogar verschiedene Endpunkte. Ebenso werden nicht zwangsweise alle Abschnitte besucht, je nachdem, welcher Pfad gewählt wird.

Lineare Texte, wie beispielsweise Bücher, sind normalerweise nicht so geschrieben und geplant, dass Abschnitte übersprungen oder gar beliebig zwischen Abschnitten gewechselt werden kann. Aber auch hier gibt es Ausnahmen. Beispielhaft sei hier ein Buch erwähnt, welches tatsächlich mehrere verschiedene Lesepfade erlaubt (allerdings ohne die relative Ordnung der Kapitel zueinander zu ändern) (7). Dieses Buch ist begleitend zu

einigen Vorlesungen, die der Autor an seiner Universität hält und definiert neben einem Abhängigkeitsgraphen der einzelnen Kapitel auch verschiedene Lesepfade, die den jeweiligen Vorlesungen entsprechen. Der erwähnte Abhängigkeitsgraph zwischen den Kapiteln findet sich in Abbildung 2.2. Im Buch folgen daraufhin vier mögliche Lesepfade, die sich an Inhalten orientieren, die der Autor in seinen Vorlesungen behandelt. Diese möglichen Lesepfade folgen den in der Abbildung dargestellten Abhängigkeiten, behandeln aber nur Teilmengen des gesamten Buches. Der Vollständigkeit halber ist das gesamte Buch ebenfalls ein Lesepfad, auch wenn dieser nicht explizit ausgewiesen ist.

Für Autoren sind lineare Texte seit langer Zeit bekannt. Die Strukturen solcher Texte und Methoden sie zu erfassen sind ausreichend gut erforscht. Die Struktur ist auch ein im-

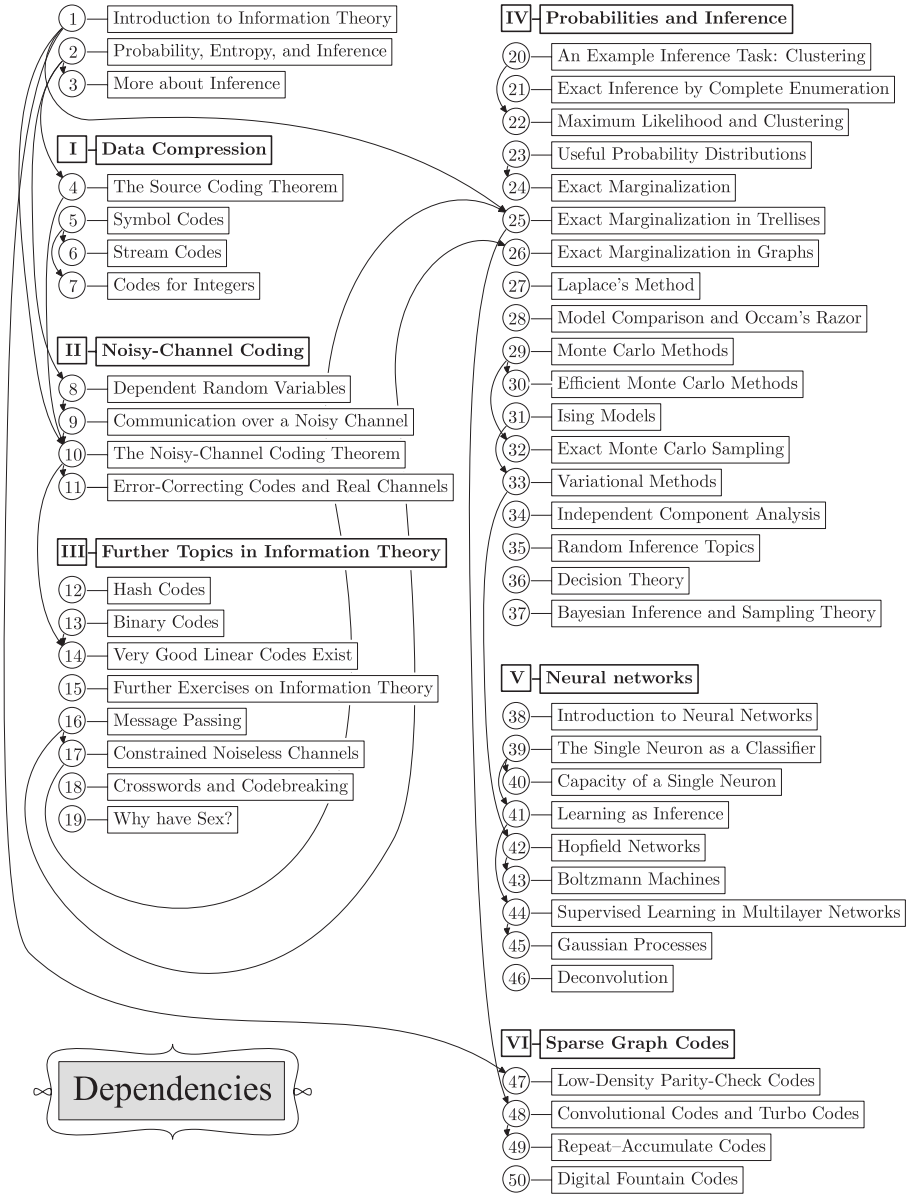


Abbildung 2.2 *Abhängigkeiten zwischen Kapiteln in (7).*

menser Vorteil für Texte, die den Leser an ein Thema heranführen sollen. Dadurch dass der Text linear von Anfang bis Ende gelesen wird, wird auch der Autor beim Schreiben geleitet. Er kann seine Argumentation entsprechend aufbauen und weiß immer, was der Leser schon kennt bzw. gelesen hat.

Hypertexte indes stellen ganz andere Herausforderungen. Da die Reihenfolge, in der Textabschnitte gelesen werden, nur teilweise vorgegeben ist, muss ein Autor deutlich vorsichtiger sein, Annahmen über die Leserschaft zu treffen. Durch bewusstes und geplantes Setzen von Hyperlinks ist der Lesefluss zwar in Grenzen steuerbar, aber selbst dann ist es schwierig, jeden möglichen Lesepfad vorauszuahnen und den Text entsprechend zu gestalten. Beispielsweise nutzen Leser eine Gliederung durchaus, um direkt zu Abschnitten zu springen, die sie lesen wollen – unabhängig davon, ob es Links gibt, die diese verbinden oder nicht.

2.1.2 Anwendungen

Die inhärenten Eigenschaften beider Textformen eignen sich jeweils für recht unterschiedliche Texte. Lineare Texte können Geschichten erzählen und Argumentationen in strukturierter Form aufbauen. Romane sind eigentlich immer dazu da, von Anfang bis Ende gelesen zu werden. In ähnlicher Weise werden mathematische Beweise nicht in einer Form geschrieben, die es ermöglicht, oder gar ermutigt, selbst auszusuchen, welche Abschnitte gelesen werden. Zum Verständnis ist immer das Ganze erforderlich und oft können einzelne Abschnitte nicht für sich genommen gelesen werden (auch wenn es natürlich immer Menschen gibt, die bei einem Roman die letzte Seite zuerst lesen). Sachtexte bilden hier aber häufig eine Ausnahme in dem Sinne, dass sie durchaus so geschrieben sind, dass Kapitel oder Abschnitte alleinstehend gelesen werden können.

Da Hypertext eine eher lose Struktur zugrundeliegt, sind einzelne zusammenhängende Abschnitte oft so geschrieben, dass sie für sich allein stehen können. Das macht Hypertext zu einem guten Medium für Nachschlage- und Referenzwerke, beispielsweise Wörterbücher, Lexika und Enzyklopädien. Hier ist die hauptsächliche Nutzung das Heraussuchen und anschließende Lesen eines einzelnen Abschnittes, der gerade von Interesse ist. Gelegentlich werden Querverweise zu verwandten Abschnitten verfolgt und ebenfalls gelesen, aber äußerst selten wird eine Enzyklopädie von Anfang bis Ende wie einen Roman gelesen werden.

Viele Texte fallen jedoch nicht in genau eine dieser beiden Kategorien. Tatsächlich ordnet sich das meiste an Inhalten irgendwo zwischen diesen beiden Extremen ein, wie in Abbildung *2.3* illustriert ist. Bücher haben Inhaltsverzeichnisse oder einen Index – beides Konzepte, die eher einer Reihe von Hyperlinks entsprechen und dem Leser erlauben, direkt zu bestimmten Stellen im Text zu springen. Ebenso gibt es gerade in Sachbüchern Querverweise zwischen Kapiteln, um verwandte Abschnitte zu referenzieren.

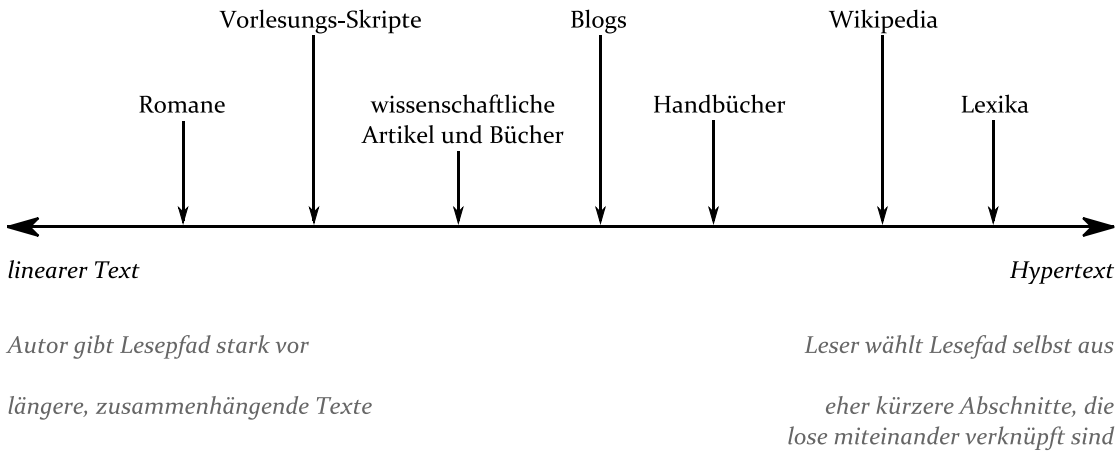


Abbildung 2.3 *Ungefähre Einordnung verschiedener Textarten und Anwendungen auf einer „Achse“ zwischen linearem und Hypertext.*

Auf der anderen Seite werden Hypertexte oft auch länger und ähneln inzwischen häufig linearen Texten, die lediglich sehr großzügig mit Hyperlinks versehen sind. Ein gutes Beispiel ist hier Wikipedia, welche im Vergleich zu einer „traditionellen“ Buch-Enzyklopädie deutlich längere Artikel hat. Einiges davon ist sicherlich dem geschuldet, dass eine rein digitale Enzyklopädie deutlich weniger Anforderungen an den Gesamtumfang hat, aber darin kann auch eine langsame Angleichung der oben beschriebenen Extreme gesehen werden.

Letztendlich aber werden Texte geschrieben, um gewissen Zielen zu genügen. Wenn diese Ziele es erforderlich machen, dass Leser sehr unterschiedliche Lesepfade durch den Text wählen, dann werden Abschnitte eher für sich selbst stehen statt in ihrer normalen Reihenfolge einen durchgängigen Argumentationsfluss zu bilden. Ebenso wird in einer Enzyklopädie auch nicht versucht, eine Argumentation durch mehrere miteinander verknüpfte Artikel zu ziehen.

2.1.3 Verständnis von Texten

Einleitend wurde schon eine kurze Definition des Textbegriffes gegeben. Diese soll hier verfeinert und erweitert werden. Während ein Text durchaus als zusammenhängende Folge von Symbolen klassifiziert werden kann, so reicht diese Definition nicht aus, wenn neben dem Text als geschlossene Einheit auch noch dessen Inhalt und Zusammenhalt betrachtet werden soll. Im Folgenden seien zwei Beispiele gegeben, die deutlich machen, dass eine nur oberflächliche Definition nicht genug ist (übernommen aus (8)):

> *„Die Frankfurter Feuerwehr hat ein Gerät vorgestellt, mit dem Menschen aus bis zu 200 Meter hohen Häusern gerettet werden können. Es ist eine mobile Seilbahn, die über am Haus befestigte Seile mit einer auf einem Lastwagen fahrbaren Gondel verbunden ist. Bisher sind Feuerwehrleitern maximal 30 Meter lang.“*
>
> *(aus: Die Welt vom 4. Juni 1980)*

> *„Ich habe leider nicht genug zu lesen. Die Kommission hat den Vorschlag abgelehnt. In den Ferien bleibt niemand gern zu Hause.“*

Im ersten Beispiel sind alle Sätze durch ein einheitliches Thema verbunden (das neue Rettungsgerät). Es wird genannt (im ersten Satz), erklärt und zu der vorherigen Methode abgegrenzt (die Feuerwehrleitern), was die drei Sätze in einer Weise verbindet, dass sie *Text* genannt werden können. Im zweiten Beispiel hingegen erscheinen die Sätze zusammenhanglos aneinandergereiht. Es gibt kein einheitliches Thema, welches die drei Sätze miteinander verbindet und einen Bezug zwischen ihnen herstellt. Im Grunde genommen könnte diese Satzfolge also als *Nicht-Text* bezeichnet werden. Klaus Brinker weist allerdings schon darauf hin, dass der Begriff „Nicht-Text“ keine objektive Größe bezeichnet. Vielmehr erschließt er sich mehr oder weniger intuitiv aus Sprache, Struktur und Inhalt, deren Zusammenhalt, sowie Verständnis des Lesers (8).

Eine grobe Definition des Alltagsbegriffes Text könnte wie folgt gegeben werden (8):

> *„ ‚Text‘ ist eine (schriftlich) fixierte sprachliche Einheit, die in der Regel mehr als einen Satz umfasst.“*

Bei Betrachtung des zweiten Beispiels wird allerdings deutlich, dass diese Definition keineswegs ausreicht, ist doch gegeben, dass es in Schriftform verfasst ist und mehr als einen Satz umfasst. Und doch soll jenen drei Sätzen nicht zugestanden werden, etwas darzustellen, was gemeinhin als *Text* bezeichnet werden würde.

In der Textlinguistik wird etwas weiter gegangen als bei der alltäglichen Definition des Textbegriffes. In der Linguistik haben Texte nicht nur eine Form, sondern auch eine Funktion, welche dadurch gegeben ist, dass ein Text als Kommunikationsmedium zwischen Autor und Leser dient. Weiterhin ist der inhaltliche Zusammenhang wesentlich. Um diese mit einzubeziehen gibt Brinker folgende Definition für *Text* (8):

> *„Der Terminus ‚Text‘ bezeichnet eine begrenzte Folge von sprachlichen Zeichen, die in sich kohärent ist und die als Ganzes eine erkennbare kommunikative Funktion signalisiert.“*

Kohärenz bezeichnet hier den inhaltlichen Zusammenhalt des Textes, der auch maßgeblich das Verständnis beeinflusst; hierauf wird im Folgenden aber noch detaillierter eingegangen.

Zu beachten ist bei obiger Definition, dass sprachliche Zeichen mitnichten Buchstaben sind, sondern ein Verbindungen von Bedeutung und Form, die im elementaren Fall Morpheme[3] und Wörter, aber auch komplexere Strukturen wie Wortgruppen und Sätze bezeichnen kann. Weiterhin wird hier wieder der Kohärenzbegriff gebraucht. Kohärenz und

[3] Ein *Morphem* bezeichnet die kleinste bedeutungstragende Einheit der Sprache, aus denen Wörter zusammengesetzt sind. Dies können zum Beispiel Wortstämme oder Endungen sein. Sie haben immer eine inhärente Bedeutung für das Wort in dem sie vorkommen.

Kohäsion bezeichnen zwei zentrale linguistische Konzepte, die ein Maß für den inhaltlichen und strukturellen Zusammenhalt eines Textes darstellen. Bei einigen Autoren werden diese Begriffe strikt getrennt, während bei Anderen *Kohärenz* für beide Konzepte verwendet wird, da diese ohnehin sehr eng miteinander verbunden sind (8). Der Begriff Kohärenz steht dann sowohl für strukturellen Zusammenhalt des Textes als auch den inhaltlichen Zusammenhalt durch Argumente. In dieser Arbeit soll Kohäsion nicht weiter betrachtet werden, stattdessen soll im Folgenden Kohärenz noch einmal genauer beleuchtet werden.

Nach Foltz (3) haben Texte, sowohl lineare als auch Hypertexte, immer den gleichen Anspruch: Das Vermitteln von Informationen in zusammenhängender Form an den Leser. Wenn ein Leser einen Text liest, möchte er gemeinhin hinterher noch wissen, was er gelesen hat und was der Text dem Leser dabei vermitteln wollte[4]. Kohärenz spielt dabei als Maß der Verständlichkeit eine wesentliche Rolle.

Ein Text präsentiert verschiedene durch Argumente gestützte Thesen und erzeugt so einen narrativen Fluss, durch den der Leser geführt wird. Überlappen sich Argumente ausreichend, so ist das jeweils vorhergehende noch im Kurzzeitgedächtnis des Lesers und er kann ohne Verständnisprobleme fortfahren zu lesen. Dies führt zu folgender Definition:

> ⓘ **Kohärenz** ist in einem *Text* das Maß, wie gut aufeinanderfolgende Argumente ineinandergreifen und sich überlappen. Hohe Kohärenz deutet auf starke Überlappung hin und erleichtert dem Leser so zumindest theoretisch das Verständnis des Textes. Wenig Kohärenz hingegen ist ein Zeichen dafür, dass sich Argumente im Text nicht ausreichend überlappen und damit das Verständnis erschweren.

In dem Fall, wo Argumente nicht mehr nahtlos ineinandergreifen, gibt es einen Bruch. Der Leser hat dann im Normalfall keinen offensichtlichen Pfad mehr und muss mit Hintergrundwissen aushelfen, sofern es vorhanden ist. Er muss eine Wissensbrücke zwischen dem letzten und dem folgenden Argument selbst finden. Passenderweise wird dies in (3) auch „Brückenfolgerung“ (*bridging inference*) genannt. Eine solche Folgerung ist also das Zurückgreifen auf vorhandenes Wissen des Lesers, um einen Bruch in der Argumentation aufzulösen. Reicht dies nicht aus, weil der Leser vielleicht mit dem Thema nicht ausreichend vertraut ist, so leidet das Verständnis des Textes. Aber selbst wenn das Hintergrundwissen des Lesers ausreicht, so erfordert es immer noch kognitiven Aufwand, um die Lücke zu schließen. Je weniger davon nötig ist, um den Ausführungen des Textes zu folgen, desto einfacher wird dem Leser das Verständnis fallen und umso besser kann er sich auf den Inhalt konzentrieren. Generell sollte vermieden werden, dem Leser Brückenfolgerungen aufzuzwingen, wenn ein Text primär zum Verständnis desselben geschrieben wird, was hier einmal angenommen wird.

---

[4] Dies gilt im Besonderen für Sach- und Fachliteratur. Belletristik soll im Rahmen dieser Arbeit nicht weiter betrachtet werden.

Bei linearen Texten hat der Autor im Wesentlichen Sorge dafür zu tragen, die Übergänge zwischen aufeinanderfolgenden Argumenten im Text so zu gestalten, dass eine Art „Lesefluss“ entsteht, dem der Leser folgen kann. Sinn ist hier, dass vom Leser eben nicht unnötigerweise kognitiver Aufwand erfordert wird, nur um der Argumentation zu folgen. Bei Hypertext hingegen obliegt diese Verantwortung dem Leser selbst – gerade dort, wo er Hyperlinks folgt. Einem Link selbst ist üblicherweise nicht anzusehen, wohin er genau führt und wie gut er den Kontext, in dem der Link gesetzt wurde, aufgreift. Online-Enzyklopädien wie Wikipedia haben zum Beispiel sehr viele Querverweise zu anderen Artikeln im Vergleich zu einer traditionellen Enzyklopädie. Da Links dort immer auf einen Artikel (oder teilweise auch einen Abschnitt im Artikel) verweisen, wird also grundsätzlich der Kontext des Ausgangsartikels verlassen und ein völlig neuer betreten. Wird einem Link mit einem bestimmten Ziel gefolgt, beispielsweise um eine kurze Begriffsdefinition zu finden, so muss aus dem verlinkten Artikel eben jenes Ziel erst einmal herausgelesen werden.

Um bei dem Beispiel zu bleiben: Da Wikipedia eine Enzyklopädie darstellen soll, ist dies kein großes Problem, da Enzyklopädien von Natur aus Referenzwerke sind. Es wird gezielt zu einem gesuchten Artikel gesprungen und dieser wird gelesen. Danach oder währenddessen wird eventuell einigen Querverweisen aus Interesse gefolgt und der Leseprozess ist irgendwann beendet. Es ist keine *Lernressource*, bei der ein Leser in sinnvoll strukturierter Form allmählich an ein Thema herangeführt wird. Wikipedia versucht dies auch gar nicht zu sein. Vielmehr hat jeder Artikel einige Voraussetzungen bezüglich Hintergrundwissens – fehlen diese, so können Artikel gelesen werden, die die nötigen Grundlagen behandeln. Dass also im Grunde an fast jedem Link eine Brückenfolgerung nötig ist, ist wenig überraschend und stört im Kontext der Wikipedia auch nicht. Für Lehr- und Lerntexte ist dies jedoch durchaus problematisch. Wenn ein Text gelesen wird, um etwas dabei zu lernen, dann ist alles an kognitivem Aufwand, der benötigt wird, um den Text auf einer Metaebene zu verstehen, nicht direkt zielfördernd. Metaebene bedeutet hierbei, dass das Verständnisproblem durch unzureichende strukturelle Unterstützung des Textes zustande kommt, anstatt durch tatsächliche Probleme beim Verständnis des Inhalts.

Foltz (3) merkt an, dass es keine messbaren Unterschiede im Verständnis desselben Inhaltes gibt, egal ob dies durch einen linearen oder Hypertext geschieht. Um dies zu überprüfen wurde ein linearer Text in einen Hypertext umgeschrieben, einzelne Abschnitte mit Hyperlinks verknüpft und eine verlinkte Struktur des Textes erzeugt. Der Autor gibt jedoch zu, dass gerade dieses Vorgehen dafür verantwortlich sein könnte, dass es keine sichtbaren Verständnisunterschiede gab. Dadurch, dass am Anfang ein linearer Text stand, war dieser schon gut strukturiert und geschrieben. Dies wurde dann natürlich auch mit in den Hypertext, der daraus erzeugt wurde, übernommen. Würde stattdessen ein Hypertext von Grund auf neu geschrieben werden – unter Berücksichtigung der Besonderheiten des Formates – so könnte das Ergebnis anders ausfallen. Er stellt jedoch auch

fest, dass Forschung darüber, wie Hypertext anders oder besser als lineare Texte strukturiert werden kann, noch selten ist.

2.2 Ziele dieser Arbeit

In dieser Arbeit soll es, wie in der Einleitung schon dargestellt, um dynamisch zusammengestellte Texte gehen. Dies schließt mögliche Modellierungen sowohl für eine Softwareimplementierung als auch für Texte durch den Autor mit ein. Ebenso soll eine prototypische Implementierung eines Autorenwerkzeuges erstellt werden. Ein wesentlicher Teil befasst sich mit möglichen Hilfen für den Autor solcher Texte.

> ⓘ **Voraussetzungen** sind die Menge an *Kriterien*, die für eine bestimmte *Textvariante* gelten.

Dem Autor kommt beim Verfassen ganzer Mengen von Textvarianten mehr Verantwortung zu als beim Schreiben lediglich eines einzelnen Textes. Er muss sich nicht nur Gedanken um eine einzelne, klar abgegrenzte Leserschaft machen, sondern um verschiedene solche Gruppen, teilweise mit sehr unterschiedlichen Voraussetzungen, Interessen und Zielen. Generell stellt dies den Autor vor ungewohnte Herausforderungen. Oft werden Texte mit unterschiedlichen Zielgruppen und Zielen nicht von einer Person verfasst – verschiedene Autoren schreiben verschiedene Inhalte. Das Verfassen solcher Texte mit mehreren Autoren soll allerdings im begrenzten Rahmen dieser Arbeit nicht betrachtet werden. Folglich wird sich im Folgenden nur noch auf den Fall mit einem Autor konzentriert.

Ein wesentlicher Punkt, den Autoren beachten müssen, ist die Struktur und der Zusammenhalt des Textes. Da das Ergebnis ein linearer Text sein soll, der gut lesbar und verständlich ist, müssen einzelne Abschnitte sinnvoll miteinander verbunden werden und dürfen nicht zu sehr für sich allein stehen. Weiterhin muss bedacht werden, dass das Ergebnis mehrere verschiedene Textvarianten sind, die sich je nach Natur des Textes entweder sehr stark oder nur wenig unterscheiden (siehe hierzu auch die Prototypen in den Abschnitten 4.2.1 und 4.2.2).

Basierend auf den ersten Ideen zu diesem Thema soll für diese Arbeit die Annahme getroffen werden, dass sich Textvarianten untereinander üblicherweise nicht zu sehr unterscheiden. Es wird zwar je nach vom Leser gewählten Kriterien mehr oder weniger Inhalt erklärt, je nach Kriterien die vom Leser angegeben werden, aber im Großen und Ganzen werden viele zugrundeliegende Textfragmente von mehreren Varianten verwendet werden.

> ⓘ Ein **Textfragment** ist ein zusammenhängendes Teilstück eines *Textes*. Dies kann ein Kapitel oder ein Abschnitt sein, ein Absatz oder gar nur ein einzelner Satz oder Satzteil. Diese werden in der im Rahmen dieser Arbeit vorgestellten Modellierung und Implementierung genutzt, um *Textvarianten* aufzubauen. Fragmente sind dann vorwiegend Teile, die in mehreren Varianten verwendet werden.

Natürlich sind die logische (inhaltliche) Struktur und der Grad der Wiederverwendung zwischen Varianten durchaus verschieden. Gerade wenn die Varianten separat verfasst worden sind und sich dadurch sehr stark unterscheiden, werden nicht viele Fragmente wiederverwendbar sein. Eine Datenstruktur für solche dynamischen Texte sollte in der Lage sein, verschiedene Grade von Wiederverwendbarkeit abzubilden. Ebenso sollte ein Autor nicht gezwungen sein, in einer bestimmten, unnatürlichen Weise zu arbeiten, nur weil dies Implementierungen einfacher macht oder besser auf eine gewählte Datenstruktur passt.

Möglicherweise braucht ein Autor, um eine solche Menge von Textvarianten schreiben zu können, auch Unterstützung durch Werkzeuge, die dabei helfen, die Inhalte sinnvoll zu strukturieren, aufzubereiten und miteinander zu verknüpfen. Denkbar sind hier Modellierungswerkzeuge, die dem Autor helfen, ein bestimmtes „Inhaltsmodell“ der Texte zu erstellen und einzuhalten. Ebenfalls möglich wären „Reflektionswerkzeuge“, welche dem Autor helfen, Inhalte und Struktur eines Textes zu betrachten um mögliche Modellierungen zu sehen.

Generell soll das Ziel sein, herauszufinden, wie ein Autor beim Schreiben von Texten mit verschiedenen Varianten unterstützt werden kann und wie Autoren solche Texte sinnvoll modellieren und strukturieren können. Die Anforderung verschiedener Varianten hat sicherlich einen Einfluss darauf, wie der Text strukturiert werden kann. Wenn Autoren einen Arbeitsablauf haben, der ihnen dabei hilft, führt dies vermutlich zu einem besseren Ergebnis.

Das Ziel eines solchen Systems sollte sein, einige spezifische Anwendungsmöglichkeiten zu unterstützen. Grob wurden diese in den vergangenen Abschnitten schon umrissen und angesprochen. Genauer schließt dies folgende mögliche Anwendungen mit ein.

Sich dem Leser anpassende Lehrtexte. Diese Anwendung bildete den Ursprung der hier vorgestellten Arbeit und sollte daher auch unterstützt werden. Dies bedeutet, dass es Lehr- und Lerntexte gibt, die bestimmte Inhalte vermitteln und vom Leser wählbare Kriterien unterstützen. Dies ermöglicht dem Leser, einen an seinen Kenntnisstand und seine Interessen angepassten Text zu erhalten bzw. auch über den eigentlich zu vermittelnden Lehrstoff hinaus zu lesen.

Texte mit verschiedenen Perspektiven auf ein Thema. Manchmal kann ein Thema unter verschiedenen Aspekten oder aus unterschiedlichen Perspektiven betrachtet werden. Dies kann einem Leser andere Sichtweisen eröffnen oder ihm ermöglichen, differenzierter über ein Problem nachzudenken. Beispiele hierfür, wenngleich in leicht anderem Kontext, finden sich in (9).

Iterative Vermittlung von Inhalten. Die Idee hierbei ist, einen Text mehrmals zu lesen, wobei er sich bei jedem Mal in Anpassung daran leicht ändert, was beim vorherigen Lesen an Information vermittelt und aufgenommen wurde. Dies würde einem Leser ermöglichen, ein Thema schrittweise zu vertiefen, je nachdem wie viel Zeit dafür verfügbar ist.

Jede dieser Anwendungsmöglichkeiten sollte nach Möglichkeit das gleiche zugrundeliegende Modell verwenden können. Das heißt nicht zwangsweise, dass jede der oben genannten Anwendungen mit dem gleichen Text umsetzbar sein muss, aber es sollten nicht verschiedene Modellierungen verwendet werden müssen, nur weil verschiedene Ziele angestrebt werden.

Kapitel 3. Bekannte Ansätze und Abgrenzung

Im Grunde genommen ist das Konzept, was in dieser Arbeit vorgestellt wird, nicht neu. Es gab in den letzten Jahren und Jahrzehnten verschiedene Ansätze, die ähnliche oder verwandte Probleme lösen. In diesem Abschnitt sollen einige davon kurz vorgestellt und von dem in dieser Arbeit Behandelten abgegrenzt werden.

3.1 Intelligente Lehrsysteme

Parallelen zum E-Learning drängen sich sicherlich auf, ist doch ein Aspekt und Ziel des in dieser Arbeit vorgestellten Konzeptes die Erstellung von adaptiven Lerninhalten, mit dem Ziel, den Lernenden zum einen besser anzusprechen und zum anderen auch potentiell für weitergehende Themen zu begeistern. Im E-Learning-Bereich gibt es ungefähr seit den 1970er Jahren Bestrebungen, die Lerninhalte individuell an den Lernenden anzupassen. Dies kam mit dem erhöhten Interesse an Künstlicher Intelligenz auf und es wurden Computer-Lernsysteme erhofft, die ähnlich einem menschlichen Lehrer auf den Lernenden eingehen können. Diese Systeme werden *Adaptive* oder *Intelligente Lehrsysteme* (ILS) genannt (*adaptive* bzw. *intelligent tutoring systems*).

Wie die Zeit seitdem gezeigt hat, ist *starke* künstliche Intelligenz nach wie vor utopisch, und menschenähnliches Verhalten eines solchen Systems nicht praktikabel. Was bleibt, sind Systeme, die auf *schwacher* künstlicher Intelligenz basieren, um den Lernenden und seinen Fortschritt zu beobachten und darauf eingehen zu können. Eine grobe Definition eines Intelligenten Lehrsystems könnte wie folgt lauten (10):

> *"Broadly defined, an intelligent tutoring system is educational software containing an artificial intelligence component. The software tracks students' work, tailoring feedback and hints along the way. By collecting information on a particular student's performance, the software can make inferences about strengths and weaknesses, and can suggest additional work."*

Intelligente Lehrsysteme verfolgen also, was der Lernende tut, geben Hinweise und „lernen" aus dessen Verhalten und Antworten, was ihnen erlaubt, auf Schwächen gezielter einzugehen. Hafner (10) gibt dazu ein Beispiel von einer Schülerin, die bei der Frage, was

−2,3 + 0,5 ergäbe, beständig 2,8 antwortet. Das ILS „erkennt“ hier die Art des Fehlers und weist sie darauf hin, das Vorzeichen zu beachten, was dann zur Lösung der Aufgabe führt.

Solche Lehrsysteme unterscheiden sich in zwei wesentlichen Punkten von den in dieser Arbeit vorgestellten Konzepten. Zunächst ist der Sinn eines ILS zwar durchaus, dem Lernenden etwas Freiheit zu gewähren und ihm zu erlauben, nach seinem eigenen Tempo zu lernen. Allerdings ist das Ziel immer, das zu vermitteln, was ein bestimmter Lehrplan vorsieht. Davon gibt es normalerweise wenig Abweichung und es wird versucht, den Lernenden in gewissen Grenzen um vorgesehene Lösungen zu halten (11). Der Einsatz des in dieser Arbeit vorgestellten Konzeptes beschränkt sich nicht ausschließlich auf Lern- und Lehrinhalte, auch wenn dies sicher passend und sinnvoll ist. Weiterhin soll der Fokus zwar schon darauf liegen, Wissen zu vermitteln, allerdings freier und mehr dem Leser überlassen, als dies bei ILS der Fall wäre.

Ein zweiter Punkt ist, dass ILS grundsätzlich interaktiv sind. Sie treten in ständigen Dialog mit dem Lernenden, um ein Modell seiner Kenntnisse zu erstellen und sich diesem anzupassen. Da das Ziel des hier vorgestellten Konzeptes ein gut lesbarer linearer Text sein soll, wird zunächst auf interaktive Elemente verzichtet. Die einzige direkte Interaktion mit dem Leser soll bei der Festlegung und Auswahl der Kriterien für die Auswahl der Textvariante im Voraus stattfinden. Ein Dialog zwischen System und Nutzer wie in ILS deckt sich nicht mit den Anforderungen an einen linearen Text zum Selbststudium. Auch Autoren müssen beim Verfassen von Inhalten für Intelligente Lehrsysteme den Aspekt der Interaktivität beachten. Das heißt, dass die Anforderungen an Autoren deutlich andere sind als bei dem hier vorgestellten Konzept.

3.2 IBIS

Ein Problem, welches in Abschnitt 2.1.3 schon angesprochen wurde, ist Kohärenz bei Hypertexten. Dadurch, dass der Benutzer gezwungen ist, über Hyperlinks einen guten Weg durch den Text zu finden, ist nicht gesichert, dass der dann gelesene Text kohärent ist, wodurch das Verständnis leiden kann. Foltz (3) spricht mit *argument-based hypertext systems* eine mögliche Alternative an:

> *“One exception to this problem of hypertext coherence may be found in argument-based hypertext systems. These systems take into account the role of coherence through only allowing jumps between nodes in which a coherent argument has been previously set between the two.* [...] *While argument-based hypertext involves a lot more hand crafting in order to create only these coherent links, it avoids the problems of readers jumping to nodes using links which may cause an incoherent transition.”*

Die Systeme, die der Autor hier anspricht, sind gIBIS (12) und JANUS (13). Beides sind Weiterentwicklungen von *issue-based information systems* (IBIS) (14).

Im Mittelpunkt von IBIS stehen Fälle (*issues*), welche mit *Argumenten* verknüpft sind, die entweder für oder gegen den Fall argumentieren (sowie eine Reihe weiterer Varianten). Der oben genannte Vorteil, dass es keine inkohärenten Sprünge gibt, obwohl letztendlich ein Hypertext zugrundeliegt, ergibt sich daraus relativ einfach: Die einzige Möglichkeit, Argumente im System anzulegen ist, dies von einem Fall (oder anderen Argument) aus zu tun. Weiterhin kann von einem Argument nicht zu einem völlig anderen eines anderen Falles gesprungen werden. Solange die Autoren der Inhalte das System tatsächlich so benutzen, wie es gedacht ist (und die Argumente nicht willkürlich einfügen), gibt es also tatsächlich nur kohärente Verknüpfungen, weil Fälle und Argumente jeweils aufeinander aufbauen.

Im Rahmen dieser Arbeit sind IBIS (und verwandte Systeme) allerdings keine sinnvolle technische Basis für die Umsetzung, da ein Autor gezwungen wäre, die Inhalte, aus denen er einen Text machen möchte, in Form von Fällen und Argumenten abzulegen und zu modellieren. Für Kollaboration, Diskurs, Diskussionen und Entscheidungsfindung sind IBIS durchaus ein nützliches Werkzeug, aber zur dynamischen Konstruktion linearer Texte anhand von Kriterien sind sie aufgrund der sehr starren Struktur nicht geeignet. Es gibt allerdings Parallelen zwischen dieser Arbeit und IBIS in dem Sinne, dass in beiden Fällen der Text auch auf einer Metaebene modelliert wird. Hierauf wird in Abschnitt 4.3 noch näher eingegangen.

3.3 Learning Objects

Learning Objects (LO), ist ein Ansatz, der zum Ziel hat, gemeinsam genutzte Lehrressourcen lediglich einmal zu erstellen und diese dann an unterschiedlichen Institutionen für die Lehre wiederzuverwenden (15). Gerade im Schulsystem gibt es schon solche zentral herausgegebenen und gemeinsam genutzten Ressourcen: Schulbücher sind nichts anderes, ebenso Landkarten, Periodensysteme, o. ä. für Klassenräume. Allerdings sind solche Dinge lediglich *Teile* von Fächern und Vorlesungen; selbst Lehrbücher werden selten vollständig und häufig auch mit anderen Materialien zusammen genutzt. Auf der anderen Seite werden Unterrichtsmaterialien oft in größerem Umfang vorbereitet – meist im Rahmen einer ganzen Vorlesung oder einer Klassenstufe.

Natürlich ist es nicht besonders sinnvoll, wenn jede Schule und jede Universität die jeweils gleichen Lerninhalte selbst aufbereitet. Dies verursacht erhebliche Kosten; hinzu kommt, dass die Autoren das Problem aus Sicht des Fernunterrichts angehen, wo die Materialien üblicherweise für den Onlinezugriff aufbereitet werden, was Aufwand und Kosten weiter erhöht. *Learning Objects* sollen hier wiederverwendbare, in sich abgeschlossene „Lerneinheiten“ sein, die zu Lehrveranstaltungen zusammengesetzt werden können. Diese bestehen aus *Inhalt* und *Metadaten*, die den Inhalt beschreiben (15):

> *“There are two major facets to authoring learning objects. The first is the content of the learning object itself; the second is the metadata describing the learning object. We might think of authoring learning objects as akin to authoring pieces of a puz-*

> *zle, in which case the content is the image or picture on the surface of the piece, while the metadata is the shape of the piece itself, which allows it to fit snugly with the other pieces."*

Learning Objects können sehr verschiedene Dinge darstellen; Downes (15) nennt als Beispiele Landkarten, Webseiten, Videos, interaktive Anwendungen – im Grunde alles, was in einer Lehrveranstaltung genutzt werden kann. Diese werden dann idealerweise in einer Markupsprache wie XML in einem standardisierten Schema umgesetzt, was erlaubt, den Inhalt in strukturierter Form abzulegen und die Metadaten direkt mit einzubetten. Entsprechende Software soll dann in der Lage sein, *Learning Objects* zu erzeugen, zu verarbeiten oder anzuzeigen. Ein vom Autor angesprochenes Beispiel ist die *Tutorial Markup Language* (TML), die im Grunde aus HTML, gemischt mit der semantischen Struktur des Lehrinhaltes in XML, besteht. Die Vorteile von XML sind in diesem Falle Unabhängigkeit vom Ausgabe- und Anzeigeformat oder -medium, womit die tatsächliche Struktur des Inhaltes beschrieben werden kann, statt die Präsentation.

Eine konkrete Umsetzung erfuhr dieses Konzept beispielsweise mit DocBook (16), wobei ebenfalls Bezug auf adaptive Inhalte genommen wurde (17). Die Autoren erläutern und demonstrieren, wie sich *Learning Objects* in DocBook (einer XML-Sprache) beschreiben lassen. DocBook war ursprünglich für das Verfassen von technischen Handbüchern gedacht, infolgedessen sehen die Autoren die so modellierten *Learning Objects* als Hand- oder Lehrbücher, die den Inhalt vermitteln sollen. Diese werden dann entsprechend im Dokumentenmodell von DocBook abgelegt. Der gesamte Veröffentlichungsprozess von DocBook basiert auf XSL-Stylesheets, die mit den Quelldaten arbeiten. Hier gibt es zwei größere Anwendungsmöglichkeiten: XSLT, eine Transformationssprache, die vorrangig XML-Daten manipuliert und in andere Strukturen umformt und XSL-FO, eine Formatierungssprache, die Dokumente für die Druckausgabe oder das Lesen am Bildschirm generiert. Der Veröffentlichungsprozess bei DocBook kann mit weiteren Stylesheets erweitert werden, die den Inhalt oder die Struktur der Dokumente verändern.

Spezifisch für adaptive Inhalte erlaubt DocBook sogenanntes *Profiling.* Dies ist ein Mechanismus, der einem erlaubt, mehr als eine Variante eines Dokumentes zu produzieren, wobei sich die einzelnen Varianten leicht unterscheiden (17). Hierbei werden zusätzliche Metadaten mit Strukturelementen aus dem Dokument verknüpft, die beschreiben, welche Varianten welchen Text beinhalten. Für die Umsetzung eben dieser Metadaten in die tatsächliche Strukturänderung, um die Varianten zu erzeugen, ist dann wieder ein XSLT-Stylesheet zuständig.[5]

Bezüglich dieser Arbeit sind weder *Learning Objects* noch ihre Umsetzung in DocBook direkt nutzbar. Die Gründe hierfür sind vielfältig. *Learning Objects* können grob mit den

[5] Diese Vorgehensweise unterscheidet sich nur wenig von dem in Abschnitt 4.2.1 und Anhang A beschriebenen Prototyp.

Textfragmenten verglichen werden, aus denen dann die eigentliche Textvariante zusammengesetzt wird. Allerdings gibt es hier gravierende Unterschiede. *Learning Objects* sind zwar dazu da, in einen größeren Kontext (beispielsweise eine Lehrveranstaltung) eingebettet zu werden, allerdings sind sie in sich auch abgeschlossen und austauschbar. Dies gilt für Textfragmente im Allgemeinen nicht, da diese oft kontextbezogen für bestimmte Varianten erstellt werden. Weiterhin werden Learning Objects aufgrund ihrer Natur sehr wahrscheinlich nicht von ihren ursprünglichen Autoren genutzt – LO sind dafür gedacht, einmal erstellt und dann weit verbreitet und wiederverwendet zu werden. In dieser Arbeit soll neben dem Leser lediglich ein Autor betrachtet werden, der Texte und Textvarianten verfasst und erstellt. Das heißt auch, dass der Autor derjenige ist, der die Textvarianten so aufbereitet, dass der Leser sie versteht und etwas daraus lernt. Bei dem *Learning-Objects*-Modell gibt es noch eine Zwischenperson und derjenige, der die LO erstellt, hat keine Kontrolle über das letztendliche Ergebnis.

Weiterhin sind *Learning Objects* kleine, nicht sinnvoll weiter unterteilbare Einheiten. In dem hier vorgestellten Konzept sind allerdings beide Kriterien nicht direkt anwendbar. Textfragmente sind nicht mehr weiter unterteilbar (oder zumindest im Rahmen der geplanten Varianten nicht weiter unterteilt worden), aber ob sie in sich abgeschlossen sind oder nicht hängt sehr vom Text und seinen Varianten ab. Textfragmente können im Extremfall eine ganze Variante umfassen, oder auch nur ein Kapitel, einen Abschnitt oder auch Teile eines Satzes.

Die technische Umsetzung dieses Konzeptes mit DocBook ist wohl prinzipiell möglich, soll allerdings hier aus den folgenden Gründen nicht weiter betrachtet werden. Zunächst ist es lediglich für die Unterstützung von Autoren beim Verfassen von Texten mit mehreren Varianten nicht nötig, sich auf eine technische Basis festzulegen. Dies ist eine Entscheidung, die für die praktische Umsetzung eines konkreten Produktes sinnvoll ist, nicht aber für theoretische Überlegungen oder prototypische Umsetzungen. Ein solcher Prototyp soll auch zunächst einfach gehalten werden, statt von Anfang an auf jede eventuell mögliche Nutzung einzugehen.

Martínez-Ortiz, et al. (17) deuten außerdem an, dass das `userLevel`-Attribut in DocBook die einzige Möglichkeit ist, dem Dokument Metadaten für adaptive Inhalte hinzuzufügen. Für simple Kriterien, wie beispielsweise eine einzige Dimension für den Anspruch an den Leser („Anfänger“, „Fortgeschritten“, „Experte“) ist dies noch möglich (ein Beispiel, welches ebenfalls dort angesprochen wird), aber für das in dieser Arbeit vorgestellte Konzept sollen auch komplexere Kriterien in mehreren Dimensionen möglich sein (siehe hierzu Kapitel 4). Potentiell würde dies nötig machen, dass der Autor selbst ein XSLT-Stylesheet schreiben und warten muss – je nach Komplexität der Kriterien und Anforderungen. Generell wäre es zu bevorzugen, wenn das System direkte Unterstützung für solch komplexere Problemfälle mitbringt, statt sie vom Autor lösen zu lassen. DocBook könnte durchaus ein *Ausgabe*format darstellen, aber für diese Arbeit, die auch nur nach einer prototypischen Umsetzung verlangt, wird es nicht in Betracht gezogen.

Kapitel 4. Modellierung

Für die Modellierung solcher dynamischen Texte gibt es zwei große Punkte: Zum einen die Modellierung für den Autor, der die Texte schreibt und in Varianten unterteilt, zum anderen die Modellierung als Datenstruktur für Umsetzungen in Software und Austausch der Daten. In diesem Abschnitt soll ein Überblick über den generellen Ansatz zur Modellierung, die Schwierigkeiten mit spezifischen Umsetzungen sowie mögliche Implementierungen gegeben werden.

4.1 Grundsätzliche Idee

Generell sollen die Texte auf zwei verschiedenen Ebenen verfasst und konzipiert werden. Eine Ebene hierbei ist die *Textebene*, in der die Textfragmente existieren, die hinterher zu den verschiedenen Textvarianten zusammengesetzt werden. Die zweite, höhere, Ebene ist eine *Metaebene*, die die verschiedenen Perspektiven auf bzw. Anforderungen an den Text beschreibt. Diese beinhaltet die im Abschnitt 4.3.1 noch genauer erwähnten Kriterien. Diese sollten nach Möglichkeit gruppiert oder anderweitig strukturiert werden können, um dem Leser eine Auswahl zu erleichtern. Auch ermöglicht dies, sofort zu erkennen, welche Art von Kriterien für einen Text ausgewählt werden können. Die Metaebene dient im Wesentlichen zur Gestaltung der Anforderungen an die Textvarianten (auf Basis der Kriterien) sowie zur Reflektion des Autors über seine Ziele, Zielgruppen und Intentionen beim Verfassen derselben.

Grundsätzlich sollen für einen Text mehrere Kriterien gleichzeitig wählbar sein – im Gegensatz zu den in Abschnitt 4.2 beschriebenen Prototypen, die jeweils nur eine Kriterien-„Dimension“ unterstützen, d. h. eine „Achse“, entlang derer genau eins der vorgegebenen Kriterien ausgewählt wird. Welche Dimensionen und spezifische Kriterien dies sind und welche Auswirkungen sie auf den Text haben, obliegt dem Autor. Dies können eher weitreichende und abstrakte Aspekte sein wie beispielsweise der Wissensstand oder allgemeine Vorkenntnisse des Lesers oder welchen ungefähren Umfang der Text haben soll. Ebenso gut können sie aber auch sehr spezifisch sein, wie beispielsweise eine bestimmte Programmiersprache, die vermittelt werden soll, bestimmte Konzepte, die dem Leser schon bekannt oder noch unbekannt sind, sprachliche Fertigkeiten (um beispielsweise Zitate im Text nicht in der Originalsprache, sondern übersetzt einzufügen), etc. Diese eher spezifischen Kriterien steuern sehr wahrscheinlich nur kleine, in sich abgeschlossene Teile der resultierenden Textvariante, während die allgemeineren Kriterien teilweise er-

heblichen Einfluss auf den gesamten Text haben können. Als Beispiel kann hier das oben schon erwähnte „Umfang“-Kriterium genannt werden, welches in einem Extremfall eine sehr grobe Übersicht in zwei Absätzen oder auf der anderen Seite ein 200-seitiges Dokument erzeugen könnte.

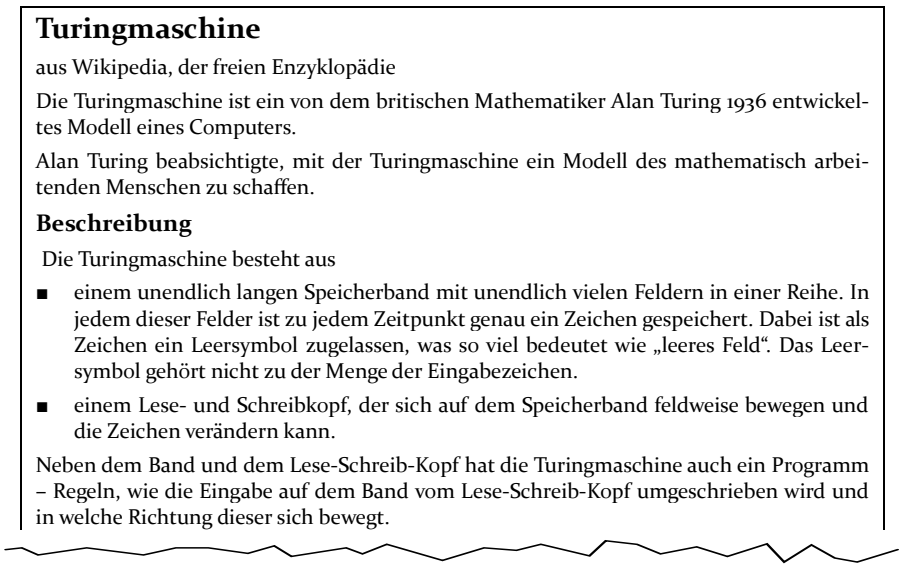
4.2 Erste Prototypen

Im Rahmen dieser Arbeit wurden zwei Prototypen erstellt um zu testen, wie solche Texte erstellt und praktisch umgesetzt werden können. Dies war zum einen der Wikipedia-Artikel „Turingmaschine“ (18), der in drei verschiedene „Anspruchsebenen“ aufgeteilt wurde. Zum anderen wurden drei eigenständige Texte geschrieben, die eine Einführung in die Programmiersprache Java darstellen, jeweils für Leser ohne jegliche Programmiererfahrung, für Leser, die die Programmiersprache C beherrschen und für Leser, die die Programmiersprache C# beherrschen. Im Folgenden werden diese beiden Prototypen kurz beschrieben.

4.2.1 Wikipedia-Artikel „Turingmaschine“

Ausgangspunkt hier war der Artikel „Turingmaschine“ aus der Wikipedia (18). Dabei wurden aus dem Originaltext drei verschiedene Anspruchsebenen herausgearbeitet, die ungefähr Schulwissen und dem Wissen eines Informatikstudenten nach dem ersten bzw. vierten Semester entsprechen. Weiterhin wurde eine praktische Umsetzung für die Anzeige und Auswahl dieser Texte erstellt.

Für den Schultext wurde fast komplett auf informelle Beschreibungen zurückgegriffen und keinerlei Definitionen aus der Mathematik oder theoretischen Informatik verwendet. Erstsemesterstudenten sind üblicherweise schon mit Mathematik vertraut, aber nicht so



Turingmaschine

aus Wikipedia, der freien Enzyklopädie

Die Turingmaschine ist ein von dem britischen Mathematiker Alan Turing 1936 entwickeltes Modell eines Computers.

Alan Turing beabsichtigte, mit der Turingmaschine ein Modell des mathematisch arbeitenden Menschen zu schaffen.

Beschreibung

Die Turingmaschine besteht aus

- einem unendlich langen Speicherband mit unendlich vielen Feldern in einer Reihe. In jedem dieser Felder ist zu jedem Zeitpunkt genau ein Zeichen gespeichert. Dabei ist als Zeichen ein Leersymbol zugelassen, was so viel bedeutet wie „leeres Feld“. Das Leersymbol gehört nicht zu der Menge der Eingabezeichen.
- einem Lese- und Schreibkopf, der sich auf dem Speicherband feldweise bewegen und die Zeichen verändern kann.

Neben dem Band und dem Lese-Schreib-Kopf hat die Turingmaschine auch ein Programm – Regeln, wie die Eingabe auf dem Band vom Lese-Schreib-Kopf umgeschrieben wird und in welche Richtung dieser sich bewegt.

Abbildung 4.1 *Anfang des aufbereiteten Wikipedia-Artikels „Turingmaschine“ ungefähr auf Schulniveau. Einleitung sowie Beschreibung sind sehr informell und einfach gehalten.*

sehr mit den Konzepten formaler Sprachen und Automaten, folglich wurden zwar einige Formalien eingeführt, jedoch nicht die komplette formale Definition und Funktionsweise einer Turingmaschine. Der letzte Text, der mit dem Wissen aus dem 4. Semester eines Informatikstudiums verständlich sein sollte, wurde fast unverändert aus dem Ausgangstext übernommen. Die beiden anderen Texte entstanden im Wesentlichen durch Streichen von Abschnitten und Textpassagen. Für einen Prototyp wie diesen ist dies noch akzeptabel, aber für realistische Texte ist das sicher keine gangbare Lösung da der Inhalt dadurch sehr reduziert wird.

Ein Beispiel sind die einleitenden Sätze des Artikels (siehe Abbildung *4.1*):

> *„Die Turingmaschine ist ein von dem britischen Mathematiker Alan Turing 1936 entwickeltes Modell, um eine Klasse von berechenbaren Funktionen zu bilden. Sie gehört zu den grundlegenden Konzepten der Informatik.“*

Turingmaschine

aus Wikipedia, der freien Enzyklopädie

Die Turingmaschine ist ein von dem britischen Mathematiker Alan Turing 1936 entwickeltes Modell eines Computers. Sie gehört zu den grundlegenden Konzepten der Informatik.

Das Modell wurde im Rahmen des von David Hilbert im Jahr 1920 formulierten Hilbertprogramms vorgestellt. Alan Turing beabsichtigte, mit der Turingmaschine ein Modell des mathematisch arbeitenden Menschen zu schaffen.

Das Besondere an einer Turingmaschine ist, dass sie mit nur drei Operationen (Lesen, Schreiben und Schreib-Lese-Kopf bewegen) alle Probleme lösen kann, die auch von einem Computer gelöst werden können. Sämtliche mathematischen Grundfunktionen wie Addition und Multiplikation lassen sich mit diesen drei Operationen simulieren. Darauf aufbauend kann man dann komplexe Operationen der üblichen Computerprogramme simulieren.

Informelle Beschreibung

Die Turingmaschine besteht aus

- einem unendlich langen Speicherband mit unendlich vielen sequentiell angeordneten Feldern. In jedem dieser Felder ist zu jedem Zeitpunkt genau ein Zeichen gespeichert. Dabei ist als Zeichen ein Leersymbol zugelassen, was so viel bedeutet wie „leeres Feld“. Das Leersymbol gehört nicht zu der Menge der Eingabezeichen. Man darf sich das

Variationen des Turingmaschinen-Modells

Universelle Turingmaschine

In der obigen Definition ist das Programm fest in die Maschine eingebaut und kann nicht verändert werden. Man kann aber eine universelle Turingmaschine definieren, welche die Kodierung einer Turingmaschine als Teil ihrer Eingabe nimmt und das Verhalten der kodierten Turingmaschine auf der ebenfalls gegebenen Eingabe simuliert. Aus der Existenz einer solchen universellen Turingmaschine folgt zum Beispiel die Unentscheidbarkeit des Halteproblems. Eine ähnliche Idee, bei der das Programm als ein Teil der veränderbaren Eingabedaten betrachtet wird, liegt auch fast allen heutigen Rechnerarchitekturen zugrunde (Von-Neumann-Architektur).

Ameise

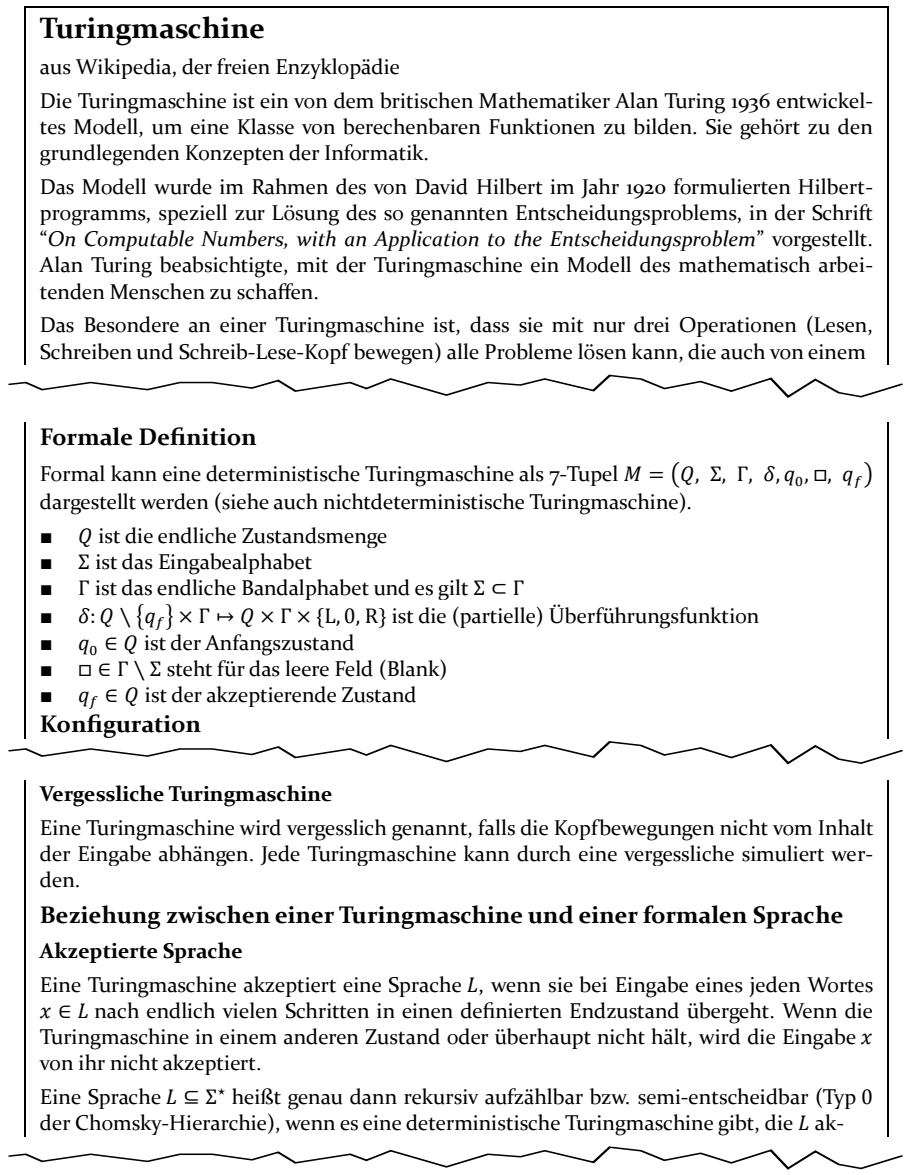
Chris Langtons Ameise ist eine Turingmaschine mit zweidimensionalem Band, sehr ein-

Abbildung 4.2 *Anfang und Ende des aufbereiteten Wikipedia-Artikels „Turingmaschine“ ungefähr auf Erstsemesterniveau. Im Vergleich zum Schultext wurde die Einleitung ausgebaut und am Ende werden zusätzlich noch Variationen des traditionellen Modells erwähnt.*

Ist nur wenig Vorwissen theoretischer Informatik vorhanden, so ist der Begriff „berechenbare Funktion“ völlig fremd. Für die Erstsemester- und Schulvariante wurde dieser Satz demzufolge umgeschrieben:

> *„Die Turingmaschine ist ein von dem britischen Mathematiker Alan Turing 1936 entwickeltes Modell eines Computers.“*

Diese Variante reduziert das nötige Wissen von *„Grundlagen der theoretischen Informatik“* auf *„der Leser weiß ungefähr, was ein Computer ist“*. An Inhalt büßt der Satz dabei nur wenig ein.



Turingmaschine

aus Wikipedia, der freien Enzyklopädie

Die Turingmaschine ist ein von dem britischen Mathematiker Alan Turing 1936 entwickeltes Modell, um eine Klasse von berechenbaren Funktionen zu bilden. Sie gehört zu den grundlegenden Konzepten der Informatik.

Das Modell wurde im Rahmen des von David Hilbert im Jahr 1920 formulierten Hilbertprogramms, speziell zur Lösung des so genannten Entscheidungsproblems, in der Schrift "*On Computable Numbers, with an Application to the Entscheidungsproblem*" vorgestellt. Alan Turing beabsichtigte, mit der Turingmaschine ein Modell des mathematisch arbeitenden Menschen zu schaffen.

Das Besondere an einer Turingmaschine ist, dass sie mit nur drei Operationen (Lesen, Schreiben und Schreib-Lese-Kopf bewegen) alle Probleme lösen kann, die auch von einem

Formale Definition

Formal kann eine deterministische Turingmaschine als 7-Tupel $M = (Q,\ \Sigma,\ \Gamma,\ \delta, q_0, \square,\ q_f)$ dargestellt werden (siehe auch nichtdeterministische Turingmaschine).

- Q ist die endliche Zustandsmenge
- Σ ist das Eingabealphabet
- Γ ist das endliche Bandalphabet und es gilt $\Sigma \subset \Gamma$
- $\delta\colon Q \setminus \{q_f\} \times \Gamma \mapsto Q \times \Gamma \times \{\mathrm{L}, 0, \mathrm{R}\}$ ist die (partielle) Überführungsfunktion
- $q_0 \in Q$ ist der Anfangszustand
- $\square \in \Gamma \setminus \Sigma$ steht für das leere Feld (Blank)
- $q_f \in Q$ ist der akzeptierende Zustand

Konfiguration

Vergessliche Turingmaschine

Eine Turingmaschine wird vergesslich genannt, falls die Kopfbewegungen nicht vom Inhalt der Eingabe abhängen. Jede Turingmaschine kann durch eine vergessliche simuliert werden.

Beziehung zwischen einer Turingmaschine und einer formalen Sprache

Akzeptierte Sprache

Eine Turingmaschine akzeptiert eine Sprache L, wenn sie bei Eingabe eines jeden Wortes $x \in L$ nach endlich vielen Schritten in einen definierten Endzustand übergeht. Wenn die Turingmaschine in einem anderen Zustand oder überhaupt nicht hält, wird die Eingabe x von ihr nicht akzeptiert.

Eine Sprache $L \subseteq \Sigma^*$ heißt genau dann rekursiv aufzählbar bzw. semi-entscheidbar (Typ 0 der Chomsky-Hierarchie), wenn es eine deterministische Turingmaschine gibt, die L ak-

Abbildung 4.3 *Anfang, Mitte und Ende des aufbereiteten Wikipedia-Artikels „Turingmaschine“ auf Viertsemesterniveau. Die Einleitung erwähnt nun die ursprüngliche Veröffentlichung von Turing. Weiterhin wird eine formale Definition des Modells gegeben und gegen Ende ebenso Bezug auf andere Gebiete der theoretischen Informatik genommen.*

Etwas weiter im Artikel wird der Bezug zur theoretischen Informatik, sowie dazu, dass berechenbare Funktionen das abbilden, was ein Computer leisten kann, hergestellt:

> *„Das Besondere an einer Turingmaschine ist, dass sie mit nur drei Operationen (Lesen, Schreiben und Schreib-Lese-Kopf bewegen) alle Probleme lösen kann, die auch von einem Computer gelöst werden können. Sämtliche mathematischen Grundfunktionen wie Addition und Multiplikation lassen sich mit diesen drei Operationen simulieren. Darauf aufbauend kann man dann komplexe Operationen der üblichen Computerprogramme simulieren. Eine Funktion, die so durch eine Turingmaschine berechnet werden kann, nennt man eine turingberechenbare Funktion.* […]*“*

Der zweite Teil dieses Absatzes (hier gekürzt, beginnend mit dem Verweis auf turingberechenbare Funktionen) fehlt in der Schul- und Erstsemestervariante, da zum Verständnis mehr Wissen über theoretische Informatik nötig ist. Ebenso fehlt die Literaturliste am Ende in der Schul- und Erstsemestervariante, da alle genannten Quellen ohne Kenntnisse der theoretischen Informatik nicht verständlich wären. Der erste Teil des Absatzes hingegen findet sich auch in der Textvariante für Erstsemester.

In ähnlicher Weise gibt es im Originaltext einen Abschnitt „Informelle Beschreibung“, dessen Titel nur dann einen Sinn ergibt, wenn der Leser den Unterschied zu einer formalen Beschreibung kennt oder diese in einem weiteren Abschnitt gegeben wird. Folglich ändert sich der Abschnittstitel lediglich auf „Beschreibung“ in der Schulvariante, welche dann auf eine formale Beschreibung des Modells vollständig verzichtet.

Beispiele sowie die informelle Beschreibung der Funktion einer Turingmaschine hingegen sind in allen drei Varianten möglich, da sie problemlos zu verstehen sind.

Dadurch, dass die Textvarianten hier durch Entfernen von Textabschnitten erzeugt wurden, waren beide Texte mit geringerem Anspruch kürzer als der dritte – ob dies sinnvoll ist, hängt ein wenig vom verfolgten Ziel ab: Soll grundsätzlich der gleiche Inhalt vermittelt werden, so liegt es nahe, dass Texte, die mit weniger Vorkenntnissen verstanden werden sollen, *länger* werden. Schließlich müssen die Voraussetzungen für den vollen Text auch noch erklärt werden. Soll hingegen, ohne auf sämtliche Details einzugehen, ein Überblick über das Thema gegeben werden, so kann natürlich ein Text, der mehr Wissen voraussetzt, auch länger werden, weil er mehr behandeln kann. Dies kann letztendlich aber auch ein wählbares Ziel des Lesers sein, welches bei der Zusammenstellung des zu lesenden Textes dann berücksichtigt werden kann.

Technisch wurde der hier dargestellte Prototyp mit HTML, CSS und JavaScript umgesetzt, da somit ein beliebiger Web-Browser zum Betrachten genutzt werden kann. Einzelne Abschnitte, Sätze und Satzteile wurden einer oder mehreren der drei „Anspruchsebenen“ zugewiesen. Von diesen Ebenen wurden dann zwei versteckt und nur eine angezeigt, um einen der drei Texte zu erhalten. Der JavaScript-Quelltext der Implementierung findet sich in Anhang A.

Die Trennung in Meta- und Textebene erfolgte bei diesem Prototyp nur relativ vage, nicht zuletzt, weil die Erstellung desselben der konkreten Aufgabenstellung dieser Arbeit um Monate vorausging. Die Metaebene beschränkte sich darauf, drei Anspruchsebenen herauszuarbeiten; diese sind im JavaScript-Quelltext festgeschrieben. Das Ziel war ein Prototyp und keine allgemein anwendbare Lösung. Während die Metaebene programmatisch mittels JavaScript umgesetzt wurde, ist die Textebene vollständig im HTML-Quelltext enthalten. Eine explizite und bewusste Modellierung beider Ebenen blieb allerdings aus oben genannten Gründen aus.

Die drei resultierenden Textvarianten sowie die in den Abbildungen *4.1*, *4.2* und *4.3* verwendeten Ausschnitte stehen aufgrund der Lizenzbestimmungen von Wikipedia unter der Lizenz *CC BY-SA 3.0*[6].

4.2.2 Java lernen mit verschiedenen Vorkenntnissen

Ein zweiter Prototyp – allerdings nicht technisch, sondern nur inhaltlich umgesetzt – ist eine Reihe dreier Texte, die einen Einstieg in die Programmiersprache Java vermitteln sollen. Während bei dem Turingmaschinen-Artikel ein bereits fertiger Text in verschiedene Textvarianten umgeschrieben wurde, so wurden die drei Texte hier jeweils von Grund auf neu geschrieben. Während bei dem vorherigen Beispiel nur drei grob umrissene „Anspruchsebenen“ herausgearbeitet wurden, wurde in diesem Falle detaillierter spezifiziert, welche Vorkenntnisse für die jeweilige Variante nötig sind und welche „Wissenslücken“ angenommen wurde – also Wissen, welches durch den Text erläutert wird und demzufolge nicht als Voraussetzung nötig ist.

Die erste der drei Textvarianten richtet sich an komplette Anfänger, die nie eine Programmiersprache verwendet haben. Anders als andere Spracheinführungen wird vollständig auf eine Erklärung des Compilers verzichtet; Sinn hierbei war, den Leser erst an die Sprache heranzuführen ohne einen Computer zu benötigen oder den Code gleich zu testen. Dies vereinfachte das Schreiben der Einführung und vermittelt gleichzeitig die Fähigkeit, Code auf dem Papier zu schreiben und zu durchdenken statt auszuprobieren. Demzufolge ist das einzig nötige Vorwissen ein wenig elementare Mathematik (für eine der Übungen) und die Fähigkeit, Text zu Papier zu bringen.

Die zweite Variante richtet sich an Leser, die bereits Erfahrung mit der Programmiersprache C haben, nicht jedoch mit objektorientierter Programmierung. Da sich Javas grundlegende Syntax sehr an C++ orientiert, welches wiederum auf C aufbaut, sind zumindest die Kontrollstrukturen identisch und müssen nicht sehr detailliert behandelt werden. Da Java aber im Gegensatz zu C eine objektorientierte Programmiersprache ist, sollten objektorientierte Konzepte natürlich erwähnt und erklärt werden.

[6] http://creativecommons.org/licenses/by-sa/3.0/de/

Die dritte Variante geht von Lesern aus, die schon mit der Programmiersprache C# vertraut sind. C# ist Java sehr ähnlich, da beide objektorientierte Programmiersprachen mit automatischer Speicherverwaltung sind, die auf einer virtuellen Maschine mit umfassender Klassenbibliothek laufen. Daher wird Wissen über objektorientierte Programmierung und Konzepte ebenfalls angenommen. Was in dieser Variante jedoch ausführlicher behandelt werden muss, sind die Unterschiede zwischen beiden Sprachen – nicht zuletzt dadurch, dass sie sich seit dem Erscheinen von C# im Hinblick auf Sprachfeatures immer weiter voneinander entfernt haben. Viele Muster, die in C# idiomatisch und auch guter Stil sind, sind in Java unmöglich und ein vollständigerer Text sollte eventuell auch auf Unterschiede der jeweiligen Klassenbibliotheken eingehen.

Alle drei Texte wurden verhältnismäßig kurz gehalten, da es sich nur um eine weitere prototypische Exploration möglicher Konzepte und Ideen handelte. Jeder Text ist kürzer als zwei Seiten. Wären sie ausführlicher geschrieben worden, so wären sicher einige Textabschnitte mehrfach nutzbar gewesen. Beispielsweise wird sich die Anfängervariante an einem gewissen Punkt auch mit objektorientierten Konzepten befassen müssen; der Abschnitt dafür ließe sich vermutlich mit dem aus der C-Variante vereinen, möglicherweise mit kleinen Änderungen. Bislang gibt es lediglich zwei gemeinsam genutzte Abschnitte – jeweils Programmbeispiele (mehr hierzu in Abschnitt 4.3.2).

Weiterhin könnten die vorhandenen drei Textvarianten Anfangspunkt für weitere „Untervarianten“ sein, die sich nur in Details und kleineren Teilen unterscheiden. Beispielsweise kann die Anfängervariante in zwei Varianten aufgeteilt werden, wovon eine gleich von Anfang an die Nutzung eines Compilers mit einbezieht und Hinweise darauf gibt, wie der Code, der für die Übungsaufgaben geschrieben wird, auch gleich ausprobiert werden kann.

Zusätzliche Varianten, gerade für die Texte, die von C- bzw. C#-Vorkenntnissen ausgehen, könnten für verschiedene Anwendungsmöglichkeiten erstellt werden, beispielsweise grafische Benutzeroberflächen, XML oder andere Nutzungen von vorhandenen Bibliotheken und Frameworks, für die Java eigene Lösungen anbietet. Diese sollten dann allerdings jeweils Erweiterungen der ursprünglichen Texte sein, statt sie komplett zu ersetzen.

Die drei erstellten Textvarianten finden sich vollständig in Anhang B.

In diesem Prototyp lag der Fokus darauf, die elementaren Anforderungen an den Leser in der Metaebene herauszuarbeiten, statt sie nur vage zu definieren. Auf der anderen Seite wurde die Textebene erheblich vereinfacht, da sie nur mehr drei große Textstücke umfasst, die jeweils zu den drei Varianten gehören.

4.3 Modellierungskonzepte

Die grundsätzlichen Anforderungen sind schon in Abschnitt 4.1 beschrieben worden. Eine Umsetzung und ein einheitliches Modell sollten mindestens die bisherigen Ideen und Prototypen umsetzen können. Weiterhin ist es wünschenswert, dass der Autor durch die

Modellierung und eine Softwareumsetzung Hilfestellung beim Verfassen und Strukturieren dynamischer Texte bekommt. Da sich solche Hilfestellung allerdings ohne längere Benutzertests mit Autoren, die das Konzept nutzen, nicht nachweisen lässt, muss der Nachweis dafür in dieser Arbeit offen bleiben. Diese Anforderungen werden damit zwar mit in die Überlegungen einbezogen, aber sie bleiben Hypothesen und sind möglicherweise nicht allgemeingültig.

Wie eingangs erwähnt, soll die Modellierung auf zwei verschiedenen Ebenen – einer Metaebene und einer Textebene – erfolgen. Auf diese wird in den beiden folgenden Abschnitten näher eingegangen. Als bisherige Arbeitsgrundlage existieren nur die beiden oben genannten Prototypen. In diesem Sinne werden die Modellierungskonzepte sich eng an dem orientieren, was in den Prototypen nötig und sinnvoll ist. Weiterhin hängen sowohl die Metaebene als auch die Textebene in gewissem Maße voneinander ab, weshalb bei der Herleitung der Modellierung oft ein wenig auf die jeweils andere Ebene eingegangen werden muss.

4.3.1 Metaebene

Die Metaebene umfasst die Anforderungen und Kriterien, nach denen eine Textvariante für den Leser erzeugt wird. Außerdem kann sie dem Autor als Erinnerung über die Anforderungen an die verschiedenen Textvarianten sowie als Strukturierungshilfe und Analysewerkzeug von Texten dienen. In der Tat soll diese Metaebene das hauptsächliche Hilfsmittel zur Planung der Textvarianten für den Autor sein.

Wesentliches Element der Metaebene sind Kriterien, die zur Komposition der Textvarianten genutzt werden. Ein Kriterium soll hier zunächst grob als *„wählbares Attribut einer Textvariante“* betrachtet werden. Diese Betrachtungsweise ist insbesondere der Fall, wie es noch um die Prototypen geht, die erstellt wurden bevor die hier noch vorzustellenden Konzeptideen erarbeitet wurden. Im weiteren Verlauf dieses Abschnittes wird noch eine differenziertere Definition erarbeitet, die aber für die Herleitung und Erarbeitung der Konzepte anhand der Prototypen noch nicht hilfreich ist.

Zunächst soll betrachtet werden, welche Kriterien in den beiden Prototypen Verwendung gefunden haben. Für den Turingmaschinen-Artikel konnten drei verschiedene Werte für einen vage definierten „Anspruch“ ausgewählt werden:

„Schüler“ „1. Semester“ „4. Semester“

Wie durch die Darstellung hier schon deutlich gemacht wird, haben die Werte untereinander eine Ordnung. Dies legt eine erste Anforderung nahe:

- Auswahl aus einer geordneten Gruppe von Kriterien

Gruppen in diesem Sinne sind dafür da, Kriterien zu logischen Einheiten zusammenzufassen und außerdem sicherzustellen, dass aus den gruppierten Kriterien nur genau eines ausgewählt werden kann.

Im zweiten Prototyp war als ungefähre Voraussetzung „Programmiersprachenkenntnis“ wählbar und es gab ebenfalls drei diskrete Werte zur Auswahl: „keine“, „C“ und „C#“. Anders als beim ersten Prototyp war hier jedoch keine offensichtliche Ordnung erkennbar. Dies würde auf eine weitere mögliche Anforderung hindeuten:

- Auswahl aus einer ungeordneten Gruppe von Kriterien

Die Unterscheidung ist vielleicht nicht nötig, könnten doch beide auf *„Auswahl aus einer Gruppe von Werten“* reduziert werden, jedoch haben geordnete Werte eine wesentliche Eigenschaft, die relevant ist: sie bauen aufeinander auf. Üblicherweise beinhalten also „höhere“ Kriterien einer Gruppe alle „niederen“. Bei einer ungeordneten Gruppe von Kriterien ist dies nicht der Fall.

Wird diese Art der Modellierung auf der Metaebene etwas weitergedacht, so kann noch eine Erweiterung aufgegriffen werden, die in Abschnitt 4.2.2 angesprochen wurde. Die Texte zur Einführung in die Programmiersprache Java haben bislang nur eine Gruppe von Kriterien – die Programmiersprache, die bisher bekannt ist (oder gar keine). Diese modelliert das *Vorwissen* des Lesers. Parallel dazu (und unabhängig davon) könnten aber durchaus auch *Interessen* bzw. *Ziele* des Lesers modelliert werden. Beispielsweise könnte ein weiteres Kriterium sein, dass der Leser nicht nur Java kennenlernen möchte, sondern das Wissen auch gleich nutzen möchte, um grafische Benutzeroberflächen mit Swing zu gestalten – oder XML-Daten zu verarbeiten. Dies wäre komplett unabhängig von der Auswahl bereits bekannter Programmiersprachen und könnte sich in zusätzlichen Kapiteln nach der Einführung in die Programmiersprache selbst äußern. Diese zusätzlichen Kapitel wären für sich genommen relativ austauschbar (auch wenn sie potentiell noch Abhängigkeiten zu weiterem Vorwissen haben – dies könnte dann eine ähnliche Komplexität annehmen wie der Abhängigkeitsgraph in Abbildung 2.2).

In diesem Fall passen die zusätzlichen Kriterien aber nicht mehr unbedingt in die beiden obigen Kategorien – Sinn der Gruppen ist es, genau eine Auswahl aus einer Gruppe zuzulassen. Es liegt also nahe, noch eine weitere Art Kriterien einzuführen:

- Auswahl von Kriterien, die unabhängig von anderen sind

Damit gibt es nun grundsätzlich drei verschiedene Arten von Kriterien, die potentiell unterschiedlich behandelt werden müssen. Zu beachten ist hier insbesondere, dass Kriterien, die in die drei genannten Kategorien fallen, jeweils unabhängig voneinander sind. Gruppierte Kriterien erlauben zwar nur jeweils eine Auswahl innerhalb der Gruppe, aber weitere Kriterien, die entweder allein stehen oder Teil einer weiteren Gruppe sind, bleiben davon unberührt und können zusätzlich dazu ebenfalls ausgewählt werden. Jedes Leserkriterium kommt letzten Endes aber nur einmal vor, darf also nicht von zwei verschiedenen Gruppen umfasst werden.

Die bisherigen Überlegungen können folgendermaßen zusammengefasst werden:

- Es gibt Kriterien, die vom Leser ausgewählt werden können, um eine bestimmte Textvariante zu wählen.
- Kriterien können in Gruppen zusammengefasst werden, wobei Gruppen die Auswahl der gruppierten Kriterien auf genau eins beschränken.
- Kriterien in Gruppen können entweder geordnet oder ungeordnet sein.
- Die Auswahl von Kriterien aus unterschiedlichen Gruppen sowie von nicht gruppierten Kriterien ist jeweils unabhängig voneinander.

Wird betrachtet, wie die oben beschriebenen Kriterien, die der Leser auswählt, verwendet werden, um als *Autor* eine Textvariante zusammenzusetzen, so fällt auf, dass Kriterien (in dieser noch sehr allgemeinen Form) eigentlich zwei verschiedene – jedoch verwandte – Konzepte modellieren können: etwas weiter gefasste *Zielgruppen* von Lesern, sowie konkrete bzw. elementare *Anforderungen*, die exakt definieren, welche Kenntnisse nötig sind, um ein Textfragment zu verstehen. In diesem Falle wären die Zielgruppen-Kriterien Hilfsmittel für den Autor, um Gruppen von Lesern zu konkretisieren und würden vom Leser ausgewählt werden[7]. Die elementaren Anforderungen wären ausschließlich für den Autor bestimmt um den Text genauer zu modellieren und für einen Leser unsichtbar.

Als Beispiel sei hier noch einmal der Turingmaschinen-Artikel genannt. Die eher vage umschriebenen Zielgruppen („Schüler“, „Erstsemester-Student“, „Viertsemester-Student“) beschreiben konkrete zugrundeliegende Kenntnisse (oder das Fehlen solcher), die für das Verständnis des Textes relevant sind. Im vorliegenden Fall existieren deshalb die folgenden elementaren Anforderungen:

- ausreichende Mathematikkenntnisse, um zwischen informellen und formalen Definitionen zu unterscheiden
- Grundlagenwissen in der theoretischen Informatik bezüglich Automaten und formalen Sprachen

Einem Schüler fehlen beide elementaren Kenntnisse, der Erstsemester-Student hat schon grundlegenden Einblick in höhere Mathematik und der Viertsemesterstudent zusätzlich Wissen in theoretischer Informatik.

Wie in Abschnitt 4.2.1 beschrieben, wurden im Prototyp die Textfragmente aufgrund der gewählten Anspruchsebene angezeigt oder ausgeblendet. Wie gerade erläutert, sind diese zu wählenden Kriterien jedoch als Auswahlkriterium auf Textebene nicht unbedingt sinnvoll. Sie modellieren grob die Zielgruppe, verbergen jedoch die elementaren Anforderungen. Im Grunde sind diese jedoch die tatsächlichen Kriterien nach denen eine Textvarian-

[7] Hierfür sollten die Kriterien natürlich so benannt sein, dass ein Leser auch in der Lage ist, sich entsprechend einzuschätzen.

te zusammengestellt oder verfasst wird. Im Quelltext des Prototyps finden sich Stellen wie die folgende:

```
<p>
  Die Turingmaschine ist ein von dem britischen Mathematiker Alan Turing 1936
  entwickeltes Modell<span class='level-3'>, um eine Klasse von berechenbaren
  Funktionen zu bilden</span><span class='level-1 level-2'> eines
  Computers</span>. <span class='level-2 level-3'>Sie gehört zu den
  grundlegenden Konzepten der Informatik.</span>
</p>
```

Dies ist ein Absatz in HTML, der aus fünf einzelnen Textfragmenten besteht. Hiervon sind zwei grundsätzlich sichtbar, die anderen haben entsprechende Auszeichnungen (das `class`-Attribut), so dass sie nur für bestimmte Zielgruppen sichtbar sind. Die Attributwerte `level-1` bis `level-3` entsprechen den drei Zielgruppen für den Text, die jeweils eine eigene Variante ausmachen. Die Auswahl einzelner Textfragmente erfolgt im Prototypen aber nach den Zielgruppen, was häufig die explizite Nennung mehrerer Werte nötig macht, wenn Textfragmente für verschiedene Zielgruppen sichtbar sein sollen.

Was hier tatsächlich zugrunde liegt, ist vielmehr eine Auswahl nach den oben genannten *konkreten Anforderungen*, die sich hinter den Zielgruppen verbergen – zumindest aus Sicht des Autors. Beispielsweise spaltet sich der Text am Anfang in die beiden alternativen Teilsätze *„Modell, um eine Klasse von berechenbaren Funktionen zu bilden“* und *„Modell eines Computers“* auf. Die Auswahl zwischen den beiden Fragmenten erfolgt darüber, dass das erste Fragment nur für Viertsemesterstudenten angezeigt wird, das zweite nur für Schüler und Erstsemester. Das eigentliche Kriterium hier ist aber *„Grundlagenwissen der theoretischen Informatik“*, welches nötig ist, um mit dem Begriff *„berechenbare Funktion“* etwas anfangen zu können. Die Auswahl über die Zielgruppe ist hier eher unpraktisch, insbesondere, wenn die Anzahl der Zielgruppen größer wird und sich auch mehr Kenntnisse zwischen diesen überlappen. Sinnvoller wäre also eher, wenn diese Stelle im Quelltext vom Prototyp folgendermaßen modelliert wäre (HTML-ähnlicher Pseudocode):

```
<p>
  Die Turingmaschine ist ein von dem britischen Mathematiker Alan Turing 1936
  entwickeltes Modell<span condition='theo'>, um eine Klasse von berechenbaren
  Funktionen zu bilden</span><span condition='not theo'> eines Computers</span>.
  <span condition='math'>Sie gehört zu den grundlegenden Konzepten der
  Informatik.</span>
</p>
```

Hierbei werden statt der Zielgruppen die konkreten Anforderungen (die von den Zielgruppen zwar umfasst, aber nicht für diesen Zweck sinnvoll beschrieben werden) als Bedingungen für Textfragmente genutzt. `math` steht hier für die mathematische Grundbildung, die etwa mit dem ersten Semester des Informatikstudiums erworben wird und `theo` für die Grundlagen der theoretischen Informatik, die etwa nach dem vierten Semester ausreichen, um den vollständigen Text zu verstehen. Weiterhin wurde hier `not theo` als Bedingung angegeben, was auf Negation der Bedingung `theo` testen soll. Dieses Konzept

stellt keine vollständige Umsetzung dar, sondern dient im Wesentlichen dazu, den Unterschied zwischen der Auswahl von Textfragmenten anhand von Zielgruppen-artigen Kriterien und elementaren Anforderungen deutlich zu machen. Verglichen mit dem ersten Codebeispiel ist es wesentlich naheliegender, die Textfragmente anhand von konkreten Anforderungen (welche hier spezifische Kenntnisse modellieren), statt aufgrund vage umschriebener Zielgruppen, zusammenzustellen.

Wie oben schon kurz angedeutet, stellen die beiden nebeneinanderstehenden Textfragmente, die nun die Bedingungen `theo` und `not theo` haben (bzw. vorher `level-3` sowie `level-1 level-2`), eine Alternative dar, bei der nur eines der beiden Fragmente im fertigen Text sichtbar ist. In diesem Falle wurde die Alternative nicht explizit kenntlich gemacht, jedoch ist die Unterscheidung zwischen Alternativen und Einschüben von Textfragmenten für die Textmodellierung eventuell sinnvoll; dies wird im folgenden Abschnitt noch genauer behandelt.

Es erscheint also sinnvoll, die Metaebene zunächst auch wieder in zwei verschiedene Teile zu gliedern. Zum einen die Modellierung der elementaren Anforderungen an eine Textvariante und ihre Fragmente, zum anderen die Zielgruppen, bzw. die Kriterien, die der Leser später wählen kann, welche die elementaren Anforderungen aggregieren.[8] Elementare Anforderungen sind dann genau jene Anforderungen, die sich nicht sinnvoll weiter unterteilen lassen im Kontext des Themas und Textes, der modelliert wird. Beispielsweise ergibt es wenig Sinn, in dem Turingmaschinen-Artikel die elementare Anforderung „Mathematikkenntnisse“ noch weiter zu unterteilen in „Algebra“ oder „Analysis“, da diese Teilgebiete im Rahmen des Textes irrelevant sind. Für die Zusammenstellung der Textvarianten können dann diese elementaren Anforderungen die einzigen Kriterien sein, die in den jeweiligen Bedingungen vorkommen – wie oben am Beispiel gesehen, ist die Modellierung damit angenehmer als über die Zielgruppen.

Im zweiten Prototyp sieht diese Modellierung anfangs sehr ähnlich aus. Es gibt wieder drei direkt genannte Zielgruppen:

- Anfänger ohne jegliche Programmiererfahrung
- C-Programmierer, die keine Erfahrung mit objektorientierter Programmierung (OOP) haben
- C#-Programmierer, die nur die Unterschiede zwischen C# und Java nicht kennen

Hier sind die zugrundeliegenden konkreten Anforderungen ebenfalls wieder hinter „Zielgruppen“-Kriterien versteckt:

- Leser hat Programmiererfahrung
- Leser hat OOP-Kenntnisse
- Leser ist vertraut mit C-ähnlicher Syntax

[8] In gewisser Weise ist dies dann eine Meta-Metaebene.

Hierbei würde der Anfänger keine der drei Anforderungen erfüllen, der C-Programmierer hat Programmiererfahrung und ist mit der Syntax vertraut, der C#-Entwickler hingegen erfüllt alle drei Anforderungen. Geht es im zugrundeliegenden Text nur darum, unter Berücksichtigung von *Konzepten,* die hilfreich beim Verständnis sind, eine Einführung in Java zu geben, dann reicht es natürlich aus, die Anforderungen wie oben zu modellieren. Damit sind aber die gegebenen Zielgruppen unnötig und es reicht, sie zu reduzieren auf:

- Leser hat keinerlei Programmiererfahrung
- Leser kennt die grundsätzliche Syntax, aber weiß nichts über OOP
- Leser kennt die grundsätzliche Syntax und OOP

Sinnvoller ist es sicherlich, die bisherigen Kenntnisse von Programmiersprachen mit in den Text einfließen zu lassen, um damit dem Leser Anknüpfungspunkte an eine bereits bekannte Programmiersprache zu geben. Wenn daran festgehalten werden soll, dass Textfragmente nur aufgrund elementarer Anforderungen ausgewählt werden, dann fehlt hier die jeweilig zu beachtende Programmiersprache. Eine durch ihre Einfachheit bestechende Lösung wäre, für die zwei Sprachen einfach zwei zusätzliche elementare Anforderungen einzuführen: „kann C“ und „kann C#“. Anders als die anderen elementaren Anforderungen sind diese jedoch im Modell dieses Textes nicht unabhängig, sondern schließen sich gegenseitig aus[9]. Aus diesem Grund könnten hier auch Alternativen auf Ebene von elementaren Kriterien eingeführt werden, allerdings verkompliziert dies die Kriterien und soll deshalb zunächst nicht betrachtet werden[10].

> ⓘ **Kriterien** stellen einzelne Anforderungen an eine *Textvariante* dar. Diese unterteilen sich in Kriterien, die der Leser auswählen kann, (**Leserkriterien**) sowie **elementare Kriterien**, die grundlegende Anforderungen beschreiben und zur Textgestaltung durch den Autor vorgesehen sind.

Das Obige kann nun wie folgt zusammengefasst werden. Es wird zwischen zwei verschiedenen Kriterien unterschieden, die in der Metaebene modelliert werden. Dies seien zum einen die sogenannten *elementaren* Kriterien (bisher zum Teil als *elementare Anforderungen* bezeichnet), die als Grundbausteine und Anforderungen nur einen booleschen Wert, also *wahr* oder *falsch*, haben. Dieses sind die Kriterien, die der Autor nutzen kann, um konkrete Anforderungen an eine Textvariante zu modellieren; infolgedessen werden mit Textfragmenten verknüpfte Bedingungen auch ausschließlich auf Basis der elementaren Kriterien gestaltet (auf dies wird im Detail im nächsten Abschnitt noch eingegangen).

---

[9] Im Modell des Prototypen wurde zumindest nicht vorgesehen, dass ein Leser *sowohl* C *als auch* C# beherrschen könnte. Prinzipiell ließe sich dies natürlich dennoch modellieren, wobei an verschiedenen Stellen im Text auf das jeweils relevanteste Vorwissen für den Teil eingegangen werden könnte. Dies ist im Prototyp jedoch nicht so geschehen.

[10] Dies ist auch nicht zu verwechseln mit Alternativen bei der Modellierung der Textfragmente, was vorher schon kurz angesprochen wurde.

Die zweite Art Kriterien sei hier mit *Leserkriterien* betitelt. Dies sind Kriterien, die der Leser wählen kann um eine bestimmte Textvariante auszuwählen. Gleichzeitig jedoch, wie oben festgestellt, decken sich gerade diese Kriterien ungefähr mit Zielgruppen, eher vagen Vorstellungen des Lesers oder Interessen und Zielen des Lesers. Da ein Autor diese beim Verfassen von Texten ebenfalls beachten muss, sind Leserkriterien nicht ausschließlich die Auswahlhilfe für den Leser, sondern auch eine Gestaltungshilfe für den Autor.

Leserkriterien sind im Grunde eine Aggregation mehrerer elementarer Kriterien. Hierzu wird jedes Leserkriterium mit einer Menge elementarerer Kriterien verknüpft, die jeweils den Wert „wahr“ erhalten, wenn das Leserkriterium ausgewählt wurde. Dies kann es in einigen Fällen nötig machen, dass Leserkriterien existieren, die lediglich zu genau einem elementaren Kriterium delegieren. Dies sollte allerdings aufgrund der normalerweise recht geringen Anzahl von Leserkriterien kein großes Problem sein und wurde daher nicht weiter als solches betrachtet. Generell haben elementare Kriterien anfänglich den Wert „falsch“; sie werden nur gesetzt, wenn ein entsprechendes Leserkriterium ausgewählt wurde.[11]

Um Gruppen von Werten für die Leserkriterien zu modellieren, können mehrere solche Kriterien in Gruppen zusammengefasst werden. Innerhalb von Gruppen ist jeweils immer nur ein Kriterium wählbar, aber nicht-gruppierte Kriterien sowie weitere Gruppen können unabhängig voneinander belegt werden. Eingangs wurde zwar erwähnt, dass es sinnvoll sein könnte, zwischen der Auswahl aus einer Gruppe von geordneten und ungeordneten Werten zu unterscheiden. Diese Unterscheidung wird hier jedoch nicht in besonderer Weise vorgenommen. Dies wäre noch eine mögliche spätere Erweiterung.

Hier wurde bewusst auf eine komplexere Modellierung der elementaren Kriterien verzichtet. Dies geschah aus zweierlei Gründen. Zunächst ist ein einfacheres Modell, wenn es ausdrucksstark genug ist um die Anforderungen zu erfüllen, zu bevorzugen. Weiterhin ist im Verlauf dieser Arbeit kein Fall aufgetreten, in dem boolesche elementare Kriterien nicht ausreichend gewesen wären. Vage Einstufungen wie „Anfänger“ und „Experte“ gehen immer auf eine Menge von konkreten Konzepten und Fakten zurück, die der Autor jeweils als bekannt oder unbekannt voraussetzt.

Die Unterscheidung in elementare und Leserkriterien erfolgt auch, weil es in komplexeren Texten wahrscheinlich viele elementare Kriterien gibt. Steht ein Leser dann vor einer Auswahl ebensolcher Kriterien, so ist anzunehmen, dass die Wahl sinnvoller und geeigneter elementarer Kriterien schwerfällt[12]. Die korrekte Einschätzung eigener Fähigkeiten oder Unfähigkeiten fällt vielen Menschen schwer. Aus diesem Grunde ist eine Vorauswahl

[11] Eine Zuordnung von expliziten Ausgangswerten für elementare Kriterien sowie den Werten, die ein Leserkriterium dann setzt, ließe sich prinzipiell überlegen. Dagegen spricht jedoch, dass es das Modell nicht mächtiger macht, wohl aber komplexer.

[12] Dies könnte analog zu der Auswahl eines geeigneten Hyperlinks zum Weiterlesen gesehen werden, was in Abschnitt 2.1.3 schon angesprochen wurde.

und Einordnung in abstraktere Begriffe etwas, was eine gemeinsame Repräsentation sowohl für den Autor als auch den Leser darstellt. Klappentexte auf Büchern erwähnen beispielsweise nicht jedes einzelne Detail, welches für das Verständnis nötig ist (oder im Buch erwähnt wird); stattdessen erfolgt wiederum nur eine Einordnung in ungefähre Zielgruppen sowie grobe Kenntnisbereiche und Ziele. Je nachdem, wie grob die Leserkriterien durch den Autor definiert wurden, kann es zwar immer noch sein, dass der Leser sich nicht exakt getroffen sieht (beispielsweise bei einer simplen Einordnung in „Anfänger“ und „Experte“), jedoch ist davon auszugehen, dass Kriterien üblicherweise so formuliert werden, dass ein Leser mit zumindest minimaler Kenntnis des jeweiligen Fachgebietes in der Lage ist, sich einzuordnen.

Ein Beispiel hierfür wäre wieder der Prototyp zur Einführung in Java in Abschnitt 4.2.2, bei dem die Kriterien bislang bekannte Programmiersprachen modellieren. Ein Leser, der noch keine Programmiersprache kennt, wird keine Probleme haben, festzustellen, dass er weder C noch C# beherrscht – ebenso ein Leser, der schon Programmiererfahrung hat. Generell sind zu vage und ungenaue Leserkriterien als ebenso schlecht geeignet anzusehen wie zu detaillierte. Ein gutes Maß für zu hohen Detailgrad wäre möglicherweise, wie bewusst einem potentiellen Leser das Wissen über ein Kriterium ist. Dass für das Erlernen von Java bisherige Erfahrung in objektorientierter Softwareentwicklung hilfreich ist, ist womöglich noch ersichtlich (gegeben mindestens durch den Bekanntheitsgrad der Sprache) – dass zum Erlernen der Programmiersprachen Python oder Ruby Wissen in funktionaler Programmierung nützlich sein kann, wohl weniger.

4.3.2 Textebene

Während die Metaebene die Anforderungen und Zielgruppen der Textvarianten definiert, werden in der Textebene die konkreten Textvarianten modelliert. Wie oben erwähnt, ist eine These dieser Arbeit, dass sich Textvarianten häufig nur wenig unterscheiden. Folglich sollten Teile des Modells für verschiedene Varianten wiederverwendbar sein ohne zusätzlichen Modellierungsaufwand zu haben. Das Textmodell muss allerdings ebenso in der Lage sein, Textvarianten darstellen zu können, die sich sehr stark unterscheiden. Beispielsweise unterscheiden sich die drei Varianten des Wikipedia-Artikels „Turingmaschine“ nur geringfügig in dem Sinne, dass einige Teile in mehreren Varianten vorkommen. Auf der anderen Seite sind die drei Texte zur Einführung in die Programmiersprache Java insgesamt so unterschiedlich, dass es nahezu keine gemeinsamen Teile gibt.

Exemplarisch dargestellt ist dies in Abbildung *4.4*. Die Darstellung als Baumstruktur spiegelt die Gliederungsstruktur der Texte wider, wie sie ungefähr im HTML-Quelltext modelliert wurde (Titel auf der ersten Ebene, Überschriften eine Ebene darunter, etc.). Der Text selbst erstreckt sich in dieser Darstellung von oben nach unten. Die Farbe eines Knotens ist ein Indikator der Variante, zu der der Knoten gehört. Weiße Knoten in der Abbildung stellen Textfragmente dar, die in jeder Variante enthalten sind. In diesem Sinne sind zwei-

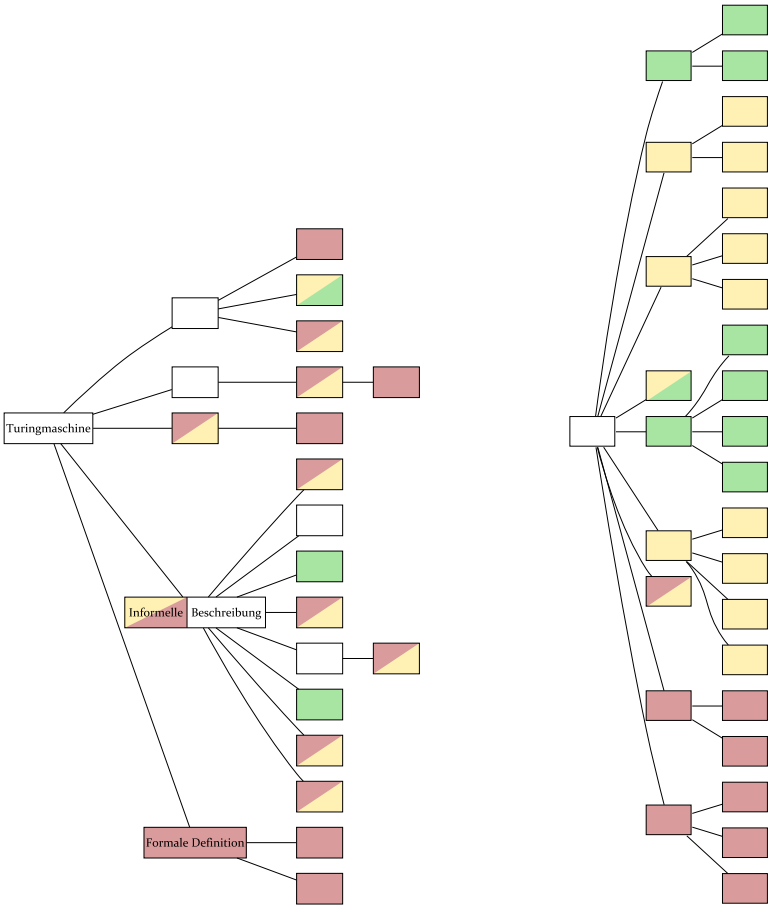


Abbildung 4.4 *Visualisierung des ersten Teils der Struktur des Wikipedia-Artikels „Turingmaschine“ sowie des Java-Einführungs-Prototypen als Baum und farbliche Hervorhebung der Zugehörigkeit einzelner Knoten zu den verschiedenen Varianten.*

farbige und weiße Knoten besonders interessant, da sie repräsentieren, wie viel Text in mehr als nur einer Variante enthalten ist.

Bevor auf mögliche Modellierungsformen der Textebene eingegangen wird, sollen zunächst einige weitere Anforderungen an diese genannt werden. Die Notwendigkeit, Strukturen und Teile einzelner Varianten weiterverwenden zu können, wurde oben schon genannt. Weitere Anforderungen lassen sich in den Strukturen und Unterschieden der beiden Prototypen aus Abschnitt 4.2 finden.

Ein häufiges Muster ist beispielsweise die Aufspaltung in verschiedene Alternativen. Dies wurde bei der Beschreibung des Turingmaschinen-Artikels in Abschnitt 4.2.1 an verschiedenen Stellen angemerkt (oder im Extremfall wie im zweiten Prototypen, wo im Grunde nur eine große Alternative für jede der Varianten existiert). In Abbildung *4.4* finden sich einige solche Alternativen im linken Baum, zum Beispiel die beiden obersten farbigen Knoten. Diese sind allerdings nur aufeinanderfolgende Textfragmente, die zu unterschiedlichen Varianten gehören, da Alternativen im Prototyp nicht modelliert werden konnten. Folgen verschiedene Fragmente für die gleiche Textstelle direkt aufeinander statt

nebeneinander, so ist das Ergebnis wenig übersichtlich und erschwert es, dem eigentlichen Verlauf des Textes zu folgen. Aus diesem Grunde wird hier eine explizite Modellierung von Alternativen als sinnvoll angesehen.

Oft sind solche Alternativen auch nur kurzzeitig getrennt, beispielsweise um einen einzelnen Satz oder Absatz an eine bestimmte Leserschaft anzupassen. Dies bedeutet, dass die Alternativen danach wieder zusammengeführt werden müssen, um den Text fortzusetzen. In Abschnitt 4.2.2 und 4.3.1 wurde außerdem die Möglichkeit diskutiert, die drei Textvarianten zur Einführung in die Programmiersprache Java durch weitere Kapitel zu erweitern, die jeweils auf spezifische Technologien oder Anwendungsfälle eingehen, welche den Leser interessieren. Da bisher alle drei Varianten verschiedene Zweige einer großen Alternative sind, wird für diese nachfolgenden Kapitel ebenfalls eine Zusammenführung nötig.

Zusammenfassend gibt es nun folgende Anforderungen an die Struktur des Textes in der Textebene:

- Nach Möglichkeit keine Wiederholung von Strukturelementen oder Textfragmenten, die in mehreren Varianten vorkommen.
- Explizite Modellierung von Alternativen im Text.
- Die Möglichkeit, nach Alternativen eine einheitliche Struktur fortzuführen.

Aufgrund obiger Anforderungen geschieht die Modellierung des Textes als gerichteter, azyklischer Graph, der die Struktur des Textes und der Varianten beschreibt. Die Knoten dieses Graphen bestehen aus einem Textfragment, sowie einer Bedingung, die festlegt, in welchen Varianten das Textfragment sichtbar ist. Bedingungen sind in diesem Modell boolesche Funktionen, die elementare Kriterien als Variablen benutzen, beispielsweise $A \vee B$, $\bar{A} \vee (C \wedge B)$ oder $\bar{C}$. Daneben gibt es noch die Möglichkeit keiner explizit festgelegten Bedingung; in diesem Fall ist sie implizit immer wahr und der Knoten demzufolge Teil jeder Variante. Der Graph enthält zudem einen einzelnen Knoten, der als Startpunkt für die Erzeugung einer Textvariante dient, im Folgenden *Startknoten* genannt.

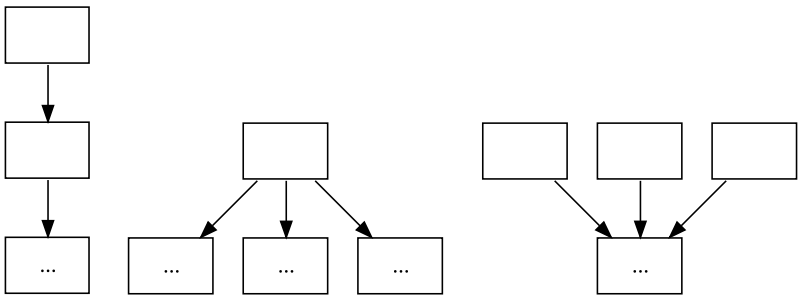


Abbildung 4.5 *Visualisierung der wesentlichen Strukturelemente bei der Modellierung des Textes als gerichteten Graphen. Links eine einfache Sequenz von Knoten, in der Mitte eine Aufspaltung in Alternativen, rechts die Wiederzusammenführung mehrerer Alternativen zu einem Nachfolgeknoten.*

Die Bedingungen werden hier den Knoten zugeordnet. Sie könnten stattdessen auch den Kanten im Graphen zugeordnet werden, jedoch wird angenommen, dass gewisse Voraussetzungen für ein Textfragment (was die Bedingungen ja modellieren) eher mit dem Textfragment zu tun haben als mit dem Pfad dorthin. Die Bedingung ist also für jede Kante die zu einem bestimmten Knoten führt, gleich. Falls obige Annahme korrekt ist (und sie gilt zumindest für alle im Rahmen dieser Arbeit erstellten Prototypen und Experimente), so würden an Kanten modellierte Bedingungen oftmals die gleichen Bedingungen an allen Zusammenführungen nötig machen.

Im einfachsten Falle – für eine einzige Textvariante – fängt der Graph beim Startknoten an und ist eine lineare Folge von Knoten bis zum Ende. Aus ersichtlichen Gründen darf der Graph keine Zyklen haben, da sonst ein endloser Text die Folge wäre. Wird mehr als eine Variante modelliert, so können Alternativen eingeführt werden, indem ein Knoten mehr als einen Nachfolger hat. In diesem Falle kommen die Bedingungen zum Tragen. Da bei einer solchen Verzweigung für eine Variante auch nur ein einziger Text entstehen kann, darf es nur einen möglichen Nachfolgeknoten geben, dessen Bedingung im Kontext der gesetzten elementaren Kriterien wahr ist. Solche Verzweigungen können als Sequenz weitergeführt werden, sich nach anderen Kriterien weiter verzweigen, oder aber wieder in einem Knoten zusammengeführt werden. Nicht jeder Text macht es nötig, dass sich alle Varianten stark voneinander unterscheiden. Diese drei Grundstrukturelemente sind in Abbildung *4.6* dargestellt.

Der Graph beschreibt alle möglichen Lesepfade eines solchen dynamischen Textes. Bedingungen steuern dann die möglichen Verzweigungspunkte, die der Autor planen und festlegen kann. Im Turingmaschinen-Artikel fällt ein weiterer häufiger Anwendungsfall auf: Es gibt an zahlreichen Stellen Einschübe, die für einzelne Varianten gelten und einen Satz um ein Wort oder einen Satzteil ergänzen. Das ließe sich mit einer kurzen Alternative lösen, die in einem Fall den eingeschobenen Knoten überspringt. Die Vermutung ist

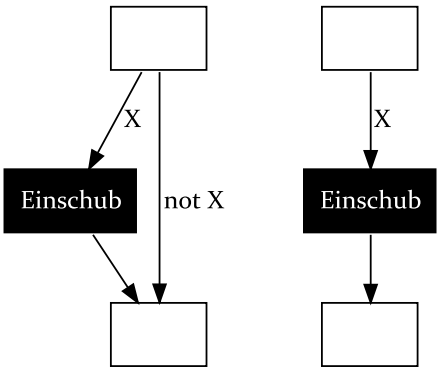


Abbildung 4.6 *Einschübe könnten so wie in der linken Illustration modelliert werden, indem als Teil einer Alternative der Einschub unter bestimmten Bedingungen übersprungen wird. Um so etwas einfacher modellieren zu können, wird hier festgelegt, dass die Modellierung wie in der rechten Illustration exakt das gleiche Ergebnis liefert und den Einschub einfach überspringt, sollte die Bedingung X nicht zutreffen.*

allerdings, dass Einschübe häufig genug sind, um eine Vereinfachung ihrer Modellierung zu rechtfertigen. Aus diesem Grunde werden Knoten innerhalb einer Sequenz (also mit nur einem Nachfolger) besonders behandelt. Falls in dem Fall die Bedingung des Knotens nicht zutrifft, wird der Knoten für den resultierenden Text ignoriert und stattdessen sein Nachfolger betrachtet. Diese Lösung ist genau äquivalent zu einer weiteren Kante, die den eingeschobenen Knoten überspringen würde (siehe auch Abbildung *4.6*).

Zusammenfassend heißt dies nun für die Modellierung der Textebene folgendes:

- Die Textebene wird durch einen gerichteten azyklischen Graph repräsentiert, dessen Knoten aus je einem Textfragment und einer Bedingung bestehen.
- Bedingungen sind boolesche Funktionen, die elementare Kriterien als Variablen nutzen. Implizit ist die Bedingung eines Knotens wahr, solange nichts anderes festgelegt wurde (im nachfolgenden Beispiel durch *true* gekennzeichnet).
- Hat ein Knoten nur einen Nachfolger, wird dieser Nachfolger übersprungen, falls dessen Bedingung nicht zutrifft.
- Hat ein Knoten mehrere Nachfolger, so muss es für jede mögliche Belegung elementarer Kriterien genau einen möglichen Nachfolger geben, dessen Bedingung zutrifft.
- Bedingungen steuern Sichtbarkeit einzelner Textfragmente bzw. ganzer Teilpfade im Graphen, indem sie wie oben beschrieben das Ablaufen des Graphen beeinflussen.

4.3.3 Beispiel

Im Folgenden soll noch ein kurzes abstraktes Beispiel gegeben werden, wie die Modellierung eines Textes ungefähr aussehen könnte. Die Metaebene soll hier aus vier elementaren Kriterien e_1 bis e_4 sowie drei Leserkriterien r_1 bis r_3 bestehen. r_1 bildet auf e_1 ab, r_2 auf e_2 und e_3, r_3 auf e_4. Die Leserkriterien r_1 und r_3 sind in einer Gruppe zusammengefasst. Diese Struktur ist in Abbildung *4.7* noch einmal illustriert.

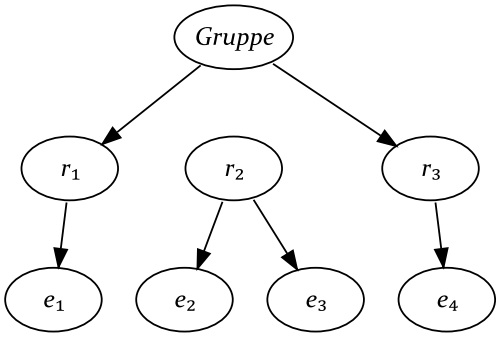


Abbildung 4.7 *Beispielhafte Metaebene mit vier elementaren Kriterien, drei Leserkriterien, sowie einer Gruppe und den jeweiligen Beziehungen zwischen diesen.*

Auf Basis oben beschriebener Metaebene wurde nun für die Textebene ein Graph erstellt, welcher in Abbildung *4.8* zu sehen ist. Jeder Knoten besteht aus einem Textfragment T_i und einer Bedingung, die entweder *true* ist oder auf elementare Kriterien zurückgreift.

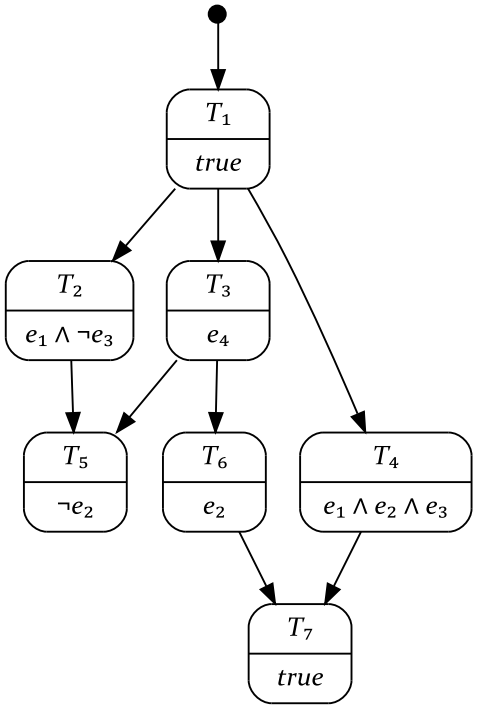


Abbildung 4.8 *Beispielgraph für eine Textebene. Die T_i repräsentieren jeweils Textfragmente, während darunter die Bedingungen für den Knoten angegeben sind.*

Es gibt für die Leserkriterien genau vier Auswahlmöglichkeiten mit entsprechenden Belegungen elementarer Kriterien (aus der Gruppe, die r_1 und r_3 enthält, darf nur ein Kriterium ausgewählt werden; r_2 ist unabhängig davon). Für jede dieser Auswahlmöglichkeiten gibt es dann einen eindeutigen Pfad durch den Graphen.

Auswahl	**Belegung**	**Pfad**
$\{r_1\}$	$e_1, \bar{e}_2, \bar{e}_3, \bar{e}_4$	$T_1 \rightarrow T_2 \rightarrow T_5$
$\{r_3\}$	$\bar{e}_1, \bar{e}_2, \bar{e}_3, e_4$	$T_1 \rightarrow T_3 \rightarrow T_5$
$\{r_1, r_2\}$	$e_1, e_2, e_3, \bar{e}_4$	$T_1 \rightarrow T_4 \rightarrow T_7$
$\{r_2, r_3\}$	$\bar{e}_1, e_2, e_3, e_4$	$T_1 \rightarrow T_3 \rightarrow T_6 \rightarrow T_7$

Die auf diesem Pfad gesammelten Textfragmente bilden dann jeweils den gewählten Text. In Kapitel 7 wird noch einmal ein umfassenderes Beispiel mit einem tatsächlichen Text gegeben.

4.3.4 Formale Spezifikation

Im Folgenden soll zumindest eine teilweise formale Spezifikation und Beschreibung der obigen Modellierungskonzepte gegeben werden. Hierzu wird die Spezifikationssprache Z verwendet (19; 20; 21), die insbesondere zur formalen Beschreibung von Softwaresyste-

men[13] entwickelt wurde (20). Z beruht auf Mengenlehre und Prädikatenlogik und kann sowohl statische als auch dynamische Aspekte eines Systems beschreiben. Ein wesentlicher Teil von Z ist das Typsystem. Jeder Wert und jeder Ausdruck hat einen Typ, wobei Typen jeweils Mengen von Werten sind, für die eine Reihe von Prädikaten als geltende Gesetze definiert sind. Beispielsweise ist $\mathbb{Z}$ der Standard-Typ für alle Ganzzahlen in Z und mitsamt den üblichen Rechenoperationen vordefiniert; sämtliche anderen verwendeten Typen einer Spezifikation müssen allerdings definiert werden.

Unterteilt werden Z-Spezifikationen in sogenannte Schemata, die jeweils einen Teilaspekt des Systems – oder auch einzelne Operationen darauf – modellieren und beschreiben. Schemata bestehen aus zwei Teilen: Einer *Deklaration* von Variablen und ihrer Typen sowie *Prädikaten*, die die möglichen Werte dieser Variablen einschränken (21). Diese werden üblicherweise durch Linien getrennt übereinander geschrieben, wie nachfolgend auch zu sehen ist. Schemata lassen sich als strukturierte Typen verwenden.

Im Rahmen dieser Arbeit soll allerdings keine ausführliche Einführung in Z gegeben werden; hierzu sei auf J. M. Spiveys *Z Reference Manual* (20) und *Using Z* (21) von Jim Woodcock und Jim Davies verwiesen. Im Verlauf der folgenden formalen Beschreibung des Systems wird jedoch jeweils erklärt, was im Einzelnen definiert und spezifiziert wird. In Anhang F findet sich eine Auflistung der in dieser Spezifikation verwendeten Syntaxelemente nebst Verweisen zu Definitionen und Erklärungen in oben genannten Quellen.

Zunächst haben viele Elemente – wie Kriterien und Gruppen – einen Namen. Wie dieser jeweils aussieht, soll in der Spezifikation zunächst nicht weiter interessieren, weshalb Namen lediglich durch je einen Basistyp repräsentiert werden.

$$[ECriterion, RCriterion, Group]$$

Die so definierten Typen sind jeweils Mengen von Namen, die für die Kriterien bzw. Gruppen verwendet werden. Die Metaebene kann dann folgendermaßen beschrieben werden.

MetaLayer

$$elementaryCriteria : \mathbb{F}\ ECriterion$$
$$readerCriteria : RCriterion \pfun \mathbb{F}\ ECriterion$$
$$groups : Group \pfun \mathbb{F}\ RCriterion$$

$$elementaryCriteria = \bigcup (\operatorname{ran} readerCriteria)$$
$$\bigcup (\operatorname{ran} groups) \subseteq \operatorname{dom} readerCriteria$$
$$\forall g_1, g_2 : \operatorname{ran} groups \bullet g_1 \cap g_2 = \varnothing$$

[13] Z wurde jedoch ebenso schon zur Spezifikation von Hardwaresystemen eingesetzt (21).

Der obere Teil des Schemas enthält die Deklarationen der Bestandteile der Metaebene. Sie enthält folglich elementare Kriterien, die hier lediglich durch ihre Namen repräsentiert werden. Weiterhin gibt es Leserkriterien, die eine Abbildung von Namen auf Mengen von elementaren Kriterien darstellen sowie Gruppen, die Leserkriterien zusammenfassen.

Im unteren Teil finden sich Prädikate, die festlegen, welche Bedingungen grundsätzlich gelten müssen. Die Menge der durch die Leserkriterien referenzierten elementaren Kriterien muss genau mit der Menge der definierten elementaren Kriterien übereinstimmen. Anderenfalls gäbe es für einige elementare Kriterien keine Möglichkeit, ihnen einen Wert zuzuweisen (dies geschieht wie beschrieben immer über Leserkriterien), wodurch sie immer ihren Standardwert hätten. Weiterhin muss die Menge der gruppierten Leserkriterien eine Teilmenge der definierten Leserkriterien sein und ein Leserkriterium darf nicht in zwei Gruppen enthalten sein.

Die Wertekonfiguration, die elementaren Kriterien Werte zuordnet, ist bewusst nicht Bestandteil der Metaebene. Das Textmodell, welches mit dieser Spezifikation definiert werden soll, beschreibt die Struktur und Anforderungen sämtlicher Varianten, während eine Wertekonfiguration lediglich eine einzige Variante beschreibt. Eine solche Wertekonfiguration lässt sich als Menge elementarer Kriterien beschreiben.

$$Configuration == \mathbb{F}\ ECriterion$$

Diese Menge enthält all jene elementaren Kriterien, die den Wert „*wahr*“ haben.

Ein Leser muss auch in der Lage sein, unter Berücksichtigung von Gruppen Leserkriterien auszuwählen. Eine solche Auswahl lässt sich, analog zu obiger Wertekonfiguration als Menge von Leserkriterien darstellen.

$$Selection == \mathbb{F}\ RCriterion$$

Die Wahl der Begriffe „Konfiguration“ und „Auswahl“ macht hier auch deutlich, mit welchem Konzept der Leser konfrontiert wird und welches das zugrundeliegende Konzept auf einer Implementierungsebene ist.

Des Weiteren ist eine Funktion hilfreich, welche für eine Auswahl von Leserkriterien in einer Metaebene die zugehörige Konfiguration elementarer Kriterien zurückgibt. Dies wird insbesondere als Zwischenschritt für das spätere Erstellen einer Textvariante aus dem Modell benötigt, da Bedingungen im Modell der Textebene auf die elementaren Kriterien zurückgreifen.

$$
\begin{array}{|l}
applySelection : Selection \times MetaLayer \longrightarrow Configuration \\
\hline
\forall\, s : Selection; ml : MetaLayer \mid \\
\quad s \subseteq \mathrm{dom}\ ml.readerCriteria \bullet \\
\quad applySelection\ (s, ml) \quad = \\
\qquad \{\, ec : ECriterion \mid \exists\, rc : s \bullet ec \in ml.readerCriteria\ rc \,\} \\
\\
\forall\, s : Selection; ml : MetaLayer; g : \mathbb{F}\ RCriterion \mid \\
\quad s \subseteq \mathrm{dom}\ ml.readerCriteria; \\
\quad g \in \mathrm{ran}\ ml.groups \bullet \\
\qquad \exists_1\, c : s \bullet c \in g
\end{array}
$$

Da sowohl die Auswahl als auch die Konfiguration auf Kriterien zurückgreifen, die in einer Metaebene definiert sind, muss diese Metaebene Teil der Funktionsdefinition sein. Die Funktion *applySelection* bildet also eine Auswahl im Kontext einer Metaebene auf eine Wertekonfiguration ab. Dies bedeutet, dass die in der Auswahl enthaltenen Kriterien in der Metaebene existieren müssen. Die resultierende Konfiguration ist dann genau die Menge elementarer Kriterien, für die ein *ausgewähltes* Leserkriterium existiert welches in der Metaebene das jeweilige elementare Kriterium mit einschließt. Weiterhin müssen die in der Metaebene definierten Gruppen beachtet werden. Es muss für jede Gruppe genau ein Leserkriterium in der Auswahl geben, da innerhalb einer Gruppe immer ein Kriterium ausgewählt sein muss.

Um die Textebene definieren zu können, werden nun Textfragmente und Bedingungen benötigt. Wie Textfragmente im Detail aussehen, soll hier wieder nicht weiter von Interesse sein.

$$[Fragment]$$

Bedingungen hingegen sind boolesche Funktionen, die Operatoren der Aussagenlogik verwenden.

$$
\begin{aligned}
Condition ::= \ & Val\ \langle\!\langle ECriterion \rangle\!\rangle \mid \\
& And\ \langle\!\langle Condition \times Condition \rangle\!\rangle \mid \\
& Or\ \langle\!\langle Condition \times Condition \rangle\!\rangle \mid \\
& Not\ \langle\!\langle Condition \rangle\!\rangle
\end{aligned}
$$

Diese werden durch einen Syntaxbaum repräsentiert, dessen Blätter Variablen (*Val*) sind, die auf elementare Kriterien zurückgreifen. Der Einfachheit halber wird die Syntax hier auf die Operatoren $\neg$, $\wedge$ und $\vee$ beschränkt.

Bedingungen können unter Berücksichtigung einer Konfiguration und einer Metaebene ausgewertet werden. Hierfür wird eine weitere Funktion benötigt.

$$
\begin{array}{|l}
\hline
satisfiableConditions : Configuration \longrightarrow \mathbb{P}\ Condition \\
\hline
\forall\ conf : Configuration; r : \mathbb{P}\ Condition; ml : MetaLayer \mid \\
\quad (\forall\ ec : conf \bullet Val\ ec \in r); \\
\quad (\forall\ ec : ml.elementaryCriteria \mid ec \notin conf \bullet Val\ ec \notin r); \\
\\
\quad (\forall\ c, c' : r \bullet \\
\qquad And\ (c, c') \in r \land \\
\qquad Or\ (c, c') \in r \land \\
\qquad Not\ c \notin r); \\
\\
\quad (\forall\ c : r; c' : Condition \mid c' \notin r \bullet \\
\qquad Not\ c' \in r \land \\
\qquad Or\ (c, c') \in r \land Or\ (c', c) \in r \land \\
\qquad And\ (c, c') \notin r \land And(c', c) \notin r) \bullet \\
\\
\qquad\quad satisfiableConditions\ (conf) = r \\
\hline
\end{array}
$$

Diese liefert zu einer Konfiguration und Metaebene sämtliche Bedingungen, die in diesem Kontext wahr sind. Zunächst muss die Konfiguration zu der Metaebene passen – dann ist jede Variable, die sich auf ein in der Konfiguration vorhandenes elementares Kriterium bezieht, wahr – ebenso wie jede Variable, die sich auf andere elementare Kriterien bezieht, falsch ist. Von dort an wird die Menge der Bedingungen jeweils unter Einbeziehung der drei erlaubten Operatoren rekursiv weiter konstruiert, so dass am Ende die Menge von Bedingungen übrig bleibt, die durch die Konfiguration erfüllt werden.

Ein Knoten im Graph besteht aus einem Textfragment sowie einer Bedingung.

$$
\begin{array}{|l}
\hline
GraphNode \\
\hline
text : Fragment \\
condition : Condition \\
\hline
\end{array}
$$

An diese werden keine weiteren Anforderungen gestellt, weshalb hier keine Prädikate vermerkt sind. Der Graph, der die Textstruktur beschreibt, kann dann analog zur mathematischen Definition durch Knoten und Kanten dargestellt werden:

Graph

$$
\begin{array}{l}
nodes : \mathbb{F}\ GraphNode \\
edges : GraphNode \leftrightarrow GraphNode \\
startNode : GraphNode \\
successors : GraphNode \rightarrow\!\!\!\!\!\!+\ \ \mathbb{F}\ GraphNode \\
\hline
\exists_1\, n : nodes \bullet n \notin \mathrm{ran}\ edges \land n = startNode \\
\forall\, n : nodes \setminus \{startNode\} \bullet (startNode \mapsto n) \in edges^+ \\
\forall\, n : nodes \bullet (n \mapsto n) \notin edges^+ \\
\\
\forall\, n : nodes \bullet successors\ n = \{\, x : nodes \mid (n \mapsto x) \in edges \bullet x \,\} \\
\\
\forall\, c : Configuration; n : GraphNode \mid \ n \in \mathrm{dom}\ edges \bullet \\
\quad \#\,(successors\ n) > 1 \Rightarrow \\
\qquad (\exists_1\, x : nodes \mid (n \mapsto x) \in edges \bullet \\
\qquad\quad x.condition \in satisfiableConditions\ c)
\end{array}
$$

Dieser besteht aus einer endlichen Menge von Knoten und einer Relation, die Knoten mit Knoten verbindet – den Kanten. In Abschnitt 4.3.2 wurde festgehalten, dass der Graph genau einen Startknoten haben soll, an dem der Text beginnt. Dies wird durch die Variablen *startNode* umgesetzt. In den Prädikaten wird dann sichergestellt, dass es lediglich einen Knoten mit den notwendigen Eigenschaften des Startknoten gibt (ein Knoten ohne Vorgänger). Weiterhin muss es für jeden anderen Knoten einen Pfad vom Startknoten zu diesem geben, damit der Graph zusammenhängend ist. Ebenso darf es keine Zyklen im Graphen geben; dies wird dadurch sichergestellt, dass die transitive Hülle über alle Kanten im Graph nicht reflexiv ist, d. h. kein Knoten von sich selbst aus erreichbar ist.

Um spätere Ausdrücke in der Spezifikation zu vereinfachen wurde ebenfalls eine Funktion definiert, die zu einem Knoten sämtliche Nachfolgeknoten zurückgibt.

Damit erfüllt der Graph fast alle in Abschnitt 4.3.2 geforderten Anforderungen. Was nun noch fehlt ist die Anforderung, dass es für jede Konfiguration elementarer Kriterien lediglich einen möglichen Pfad durch den Graphen gibt. Diese Bedingung ist etwas komplizierter, da hier zwei Fälle unterschieden werden müssen. Hat ein Knoten nur einen Nachfolger, so wird dieser einfach übersprungen, falls die Bedingung nicht zutrifft – folglich ist es dafür, ob es nur einen Pfad gibt im Grunde egal, ob die Bedingung wahr oder falsch ist. Gibt es jedoch mehrere Nachfolger, so darf es nur genau einen Nachfolgeknoten geben, für den die Bedingung wahr ist.

Im Folgenden ist es noch hilfreich, wenn aus einem Graphen die darin enthaltenen Textfragmente extrahiert werden können.

$$fragments : Graph \longrightarrow \mathbb{F}\ Fragment$$

$$\forall\, g : Graph \bullet\ fragments\ g = \{\, n : g.nodes \bullet n.text \,\}$$

Mittels der obigen Definitionen für den Graphen und die Textfragmente lässt sich die Textebene folgendermaßen beschreiben:

TextLayer

$$textStructure : Graph$$
$$textFragments : \mathbb{F}\ Fragment$$
$$textFragments = fragments\ textStructure$$

Kernpunkt der Textebene ist also der Graph aus Textfragmenten und Bedingungen sowie die Textfragmente, die Verwendung finden. Theoretisch müssten die Textfragmente hier nicht explizit modelliert werden, da sie jederzeit aus dem Graphen ableitbar sind. Jedoch soll die Spezifikation hier dicht am Text der in den beiden vorherigen Abschnitten genannten Modellierungskonzepte gestaltet werden, so dass einzelne Teile direkt nachvollziehbar sind[14].

Sind nun beide Ebenen spezifiziert, so können sie in einem Schema zusammengefasst werden:

Model

$$MetaLayer$$
$$TextLayer$$

Der Typ *Model* ist nun die Kombination beider Ebenen und enthält sämtliche Variablen und Prädikate, die in den beiden Ebenen vorkommen. Dies ist allerdings immer noch nur die Beschreibung des *Modells* eines dynamischen Textes – es beinhaltet alle nötigen Bestandteile, um Varianten zu beschreiben, allerdings nicht, um sie zu erzeugen.

Um eine Variante generieren zu können, lässt sich eine weitere Funktion definieren, die aus einem Modell, sowie einer Auswahl von Leserkriterien einen lesbaren Text macht. Dieser Text soll durch eine Sequenz von Textfragmenten repräsentiert werden.

[14] In ähnlicher Weise müsste der weiter oben definierte Graph auch die Menge der Knoten nicht explizit enthalten, der besseren Verständlichkeit halber und auch um näher an der eigentlichen mathematischen Definition zu sein, wurde dies jedoch trotzdem so modelliert.

Damit dies funktioniert, sind einige Hilfsfunktionen vonnöten. Zunächst muss es möglich sein, von einem Knoten aus den richtigen Nachfolger zu finden.

$$
\begin{array}{|l}
\mathit{findSuccessor} : \mathit{Graph} \times \mathit{GraphNode} \times \mathbb{P}\ \mathit{Condition} \longrightarrow \mathit{GraphNode} \\
\hline
\forall\ g : \mathit{Graph}; n : \mathit{GraphNode}; \mathit{conditions} : \mathbb{P}\ \mathit{Condition} \mid \\
\quad n \in g.\mathit{nodes}; \\
\quad g.\mathit{successors}\ n \neq \emptyset \bullet \\
\quad\quad \mathit{findSuccessor}\ (g, n, \mathit{conditions}) = (\textbf{let}\ s == g.\mathit{successors}\ n \bullet \\
\quad\quad\quad (\mu\ x : s \mid \#\ s = 1 \lor x.\mathit{condition} \in \mathit{conditions}))
\end{array}
$$

Diese Funktion benötigt den Graphen, den zu betrachtenden Knoten im Graphen sowie die Menge aller wahren Bedingungen. Es wird davon ausgegangen, dass der Knoten tatsächlich aus dem Graphen stammt und mindestens einen Nachfolger hat. Dann ist der richtige nachfolgende Knoten entweder der einzige Nachfolger oder – falls es mehrere gibt – der einzige für den die verknüpfte Bedingung wahr ist. Die Einschränkung, dass es für jeden Knoten mit mehreren Nachfolgern nur einen einzigen Nachfolger gibt, dessen Bedingung wahr ist, wurde schon in der Definition des Graphen festgelegt.

Der μ-Operator wählt hier aus den Nachfolgern des Knotens den einzigen Wert aus, für den das nachfolgende Prädikat wahr ist. In diesem Falle darf es entweder nur einen Nachfolger geben oder die Bedingung des Knotens muss wahr sein (und somit in der Menge aller wahren Bedingungen enthalten sein).

Als Nächstes wird eine Funktion benötigt, die ausgehend von einem bereits vorhandenen Textfragment und einem Knoten im Graphen je nach Bedingung des Knotens ein weiteres Textfragment anhängt und dann rekursiv mit dem richtigen Nachfolger fortfährt. Die Bestimmung des richtigen Nachfolgers erfolgt dann mittels der eben definierten Funktion *findSuccessor*.

$$
\begin{array}{|l}
\mathit{append} : \mathit{Graph} \times \mathit{GraphNode} \times \mathbb{P}\ \mathit{Condition} \times \mathrm{seq}\ \mathit{Fragment} \longrightarrow \mathrm{seq}\ \mathit{Fragment} \\
\hline
\forall\ g : \mathit{Graph}; n : \mathit{GraphNode}; \mathit{conditions} : \mathbb{P}\ \mathit{Condition};\ f : \mathrm{seq}\ \mathit{Fragment} \mid \\
\quad n \in g.\mathit{nodes} \bullet \\
\quad\quad \mathit{append}\ (g, n, \mathit{conditions}, f) = \\
\quad\quad\quad \textbf{if}\ n.\mathit{condition} \in \mathit{conditions}\ \textbf{then} \\
\quad\quad\quad\quad \textbf{if}\ (g.\mathit{successors}\ n) = \emptyset\ \textbf{then}\ f \frown \langle n.\mathit{text} \rangle\ \textbf{else} \\
\quad\quad\quad\quad\quad \mathit{append}\ (g, \mathit{findSuccessor}\ (g, n, \mathit{conditions}), \mathit{conditions}, f \frown \langle n.\mathit{text} \rangle) \\
\quad\quad\quad \textbf{else if}\ (g.\mathit{successors}\ n) = \emptyset\ \textbf{then}\ f\ \textbf{else} \\
\quad\quad\quad\quad \mathit{append}\ (g, \mathit{findSuccessor}\ (g, n, \mathit{conditions}), \mathit{conditions}, f)
\end{array}
$$

Diese Funktion bildet einen Graphen sowie einen Knoten daraus, eine Menge von wahren Bedingungen (um diese an *findSuccessor* weiterzureichen) sowie eine bisherige Sequenz

von Textfragmenten auf eine neue Sequenz von Fragmenten ab. Die zurückgegebene Sequenz ist die gleiche wie vorher, nur potentiell um ein weiteres Fragment erweitert. Wie vorher auch schon wird gefordert, dass der Knoten auch tatsächlich zum Graphen gehört. Zunächst ist das Ergebnis der Funktion abhängig davon, ob die Bedingung des Knotens wahr ist. Falls dies der Fall ist, wird das zum Knoten gehörige Textfragment an die vorhandene Sequenz angehängt, sonst wird die bisherige Sequenz unverändert weitergereicht. Für den Fall, dass der aktuell betrachtete Knoten keine Nachfolger hat, so ist die bis dahin konstruierte Sequenz auch das Ergebnis der Funktion, anderenfalls wird rekursiv mit dem richtigen Nachfolger weitergemacht, bis ein Knoten erreicht ist, der keine Nachfolger mehr hat.

Mit diesen beiden Hilfsfunktionen lässt sich nun eine Funktion definieren, die aus einem Modell sowie einer Auswahl von Leserkriterien einen Text erzeugt (wiederum repräsentiert als Sequenz von Textfragmenten).

$$
\begin{array}{l}
\mathit{generateText} : \mathit{Model} \times \mathit{Selection} \longrightarrow \mathrm{seq}\ \mathit{Fragment} \\
\hline
\forall\ m : \mathit{Model}; ml : \mathit{MetaLayer}; s : \mathit{Selection} \mid \\
\quad ml.\mathit{readerCriteria} = m.\mathit{readerCriteria}; \\
\quad ml.\mathit{elementaryCriteria} = m.\mathit{elementaryCriteria}; \\
\quad ml.\mathit{groups} = m.\mathit{groups}; \\
\quad s \subseteq \mathrm{dom}\ m.\mathit{readerCriteria} \bullet \\
\qquad \mathit{generateText}\ (m, s) = \\
\qquad\quad (\textbf{let}\ \mathit{cond} == \mathit{satisfiableConditions}\ (\mathit{applySelection}\ (s, ml)) \bullet \\
\qquad\quad \mathit{append}\ (m.\mathit{textStructure}, \\
\qquad\qquad \mathit{findSuccessor}\ (m.\mathit{textStructure}, m.\mathit{textStructure}.\mathit{startNode}, \mathit{cond}), \\
\qquad\qquad \mathit{cond}, \\
\qquad\qquad \langle\,\rangle))
\end{array}
$$

Hier wird nun lediglich auf die eben definierte Funktion *append* zurückgegriffen, die mit dem richtigen Nachfolger des Startknotens beginnt (der Startknoten selbst taucht folglich nicht im Text auf und dessen Bedingung wird nicht betrachtet). Eine leere Sequenz bildet den Anfangswert für den Text, welcher dann stückweise erst aufgebaut wird. Aus der Auswahl von Leserkriterien wird dann zunächst eine Menge von elementaren Kriterien ermittelt, die durch diese Leserkriterien wahr werden; daraus wird dann die Menge der erfüllenden Bedingungen gemacht, die die beiden Hilfsfunktionen benötigen.

Dies beendet die formale Spezifikation der vorgeschlagenen Modellierungskonzepte. Wie eingangs erwähnt, ist dies nur eine teilweise Spezifikation. *Teilweise*, da sämtliche Mittel fehlen, als Autor ein solches Modell schrittweise zu erstellen, sowie einige speziellere Anforderungen nicht genauer modelliert wurden. Beispielsweise sollen Gruppen sowohl geordnete als auch ungeordnete Gruppierungen von Leserkriterien darstellen – dies ist je-

doch nicht direkt mit der Formalisierung der Gruppen als Menge vereinbar. Es wäre auch wünschenswert, den Prozess des Zerteilens eines bereits vorhandenen Textes in Fragmente spezifizieren zu können.

Die Spezifikation konzentriert sich auf die statischen Aspekte eines bereits abgeschlossenen Modells, und berücksichtigt dabei nicht den Prozess der Entstehung desselben. Dies würde weitere Operationen notwendig machen, um Modellbestandteile (wie Textfragmente, Knoten im Graph, Kriterien, etc.) hinzufügen, ändern oder entfernen zu können. Ebenso müssten viele Invarianten der einzelnen Komponenten abgeschwächt werden, um einen unfertigen Zustand des Modells repräsentieren zu können. Dies könnte erreicht werden, indem zunächst Schemata für ein „unfertiges“ Modell definiert werden, welche dann in weiteren Schemata um die notwendigen Prädikate erweitert werden, um ein „fertiges“ Modell mit allen Anforderungen daran zu repräsentieren. Im Sinne einer verständlichen Spezifikation und im Hinblick auf den sonst nötigen Umfang wurde aber davon abgesehen.

Kapitel 5. Vorschlag einer Arbeitsweise für den Autor

Ausgehend von den Erfahrungen bei der Erstellung der Prototypen sowie den nötigen Zielstellungen, die an einen solchen Text gestellt werden, soll hier ein kurzer Vorschlag erfolgen, wie Autoren vorgehen können, um solche dynamischen Texte zu verfassen. Ähnlich wie Hypertext haben dynamische Texte einige Unterschiede zu traditionellen linearen Texten. Dies erfordert möglicherweise andere Arbeitsweisen, um gute Ergebnisse zu erhalten.

Zu Beginn dieser Arbeit standen zwei verschiedene und sehr gegensätzliche mögliche Arbeitsweisen im Raum: Zum einen das Schreiben eines oder mehrerer linearer Texte und dann die Adaption auf dynamische Texte durch Unterteilen in Textfragmente und Herausarbeiten von Varianten, indem für einzelne Abschnitte Alternativen verfasst werden. Zum anderen das direkte Verfassen größerer und kleinerer Textfragmente, um diese dann zum fertigen dynamischen Text zusammenzusetzen. Es ist anzunehmen, dass Autoren dynamischer Texte schon Erfahrung im Verfassen linearer Texte haben. Außerdem ist das Schreiben von mehreren Texten gleichzeitig in bruchstückhafter Form keine Hilfe dabei, diese Texte sinnvoll und kohärent zu halten. Aus diesem Grunde wird die eben genannte zweite mögliche Arbeitsweise nicht mehr betrachtet, und sich eher auf die erste Möglichkeit konzentriert.

Hierbei fängt der Autor damit an, zunächst einen oder mehrere lineare Texte zu schreiben, die das zu vermittelnde Thema für eine oder mehrere Kernzielgruppen darstellen und erläutern. Beim Verfassen dieser Texte macht sich ein Autor mindestens unbewusst schon Gedanken um Anforderungen und notwendiges Vorwissen beim Leser. Was nun folgt, ist „lediglich“ die Formalisierung dieser Gedanken in die Metaebene des Textmodells. Ein Autor sollte also möglichst früh, entweder vor oder direkt nach den anfänglichen Texten die erste Modellierung der Metaebene vornehmen, um Zielgruppen und elementare Anforderungen zu beschreiben und festzulegen. Begleitend dazu erfolgt schon die Aufteilung dieser anfänglichen Texte in große Alternativen im Textmodell, die jeweils nach den relevanten elementaren Kriterien unterschieden werden.

Gibt es in den anfänglichen Texten schon Überlappungen, so dass Teile der Texte zwischen den Varianten identisch sind, so werden diese schon im ersten Arbeitsschritt als

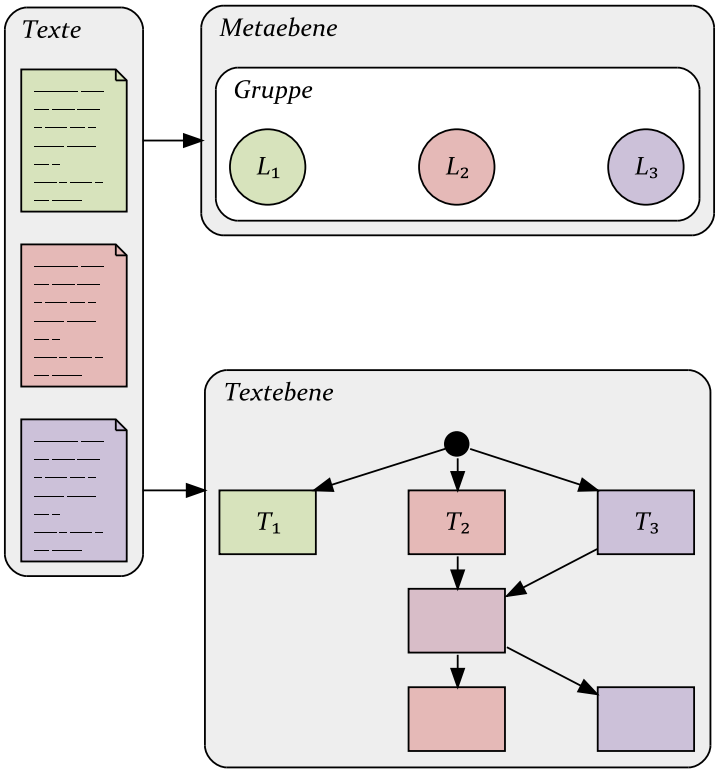


Abbildung 5.1 *Illustration des ersten Teils der empfohlenen Arbeitsweise. Hier werden zunächst Texte verfasst, die direkt in die Metaebene als Leserkriterien einer Gruppe und in der Textebene als eine Alternative übernommen werden. Dies bildet den Ausgangspunkt für weitere Unterteilungen der Texte. In diesem Falle wurde eine Überlappung der beiden letzten Texte schon direkt in der Textebene notiert; dieser Schritt entfällt, wenn es zwischen den anfänglichen Texten keine Gemeinsamkeiten gibt.*

einzelne Textfragmente extrahiert und für mehrere Varianten verwendet (siehe hierzu Abbildung 5.1). In der Metaebene bilden die zu den anfänglichen Texten gehörenden Leserkriterien in aller Regel zunächst eine große Gruppe, da sich diese Grundtexte gegenseitig ausschließen.

Anschließend werden die Texte weiter unterteilt. Dies geschieht in einem iterativen Prozess. Hierzu werden kleinere Einschübe oder Alternativen modelliert, um für einzelne Grundtexte zusätzliche Zielgruppen zu erschließen. Für die Textebene bedeutet dies, dass einzelne Knoten in mehrere zerteilt werden und an gegebenen Stellen entweder Einschübe oder Alternativen eingefügt werden. Die Metaebene wird entsprechend um neue nötige Kriterien ergänzt. Für diesen Prozess des schrittweisen Umformens eines eher flachen Graphen, der den Anfang der Modellierung darstellt (wie in Abbildung 5.1) zu dem letztendlichen Textmodell ist auch Werkzeugunterstützung denkbar, die es einem Autor ermöglicht, direkt am Text die Modellierung von Alternativen oder Einschüben vorzunehmen. Dieser Prozess wird so lange wiederholt, bis nach Meinung des Autors oder anderen Gegebenheiten alle nötigen Zielgruppen und Anforderungen abgedeckt sind.

Ein Autor sollte jedoch nicht versuchen, mit einem dynamischen Text eine gesamte Enzyklopädie zu kapseln, sondern vielmehr ein einzelnes Thema so detailliert wie nötig für verschiedene Leser oder aus verschiedenen Perspektiven aufzubereiten. Das Ziel soll nicht sein, einen idealen Text für die Gesamtheit aller Leser zu erstellen, sondern lediglich mehr als nur eine Gruppe von Lesern anzusprechen und diesen jeweils auf ihre Anforderungen (Vorkenntnisse, Interessen, ...) angepasste Texte zu liefern. Eine sinnvolle Vorstellung dessen, wo obiger Iterationsprozess aufhören sollte, ist also nötig.

Ein wesentlicher Aspekt, der in Abschnitt 2.1.3 eingeführt wurde und später zur Abgrenzung von einigen anderen Ansätzen herangezogen wurde, ist Kohärenz der resultierenden Texte. Kohärenz eines Textes lässt sich weder automatisiert messen noch garantieren, Werkzeugunterstützung in diesem Bereich ist also nur begrenzt zu erwarten. Der Autor muss also im Wesentlichen selbst dafür Sorge tragen, dass alle resultierenden Varianten kohärent sind und keine plötzlichen Brüche aufweisen. Obiger Vorschlag, wie ein Autor arbeiten könnte, ist bewusst so angelegt, dass zumindest innerhalb kürzerer Segmente die Wahrscheinlichkeit inkohärenter Übergänge zwischen angrenzenden Textfragmenten minimiert wird. Der Autor ergänzt die Grundtexte schrittweise um weitere Varianten, die jeweils neue Anforderungen umfassen, aber es kommt nie dazu, dass Textfragmente aneinandergrenzen, die vorher nicht miteinander in Kontext gestanden haben. Für längere Texte gibt es natürlich immer noch das Problem von Bezügen über große Strecken hinweg. Falls beispielsweise Bezug genommen wird auf etwas, was fünf Kapitel vorher erwähnt wurde und der Inhalt, auf den sich bezogen wird, in einer Variante nicht existiert, so ist dies etwas, was mit dem hier beschriebenen Modell und der Arbeitsweise nur schwer vermeidbar ist. Einige Gedanken, wie Werkzeuge dabei helfen können, Kohärenz zu sichern (auch in dem eben erwähnten problematischen Fall) sind im Ausblick in Kapitel 8 erwähnt.

Kapitel 6. Umsetzung

Basierend auf den in Abschnitt 4.3 beschriebenen Konzepten wurde eine prototypische Implementierung erstellt. Diese umfasst die Trennung des Modells in eine Text- und Metaebene sowie das Erstellen von Textvarianten aus dem Modell.

Da die Umsetzung nur prototypisch erfolgen sollte, wurden einige Einschränkungen vorgenommen. Zunächst wurde für die Implementierung kein besonderes Augenmerk auf Wiederverwendbarkeit oder Erweiterbarkeit gelegt. Während sicherlich eine durchdachte Softwarearchitektur sowohl als Prototyp als auch als späteres Produkt einsetzbar wäre, so ist doch der anfängliche Aufwand *nur* für einen Prototypen recht hoch. Infolgedessen ist die hier erarbeitete Implementierung statisch gehalten und dient nur zur Exploration der vorgestellten Konzepte, nicht als Basis für spätere Erweiterungen.

Die Modellierung von Textfragmenten wurde auf Text ohne Formatierung (*plain text*) beschränkt, um die Komplexität weiter zu reduzieren. Eine grafische Benutzeroberfläche wurde aus Zeitgründen nicht umgesetzt.

Die in Abschnitt 4.3 erläuterten Konzepte sind vollständig umgesetzt. Hierzu gehören die Meta- und Textebene, die Modellierung von Kriterien anhand von elementaren und Leserkriterien sowie Gruppierung von Leserkriterien. Das Erstellen einer Textvariante aus einem Modell ist ebenfalls so, wie es in Abschnitt 4.3.4 spezifiziert wird, umgesetzt. Eine Abweichung von der Spezifikation bilden jedoch die Prädikate, die in der Spezifikation den „fertigen“ Zustand eines Modells garantieren. Wie schon im entsprechenden Abschnitt angemerkt, geht die Spezifikation nicht auf einen „unfertigen“ Zustand des Modells ein – im Gegensatz zur vorliegenden Implementierung. Dies zieht jedoch nach sich, dass im gleichen Moment die Überprüfung der Invarianten für ein fertiges Modell *nicht* umgesetzt sein kann. Es wurde allerdings eine Funktion implementiert, die ein Modell auf solche Probleme hin überprüft (siehe hierzu Abschnitt 6.2).

Teile der Implementierung wurden mittels Unit-Tests auf korrektes Funktionieren hin überprüft.

6.1 Datenstrukturen

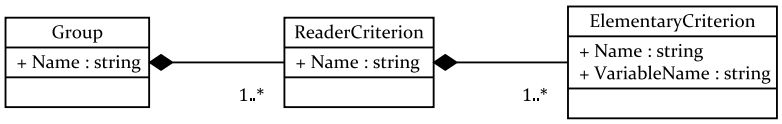


Abbildung 6.1 *UML-Klassendiagramm der Klassen der Metaebene.*

Die Metaebene wurde in drei Klassen umgesetzt, die elementare Kriterien, Leserkriterien sowie Gruppen darstellen. Gruppen gruppieren Leserkriterien, welche ihrerseits wiederum elementare Kriterien umfassen. Analog zur Nutzung von Basistypen (*Group*, *ECriterion*, *RCriterion* und *Fragment* waren solche) in der Spezifikation sind die hier modellierten Klassen nur Datenstrukturen, die ihre Daten nicht benutzen. Dies obliegt in der vorliegenden Implementierung anderen Komponenten.

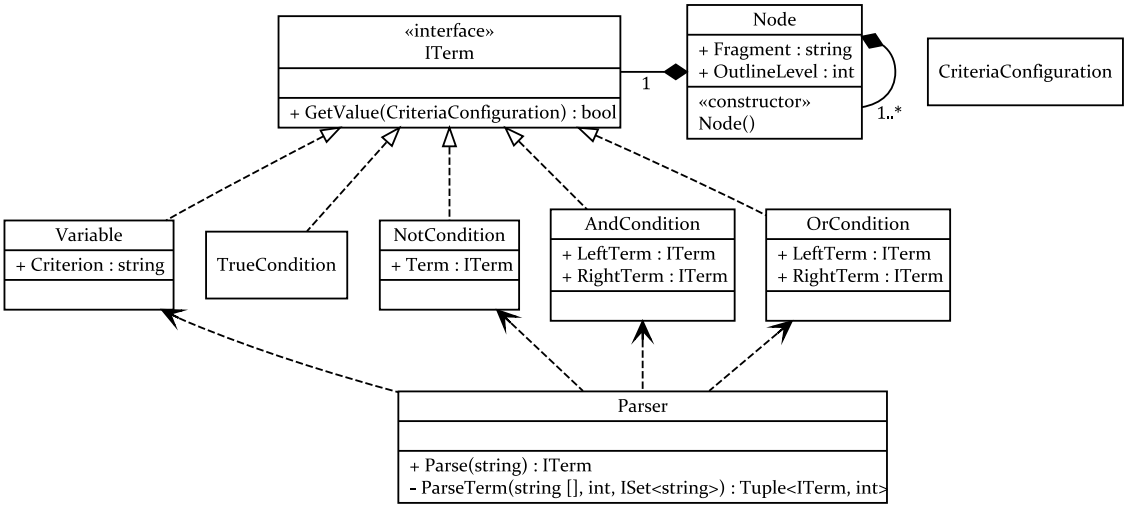


Abbildung 6.2 *UML-Klassendiagramm der Klassen der Textebene.*

Den Kern der Textebene bildet die Klasse *Node,* die einen Knoten im Graphen darstellt. Ein Knoten beinhaltet seine Nachfolger, infolgedessen ist eine separate Modellierung eines Graphen nicht notwendig. Der Startknoten des eigentlichen Textmodells findet sich noch nicht in diesem Diagramm wieder, da (anders als in der Spezifikation) für die Meta- und Textebene in dieser Implementierung nur die einzelnen Komponenten umgesetzt wurden, jedoch diese nicht in entsprechenden Typen gekapselt wurden. Der Startknoten gehört letztendlich zum Modell, welches weiter unten beschrieben wird.

Der Großteil der Klassen in der Textebene umfasst jedoch die Modellierung der Kriterien in den Knoten. Hierfür dient die Schnittstelle *ITerm*, die einen booleschen Ausdruck darstellt und eine einzige Methode bereitstellt, um sich selbst im Kontext einer Konfiguration elementarer Kriterien auszuwerten. Diese Schnittstelle wird von den verschiedenen möglichen Ausdrücken implementiert: Die Konstante *true*, Variablen (die elementare Kriterien repräsentieren), sowie die booleschen Operatoren *not*, *and* und *or*. Weiterhin wurde

ein Parser implementiert, der eine simple Grammatik boolescher Ausdrücke umsetzt. Dies dient im Wesentlichen dazu, ein Modell einfacher erstellen zu können, da der Syntaxbaum nicht immer explizit angegeben werden muss. Auf umfangreiche Fehlerbehandlung beim Parsen fehlerhafter Ausdrücke wurde entsprechend ebenfalls verzichtet.

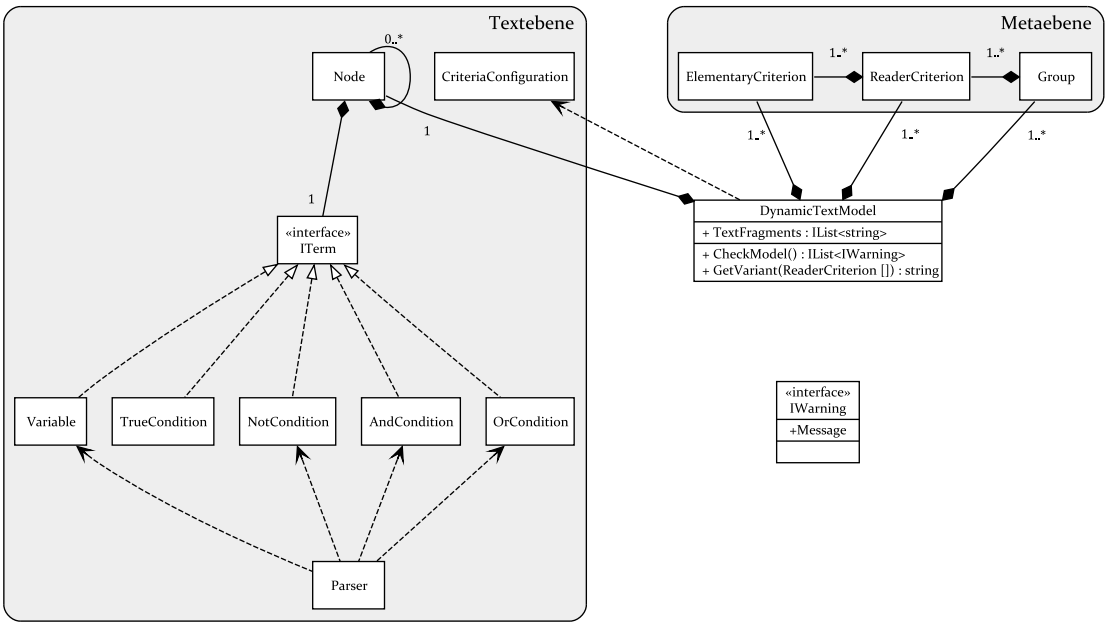


Abbildung 6.3 *Teilweise vereinfachtes UML-Klassendiagramm sämtlicher Klassen der Implementierung. Die Klassen, die jeweils zur Meta- und Textebene gehören, wurden entsprechend gruppiert. Die verschiedenen Warnungen, die* IWarning *implementieren, wurden hier aus Gründen der Übersichtlichkeit nicht dargestellt.*

Zusammengeführt wird dies alles in der Klasse *DynamicTextModel*, welche ungefähr dem Typ *Model* in der Spezifikation entspricht. Hier finden sich sämtliche Bestandteile des Modells: die elementaren und Leserkriterien nebst Gruppen sowie Textfragmente und der Graph, der das Textmodell repräsentiert, beginnend mit einem Startknoten.

Zwei Methoden, die Kernfunktionen des Konzeptes implementieren wurden als Erweiterungsmethoden in C#[15] implementiert, werden hier jedoch der Übersichtlichkeit halber als Instanzmethoden dargestellt. Diese werden in den folgenden zwei Abschnitten kurz vorgestellt.

[15] Erweiterungsmethoden (*extension methods*) in C# sind „syntaktischer Zucker“ für statische Methoden, die als erstes Argument ein bestimmtes Objekt erwarten. Diese können auf selbigen Objekten aufgerufen werden als wären sie Instanzmethoden, die in der jeweiligen Klasse deklariert wären (25). Sie haben jedoch, da sie statische Methoden einer anderen Klasse sind, nur Zugriff auf das öffentliche Interface.

6.2 Validierung des Modells

Fast alle Klassen des Modells sind lediglich Datenstrukturen ohne Validierung ihrer Daten oder sonstige Einschränkungen. Während ein Modell erstellt wird, gibt es auch häufiger Inkonsistenzen, die beispielsweise von den in der Spezifikation aus Abschnitt 4.3.4 definierten Prädikaten gar nicht zugelassen werden würden. Dies umfasst beispielsweise Handlungen wie Hinzufügen oder Umbenennen eines Kriteriums.

Aus diesem Grunde wurde das Überprüfen des Modells als separate Methode implementiert, die bei Bedarf aufgerufen werden kann. Folgende Modellierungsprobleme und Fehler werden durch die Validierung erkannt und gemeldet:

Elementare Kriterien, die von keinem Leserkriterium referenziert werden. Da ein Leser nur Leserkriterien wählen kann, würden solche elementaren Kriterien immer auf ihrem Standardwert verbleiben. Folglich müssten sie auch gar nicht modelliert werden.

Ungenutzte elementare Kriterien. Für elementare Kriterien, die von keiner Bedingung an Knoten im Graphen verwendet werden, ist ihr Wert im Grunde egal, weshalb sie ebenfalls nicht benötigt werden.

Leserkriterien, die elementare Kriterien umfassen, die nicht im Modell existieren. Da nirgends gesichert ist, dass Leserkriterien nur elementare Kriterien aus dem gleichen Modell referenzieren können, könnte es theoretisch vorkommen, dass ein Leserkriterium auf ein elementares Kriterium abbildet, welches im Modell nicht existiert. Ein möglicher Fall hierfür wäre, wenn elementare Kriterien gelöscht werden.

Gruppen, die Leserkriterien beinhalten, die nicht im Modell existieren. Analog zum obigen Fall kann das gleiche auch für Gruppen und Leserkriterien vorkommen.

Knotenbedingungen, die nicht existente elementare Kriterien nutzen. Es kann passieren, dass Bedingungen elementare Kriterien als Variablen benutzen, die es im Modell nicht gibt.

Zyklen im Graphen. Das Textmodell ist ein gerichteter Graph. Für das korrekte Funktionieren, insbesondere zum Erstellen der Varianten aus dem Modell, darf der Graph keine Zyklen haben. In diesem Fall wird für jeden Knoten, der auf einem Kreis liegt, eine Warnung zurückgegeben.

Knoten mit leeren Textfragmenten. Streng genommen ist dies kein Fehler, kann aber potentiell auf weitere Probleme im Modell hinweisen. Da ein solcher Knoten keinerlei Auswirkung auf eine Variante hat, ist er im Grunde genommen nicht nötig.

Inkonsistente Bedingungen an Knoten. Es muss gemäß dem in Abschnitt 4.3.2 beschriebenen und in Abschnitt 4.3.4 spezifizierten Algorithmus für jede mögliche Konfiguration elementarer Kriterien genau einen eindeutigen Pfad durch den Graphen geben. Knoten mit mehreren Nachfolgern, von denen aus es keinen möglichen Nachfolgeknoten (aufgrund der Bedingungen) gibt, sind ein Problem, ebenso wie Knoten, die mehrere mögli-

che Nachfolger haben. Diese Warnungen werden mit dem Knoten und den jeweilig ausgewählten *Leserkriterien* zurückgegeben, die zu dem beschriebenen Problem führen.

Die beschriebenen Überprüfungen stellen die möglichen schweren Fehler (Zyklen, inkonsistente Bedingungen) und kleinere Probleme (Knoten ohne Textfragmente, ungenutzte Kriterien) dar, die beim Modellieren eines dynamischen Textes auftreten könnten. Diese sind zunächst auf die reine Modellierung der Datenstruktur beschränkt. Denkbar sind hier einige Erweiterungen, die in einer Benutzeroberfläche den Autor unterstützen können. Dies wird in Kapitel 8 kurz angedacht.

Der Quelltext der Validierungsmethode findet sich als Beispiel einschließlich der Dokumentations-Kommentare in Anhang C.

6.3 Erstellen der Varianten

Der wichtigste Punkt dieser prototypischen Umsetzung ist zweifellos das Erstellen der Varianten aus dem Modell. Dies folgt der in Abschnitt 4.3.4 vorgestellten Spezifikation und erstellt aus einer Auswahl von Leserkriterien eine Variante in Textform. Der Algorithmus kann ungefähr folgendermaßen beschrieben werden:

1. Beginne mit dem Startknoten als aktuellen Knoten.
2. Untersuche die Nachfolger des aktuellen Knotens
   - 2a. Falls nur ein Nachfolger existiert, setze diesen Nachfolger als aktuellen Knoten.
   - 2b. Ansonsten wähle den ersten (in einem korrekten Modell einzigen) Nachfolger als aktuellen Knoten, bei dem die Bedingung im Kontext der gewählten Kriterien zutrifft.
3. Falls die Bedingung des aktuellen Knotens zutrifft, hänge das zugehörige Textfragment an den Text an.
4. Solange der aktuelle Knoten Nachfolger hat, fahre mit Schritt 2 fort.

Obwohl dieser Algorithmus für das hier entwickelte Modell zentral ist, ist er vergleichsweise einfach. Obwohl Knoten, deren Bedingung nicht zutrifft, eigentlich übersprungen werden sollten, werden sie dennoch betrachtet, sie haben nur keine Auswirkung auf den resultierenden Text. Dies ist darin begründet, dass der jeweilige Algorithmus zum Finden des Nachfolgers auch für Knoten, die eigentlich übersprungen werden, angewandt werden muss.

Kapitel 7. Fallbeispiel

Als ein Fallbeispiel für komplexer modellierte Texte dient hier eine Einführung in das Textsatzsystem LaTeX von Andreas Dähn (22), die freundlicherweise für diesen Zweck zur Verfügung gestellt wurde. Der ursprüngliche Text war die schriftliche Ausarbeitung eines einführenden Vortrags, der an der Universität Rostock gehalten wurde und richtete sich an Studenten der Informatik.

Dieser Text wurde als Ausgangspunkt für einen dynamischen Text verwendet, welcher neben der allgemeinen Einführung in LaTeX auch gezieltere Themenbereiche sowie andere Zielgruppen abdecken sollte. Das vorliegende Material eignete sich dafür recht gut, da LaTeX ein System ist, welches in vielen Fachbereichen Anwendung findet und für nahezu jede besondere Anforderung entsprechende Pakete bietet. In diesem Kapitel erfolgen nur die Beschreibung der Vorgehensweise und die Modellierung der Metaebene. Der resultierende Text, der auf die aus Modellierungssicht interessanten Teile gekürzt wurde, findet sich mitsamt Kenntlichmachung von Alternativen und Einschüben und ihren Bedingungen in Anhang D.

Zunächst wurde, gemäß dem in Kapitel 5 vorgeschlagenen Arbeitsablauf, der Text direkt übernommen. Die einzige ungefähre Anforderung, die dem ursprünglichen Text zugrunde lag, war die Eingrenzung auf Informatikstudenten als Leser. Dies bringt einige implizite Annahmen mit sich, beispielsweise Vertrautheit mit einem Texteditor oder einem Befehlszeilen-Interface. An dieser Stelle wurde, obwohl möglich und sinnvoll, im resultierenden dynamischen Text keine weitere Unterscheidung vorgenommen. Würde die Modellierung verfeinert werden, wäre dies noch eine mögliche Ergänzung.

An einigen Stellen im Text fanden sich Alternativen, die jeweils unterschiedliche Lesergruppen ansprechen sollten. Diese wurden entweder mit Nebensätzen, Klammern oder Fußnoten gelöst, beispielsweise die Anleitung zur Installation von LaTeX oder der folgende Teil:

> *„Um zu überprüfen, ob die Installation von LaTeX erfolgreich verlaufen ist, gebe man in einer Shell*[16] *den Befehl* `latex` *ein. Wenn keine Fehlermeldung, sondern ein Text*

[16] Windows-Benutzer starten unter Windows 2000 und XP eine `cmd`; unter Windows 95, 98, 98SE, ME eine `command.com`, wer ein Linux oder Unix benutzt nimmt die Shell seiner Wahl.

> *im Stile von* `This is *Tex. Version ***` *ausgegeben wird, so war die Installation erfolgreich.“*

Die Fußnote in diesem Beispiel nimmt die Unterscheidung nach Betriebssystem vor, was sich in einem dynamischen Text eleganter lösen lässt, womit die Fußnote nicht mehr nötig ist. Dies führt auch direkt zu einer ersten Gruppe von Leserkriterien:

- **Gruppe: Betriebssystem**
  - Windows[17]
  - Linux
  - BSD
  - Mac OS X
  - anderes

Diese Unterscheidung wird am deutlichsten in der Installationsanleitung sowie in kleineren Details wie dem oben gezeigten. Hauptzweck ist dabei, die für den Leser nicht relevanten Teile aus dem Text zu entfernen. Gerade die Installationsanleitung kann auch weggelassen werden, falls LaTeX schon installiert ist:

- nicht gruppiert: LaTeX ist schon installiert

An anderer Stelle geht der Ausgangstext kurz auf die LaTeX-Befehle zur Gliederung eines Textes ein:

> *„Um eine Gliederung zu erstellen, benötigt man folgende Befehle (diese gelten in der Dokumentenumgebung* `article` *– für* `book` *gelten andere),*
>
> - `\section{Kapitel}`
> - `\subsection{Unterkapitel}`
> - `\subsubsection{Unter-Unterkapitel}`
>
> *Für* `book` *ist* `section` *durch* `chapter` *zu ersetzen.“*

Hier wurde in zusätzlichen Anmerkungen deutlich gemacht, dass sich diese je nach Umgebung unterscheiden. Für den dynamischen Text wurde entsprechend eine weitere Gruppe Leserkriterien modelliert, die die Zielstellung des Lesers an seine Nutzung von LaTeX abbildet:

- **Gruppe: Zielstellung**
  - Buch
  - wissenschaftliche Veröffentlichung
  - Vortragsfolien

[17] Die ursprüngliche Fußnote hatte hier noch eine Unterscheidung in unterschiedliche Windows-Versionen vorgenommen. Da sich die Marktanteile dieser Versionen seitdem sehr geändert haben, wurde für dieses Beispiel auf die weitere Unterteilung verzichtet.

Das ist nur eine sehr grobe Einordnung, insbesondere da LaTeX für sehr viele Zwecke eingesetzt werden kann. Im Sinne dieses Beispiels soll das aber ausreichen. Die Zielstellungen wirken sich auf verschiedene Aspekte von LaTeX aus. Zunächst sind es unterschiedliche Dokumentvorlagen, die teilweise anders genutzt werden oder andere Befehle bereitstellen. Ebenso folgen daraus besondere Anforderungen, die jeweils behandelt werden sollten. Beispielsweise ist für Vortragsfolien die Formatierung von Kopf- und Fußzeile unerheblich, für das Setzen von Büchern jedoch nicht. Wissenschaftliche Veröffentlichungen brauchen fast immer ein Literaturverzeichnis; folglich ist dort eine Einführung in BibTeX unabdingbar.

Der Fachbereich der Autoren bringt ebenfalls eigene Anforderungen an verfasste Texte mit sich. Der ursprüngliche Text richtete sich an Informatikstudenten, weshalb einige LaTeX-Pakete Erwähnung fanden, die sicher nicht für jeden Nutzer relevant sind. Aus diesem Grunde wurde eine weitere Unterscheidung bezüglich des Fachgebiets des Lesers (und zukünftigen Autors von LaTeX-Dokumenten) vorgenommen:

- **Gruppe: Fachgebiet**
  - Informatik
  - Mathematik
  - Chemie

Hier wären noch viele weitere Fachgebiete denkbar, der Wartungsaufwand des dynamischen Textes bleibt aber relativ gering, da in den meisten Fällen diese Fachgebiete lediglich bestimmte Abschnitte sichtbar machen oder verbergen. Beispielsweise benötigen Mathematiker Formelsatz, so wie Informatiker häufig Quelltext mit einbinden müssen. In der Chemie werden chemische Gleichungen und Strukturformeln benötigt. In all diesen Fällen gibt es bestimmte Pakete in LaTeX, die den Autor dabei unterstützen. Da der zweite Teil der LaTeX-Einführung im Wesentlichen aus einer Auflistung nützlicher Pakete und Befehle besteht, lassen sich fachspezifische Pakete und Befehle relativ einfach durch Hinzufügen weiterer Unterabschnitte einführen.

Ein früher Abschnitt im anfänglichen Text befasst sich kurz mit den Unterschieden zwischen LaTeX und einer Textverarbeitung. Da hier auch auf eventuell vorhandene Vorkenntnisse eingegangen werden kann, wurden daraufhin unterschiedliche Stufen an Vorwissen bezüglich der Textverarbeitung Microsoft Word als weitere Kriterien modelliert:

- **Gruppe: Vorwissen in Word**
  - kein Vorwissen
  - nur Textformatierung
  - Formatvorlagen und Verfassen längerer Dokumente

Tatsächlich sind die Unterschiede zwischen einer Textverarbeitung und LaTeX deutlich geringer, wenn erstere gekonnt eingesetzt wird. Hierauf kann mit obigen Kriterien auch Bezug genommen werden.

Zu guter Letzt bilden die oben beschriebenen und eingeführten Leserkriterien auf eine Reihe von elementaren Kriterien ab. Diese werden in der untenstehenden Tabelle kurz genannt.

Gruppe	Leserkriterium	Elementare Kriterien
Betriebssystem	Windows	Betriebssystem: Windows
	Linux	Betriebssystem: Linux
	BSD	Betriebssystem: BSD
	Mac OS X	Betriebssystem: Mac OS X
	anderes	Betriebssystem: anderes
Zielstellung	Buch	Dokumentenklasse: `book` Abbildungen Tabellen gespiegelte Seitenanordnung
	wissenschaftlicher Artikel	Dokumentenklasse: `article` BibTeX Abbildungen Tabellen Spaltensatz
	Vortragsfolien	Dokumentenklasse: `beamer` Abbildungen
Fachbereich	Informatik	Quelltext-Listings Formelsatz
	Mathematik	Formelsatz
	Chemie	Formelsatz chemische Formeln
Vorwissen in Word	kein Vorwissen	—
	nur Textformatierung	Leser hat nur Erfahrung mit Schriftformatierung in Word
	Formatvorlagen und Verfassen längerer Dokumente	Leser hat Erfahrung mit Formatvorlagen in Word
	LaTeX ist schon installiert	LaTeX ist schon installiert

Die oben aufgeführten elementaren Kriterien werden im in Anhang D ausformulierten Text verwendet.

Kapitel 8. Zusammenfassung und Ausblick

8.1 Zusammenfassung

In dieser Arbeit wurden sogenannte dynamische Texte vorgestellt, die gewisse Vorteile linearer Texte mit denen von Hypertexten verbinden. In der hier erläuterten Form ermöglichen sie einem Autor, verschiedene Textvarianten zu modellieren, die vom Leser anhand von Kriterien vor dem Lesen ausgewählt werden können. Weiterhin wurde ein Vorschlag unterbreitet, wie Autoren beim Verfassen solcher Texte vorgehen können, da es dabei andere An- und Herausforderungen gibt als beim Verfassen traditioneller linearer Texte. Ebenso wurden die vorgestellten Modellierungskonzepte sowohl formal spezifiziert als auch in Form einer prototypischen Implementierung umgesetzt.

Ausgehend von den in dieser Arbeit vorgestellten Konzepten lassen sich auch zusätzliche Punkte finden, an denen diese Gedanken weitergeführt werden können. In Abschnitt 1.1 wurden drei Kernthemen genannt, die für dynamische Texte relevant sind. Von diesen wurden in dieser Arbeit nur die ersten beiden betrachtet.

8.2 Mögliche Weiterführungen des Themas

8.2.1 Autorengruppen

In dieser Arbeit wurde sich darauf beschränkt, einen einzelnen Autor zu betrachten. Gerade Lehrmaterialien (Lehrbücher zum Beispiel) werden aber oft von mehreren Autoren verfasst. Denkbar wären hier Änderungen bzw. Erweiterungen der in Kapitel 5 vorgeschlagenen Arbeitsweise, so dass diese auch für mehrere Autoren nutzbar sind. Der derzeitige Vorschlag ließe sich allerdings in beiden Phasen gut parallelisieren; dennoch war dies nicht Ziel der Arbeit.

8.2.2 Rich Text und andere Medien

Wie kurz in Kapitel 6 angesprochen, geht die Umsetzung bisher nur auf Text ohne Formatierungen ein; in der Spezifikation wurden Textfragmente bewusst nicht konkreter beschrieben. Wünschenswert wäre die Unterstützung für formatierte Texte, da Formatierung dem Lesefluss helfen kann und daher oftmals vom Leser erwartet wird. Dies birgt jedoch noch einige Komplexität. Eine Möglichkeit wäre, die jetzige technische Basis der Umsetzung zu nutzen und statt eines fertigen Textes das Ergebnis in einer Markupsprache auszugeben, die Formatierungen beschreiben kann, beispielsweise LaTeX, HTML oder RTF.

Bei diesem Ansatz müsste allerdings eine Strategie entwickelt werden, wie bei überlappenden Formatanweisungen vorzugehen ist. Eine Alternative wäre das Erzwingen der Bedingung, dass Textfragmente mitsamt ihrer Formatierung in sich abgeschlossen sind und keinen Einfluss auf angrenzende Fragmente haben können. Eine andere Möglichkeit wäre noch, die Formatierung völlig separat von den Textfragmenten, quasi in einem zusätzlichen Textmodell, zu halten. Diese, sowie weitere Probleme könnten in zukünftigen Arbeiten untersucht werden.

Analog zu einer Erweiterung auf *Rich Text* ist auch eine Erweiterung auf andere Medien vorstellbar. Statt Texte zu betrachten, ließen sich die gleichen Konzepte auch für Audio- oder Videodaten umsetzen (hier ist die Linearität allein schon durch das Medium sehr stark vorgegeben). An den Hörer angepasste Hörbücher ließen sich im Grunde auf die gleiche Weise erzeugen wie an den Leser angepasste Texte. Bild- und Tonspuren müssten aller Wahrscheinlichkeit nach aber separat betrachtet werden, um gerade an Übergangsstellen Aspekte wie Hintergrundmusik mit Sprache darüber problemlos behandeln zu können. Dies ist im Radiobereich schon bekannt und dort gibt es auch seit längerem unter dem Begriff *Voicetracking* Lösungen, die genau auf diese Problematik eingehen.

8.2.3 Entwicklung eines grafischen Autorenwerkzeugs

In Kapitel 5 und an anderen Stellen wurde Werkzeugunterstützung für den Autor angesprochen. Diese Arbeit befasste sich im Wesentlichen mit den Grundlagen, der Erarbeitung von Modellierungskonzepten für dynamische Texte und einer prototypischen Umsetzung dieser Konzepte. Gerade im Hinblick auf den in Kapitel 5 beschriebenen Arbeitsablauf wäre ein grafisches Autorenwerkzeug denkbar, um diesen zu unterstützen.

Beispielsweise besteht der Iterationsprozess, der die anfänglichen Texte in weitere Varianten unterteilt, im Wesentlichen daraus, Teile der bisherigen Texte in weitere Fragmente zu unterteilen und weitere Alternativen zu erstellen. Hier wären einige Möglichkeiten der direkten Manipulation denkbar, so dass der Autor direkt Text einer Variante auswählen könnte und in ein neues Fragment überführen, welches dann in den Graphen des Textmodells eingefügt wird. Konsequent weitergeführt bedeutet dies: Die Modellierung des Graphen könnte komplett versteckt und dem Autor lediglich den Text der momentan betrachteten Variante gezeigt werden. Diese kann er dann direkt bearbeiten kann, was im Hintergrund zu den Änderungen am Modell führt (siehe Abbildung *8.1*).

Denkbar wäre auch, die Metaebene für ein Autorenwerkzeug nochmals zu abstrahieren, und den Autor explizit Zielgruppen und Interessen modellieren zu lassen (welche auf Leserkriterien mit entsprechender automatischer Gruppierung abgebildet werden könnten). Von den in dieser Arbeit vorgestellten Modellierungskonzepten wird lediglich angenommen, dass sie konzeptuell dicht an dem sind, was ein Autor modellieren würde; möglicherweise müsste ein Werkzeug dem Rechnung tragen, indem Teile weiter abstrahiert oder gar ganz versteckt werden.

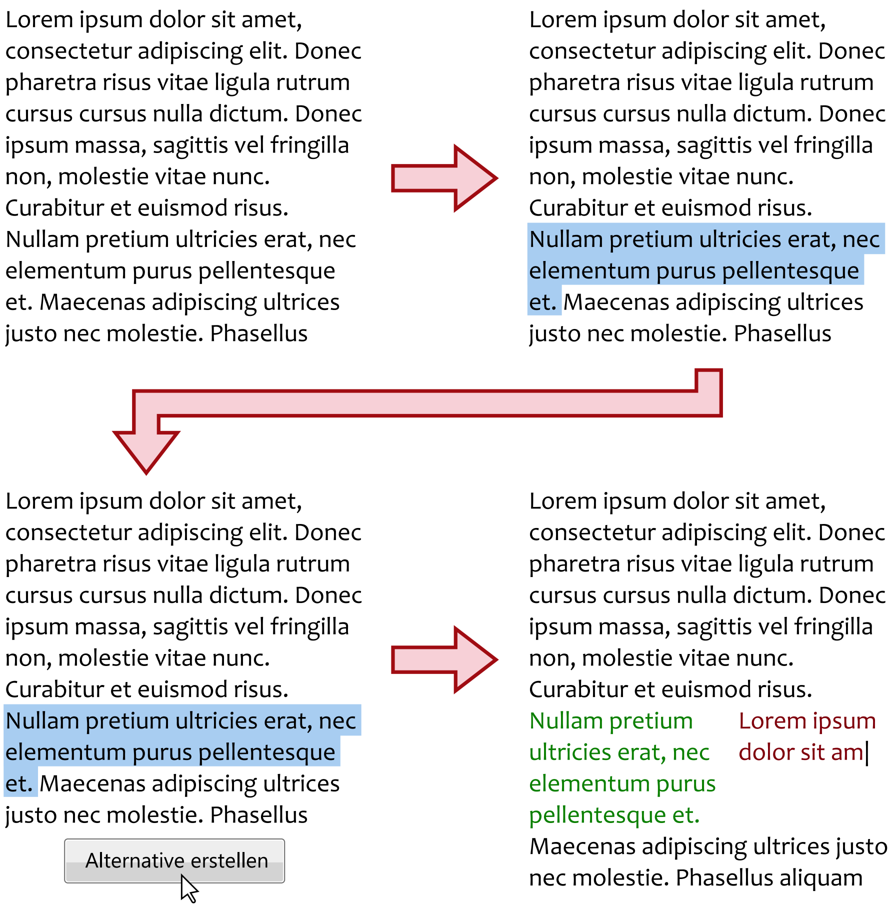


Abbildung 8.1 *Mockup einer ungefähren Idee der Benutzerinteraktion zum Erstellen von Alternativen ausschließlich anhand des Textes ohne explizit Änderungen am Graphen des Textmodells vorzunehmen. Der Autor könnte einen Teil des Textes markieren, um daraus eine neue Alternative zu erstellen. Die beiden farblich hervorgehobenen Stücke im vierten Bild stellen die beiden Teilpfade jener Alternative dar, die unabhängig bearbeitet werden könnten.*

Ein Problem können immer noch inkohärente Übergänge zwischen aufeinanderfolgenden Textfragmenten sein. Die Handhabung ebendieser wird zwar durch den in Kapitel 5 beschriebenen Prozess vereinfacht, aber birgt im Einzelfall immer noch Probleme. Ein Autorenwerkzeug könnte aber beispielsweise alle möglichen Übergänge zwischen Textfragmenten betrachten und sie einordnen in *kohärent* und *möglicherweise nicht kohärent*. Übergänge zwischen Fragmenten, die direkt unverändert aus den Ursprungstexten übernommen wurden, wären automatisch als kohärent markiert. Wann immer danach zwei Fragmente aneinanderstoßen, die vorher nie in diesem Kontext kohärent waren, wäre die Einstufung *möglicherweise nicht kohärent*, die vom Autor jeweils geändert werden kann, wenn er der Meinung ist, die Einstufung sei ungerechtfertigt. Gleiches gilt auch, wenn sich Textfragmente ändern; in diesem Falle würden die Grenzen zu benachbarten Fragmenten ebenso eine Warnung hervorrufen. Dies ist natürlich viel zusätzlicher Aufwand für den Autor und infolgedessen wären auch Möglichkeiten denkbar, dies in den beschriebenen Arbeitsablauf einzuarbeiten und die beim Erstellen von Alternativen entste-

henden Übergänge auch zunächst als kohärent anzusehen, was die Anzahl der Warnungen reduzieren sollte.

Eine weitere Möglichkeit, den Autor beim Sicherstellen von Kohärenz zu unterstützen, wäre, die Einarbeitung von „Brückenfragmenten“ explizit zu modellieren. Falls sich beispielsweise zwei Textfragmente nicht ohne Weiteres verbinden lassen, sind eventuell zusätzliche nötig, um sie an den umgebenden Text anzupassen (dies wären die „Brückenfragmente“). Nun könnte das Textmodell dahingehend erweitert werden, dass jeder Knoten neben dem eigentlichen Textfragment für jeden Vorgänger und jeden Nachfolger je ein weiteres Brückenfragment enthalten kann, welches dann ausschließlich dazu da ist, den Anschluss zum umgebenden Text zu glätten.

Gleichsam wurde angemerkt, dass es gerade bei längeren Texten Probleme über große Leseentfernungen geben kann, wenn zum Beispiel in einer Variante Abschnitte referenziert werden, die in dieser Variante gar nicht vorhanden sind. Vorstellbar ist hier eine Lösung, bei der Querverweise ähnlich wie in LaTeX oder Word explizit modelliert werden. Damit ist für das System die Information vorhanden, dass eine Referenz existiert und es kann entsprechend gewarnt werden, falls eine Variante zwar eine Referenz aber nicht ihr Ziel enthält.

8.2.4 Benutzertests

Alle der gerade genannten Gedanken haben eins gemein: Sie gehen von einem grafischen Autorenwerkzeug aus, welches bislang noch nicht existiert. Weiterhin ist auch mehr als nur ein Prototyp nötig, um Varianten aus einem dynamischen Text zu erzeugen, die dann gelesen werden können. Ebenso ist offen, ob und wie solche dynamischen Texte Vorteile für Leser haben – dies sollte durch Beobachtungen und Befragungen von Lesern zu ermitteln sein. Die erarbeiteten Vorschläge müssen gründlich erprobt werden.

8.2.5 Weitere Einsatzmöglichkeiten

In dieser Arbeit wurde sich auf Texte in der Lehre sowie Material in interdisziplinären Arbeitsgruppen konzentriert. Weitere Anwendungsgebiete sind denkbar, in denen verschiedene Texte nutzbar sind, die das Gleiche, jedoch mit anderen Worten, beschreiben. Als prominentes Beispiel hierfür ist Creative Commons (23) zu nennen – ein Projekt welches zum Ziel hat, Lizenzen für die Verwendung von Inhalten zu verfassen, die einfache Wiederverwendung und Kooperation fördern. Diese Lizenzen werden auf drei Ebenen umgesetzt, die in Abbildung *8.2* dargestellt sind. Zum einen ist dies eine maschinenlesbare Repräsentation der Lizenz, um beispielsweise automatisiert Inhalte zu finden, die sich auf bestimmte Weise verwenden lassen. Dann gibt es eine Darstellung, die sich klarer, einfacher Sprache und Piktogrammen bedient, um Erstellern oder Nutzern von Inhalten das Verständnis der Lizenzen erleichern. Zuletzt existiert die Beschreibung der Lizenz in juristischer Fachsprache, die für die Umsetzung innerhalb des Urheberrechts nötig ist. Dies ist eine Ebene, die rein rechtlich notwendig ist, aber von einem normalen Nutzer

nicht gelesen bzw. verstanden werden muss. Die letzten beiden Ebenen existieren außerdem in angepasster Form für verschiedene Staaten bzw. Gerichtsbarkeiten. Insbesondere in den beiden letztgenannten Ebenen finden sich durchaus Parallelen zu den hier erarbeiteten Konzepten. Die Sprache der Lizenz könnte zum Beispiel durch Leserkriterien in einer Gruppe *Sprache* dargestellt werden; die Auswahl der lesbaren Kurzfassung oder des zugrundeliegenden juristischen Dokuments mit einem ungruppierten Leserkriterium *juristischer Text*.

8.2.6 Interaktivität für den Leser

Die bisher betrachteten dynamischen Texte sind zum Zeitpunkt des Lesens statisch. Dies kann durchaus ein Vorteil sein, da der Leser in diesem Falle keinerlei Interaktionsmöglichkeit benötigt und folglich auch weniger Gelegenheit für Ablenkung hat (ein häufiger Grund, weshalb Menschen ungern am Bildschirm lesen). Allerdings könnte es durchaus sein, dass der Leser Probleme mit der anfänglichen Einschätzung der Kriterien hatte. Derzeit müsste der Lesekontext verlassen werden, eine neue Variante basierend auf einer neuen Auswahl von Kriterien erstellt werden und der Leser müsste die Stelle wiederfinden, an der er aufgehört hat zu lesen (oder eine vergleichbare, falls sie in der neuen Variante nicht mehr existiert). Gerade bei nur kleinen Änderungen am Text (beispielsweise dadurch, dass anfänglich nur ein Kriterium falsch gewählt war, welches kaum Auswirkungen auf den Gesamttext hat) ist dies sehr umständlich.

Für diesen Zweck wäre es denkbar, dem Leser direkt die Möglichkeit zu geben, zumindest solche Kriterien neu zu wählen. Dies muss beispielsweise auch nicht durch die vollständi-

Abbildung 8.2 *Creative Commons stellt die jeweils gleichen Informationen auf drei verschiedenen Ebenen dar: Maschinenlesbar, in Form von Piktogrammen und kurzen Beschreibungen für Nutzer der Lizenzen sowie in juristischer Fachsprache. Abbildung aus (24) übernommen.*

ge Kriterienauswahl vom Anfang geschehen. Ein subtiler Vermerk neben dem Text, der dem Nutzer die Möglichkeit gibt festzulegen, ob er das in einem Abschnitt behandelte Thema schon kennt, könnte hier ausreichen. Umgekehrt wäre auch denkbar, dass es weiterführende Informationen gibt, die für die meisten Leser nicht relevant sind. Diese könnten sich ad-hoc einblenden lassen. Dies wäre beispielsweise eine Möglichkeit, Fußnoten oder andere Anmerkungen in einem Text darzustellen – digitale Texte sind nicht auf eine Seitendarstellung festgelegt.

Ebenfalls unter Interaktivität würde ein Punkt fallen, der anfangs als eine der Nutzungsmöglichkeiten diskutiert wurde: Iteratives Lesen des gleichen Textes, der sich jeweils leicht erweitert. Dies könnte zwar mit den in dieser Arbeit beschriebenen Modellierungskonzepten rein technisch möglich sein, benötigt aber noch zusätzliche Unterstützung einer Softwarelösung zum Anzeigen solcher Texte, da sonst der iterative Aspekt nicht umsetzbar ist.

8.3 Abschluss

Abschließen lässt sich sagen, dass dynamische Texte einen interessanten Mittelweg zwischen traditionellen linearen Texten und Hypertext bilden. Sie eignen sich insbesondere für Anwendungsgebiete, in denen Leser verschiedene Voraussetzungen oder Ziele bezüglich eines zu vermittelnden Themas mitbringen.

Anhang

Anhang A. Prototyp: Wikipedia-Artikel „Turingmaschine“

Als früher Prototyp wurde der Artikel „Turingmaschine“ aus der Wikipedia (18) als Beispiel genommen. Hier wurden aus dem ursprünglichen Text drei verschiedene gemacht mit jeweils unterschiedlichem Anspruch an das Vorwissen des Lesers. Die Anspruchsebenen, die hier gewählt wurden, sollten ungefähr passend für Schüler mit geringen Informatik- und Mathematikvorkenntnissen, Studenten des ersten Semesters und Studenten des vierten Semesters sein. Letztere sollten in meiner Erfahrung alle nötigen Voraussetzungen an Wissen mitbringen, um den kompletten Text verstehen zu können.

Der ursprüngliche Text aus der Wikipedia wurde fast unverändert als Grundlage für die umfangreichste Variante, also den Viertsemestertext, genommen. Die anderen beiden Varianten wurden durch Entfernen von größeren Fragmenten (meist ganzen Abschnitten) erlangt. Dies ist zwar eine sehr simple Methode, aus einem Text mehrere mit verschiedenem Anspruch zu machen, jedoch birgt sie natürlich Probleme hinsichtlich vernünftig gestalteter Texte. Der Originaltext war vom Aufbau und Inhalt gut lesbar; werden ganze Abschnitte oder Absätze herausgestrichen, weil sie Anspruch her zu hoch sind, so ist das Ergebnis nicht zwangsweise brauchbar. An einigen Stellen mussten also Sätze umformuliert oder eingefügt werden, damit am Ende wieder ein lesbarer Text entsteht, der nicht den Eindruck erweckt, dass Teile fehlen.

Umgesetzt wurde der Prototyp mittels HTML und JavaScript und ist somit in Web-Browsern lauffähig. Abschnitte, Sätze und auch Satzteile wurden mittels des `class`-Attributes in eine oder mehrere von drei verschiedenen Ebenen eingeteilt (`level-1` bis `level-3`). Der Einfachheit halber wurde Text, der in jeder der drei Varianten enthalten war, nicht separat ausgezeichnet und ist somit immer sichtbar. Ein wenig JavaScript-Code versteckte dann je nach ausgewähltem Anspruch zwei Ebenen und zeigte nur die dritte (zuzüglich allem, was nicht explizit gekennzeichnet war). In diesem Modell hat „zeigen“ Vorrang vor „verstecken“ gehabt. In einer tatsächlichen Implementierung sollte so etwas möglicherweise wählbar sein.

```
function hide(level) { hideShowHelper(level, 'none'); }

function show(level) { hideShowHelper(level, ''); }

function hideShowHelper(level, displayString) {
  var classToFind = 'level-' + level;
  var elements = document.getElementsByClassName(classToFind);
  for (var i = 0; i < elements.length; i++) {
        elements[i].style.display = displayString;
  }
}

function handleSelectionChanged(e) {
  var level = e.currentTarget.value;
  switch (level) {
        case '1': hide(2); hide(3); show(1); break;
        case '2': hide(1); hide(3); show(2); break;
        case '3': hide(1); hide(2); show(3); break;
  }
}

function createWidget() {
  var div = document.createElement('div');
  div.setAttribute('style', 'position: fixed; top: 2em; left: 42em;' +
                   'border: 1px solid #000; padding: 1ex; border-radius: 1ex;');

  var text = document.createElement('p');
  var label = document.createElement('label');
  label.htmlFor = 'level';
  label.textContent = 'Anspruch:'
  text.style.margin = '0';
  text.appendChild(label);
  div.appendChild(text);

  var select = document.createElement('select');
  select.name = 'level';
  select.id = 'level';
  select.appendChild(new Option('Schule', 1));
  select.appendChild(new Option('1. Semester', 2));
  select.appendChild(new Option('4. Semester', 3));
  div.appendChild(select);

  select.addEventListener('change', handleSelectionChanged, false);

  document.body.appendChild(div);
  document.body.style.maxWidth = '40em';

  select.oldValue = 1;
  hide(2);
  hide(3);
  show(1);
}

document.addEventListener('DOMContentLoaded', createWidget, false);
```

Anhang B. Prototyp: Java lernen mit unterschiedlichen Vorkenntnissen

Anders als der Turingmaschinen-Prototyp wurde dieser nicht als Implementierung umgesetzt; die Textvarianten stehen für sich allein und wurden auch so geschrieben. Zwar könnten die drei Textvarianten in ein ähnliches Gerüst eingebettet werden wie der Turingmaschinen-Prototyp, aber das Ergebnis ist wahrscheinlich nur bedingt nützlich, da dann drei große Textabschnitte existieren, von denen jeweils nur einer sichtbar und die anderen beiden versteckt sind.

Insgesamt wurden drei Varianten wurden erstellt, alle mit dem Ziel, einen Einstieg in die Programmiersprache Java zu geben. Da die Varianten bewusst kurz gehalten wurden (ungefähr eine bis zwei Seiten) ist der Einstieg nicht allzu tief. Die Texte, wie sie sind, könnten aber erweitert und vervollständigt werden um einen umfassenderen Einstieg in die Sprache zu geben.

Die Varianten gehen von folgenden Vorkenntnissen aus:

- Keine Vorkenntnisse irgendwelcher Programmiersprachen.
- Kenntnis der Programmiersprache C, kein Wissen über objektorientierte Softwareentwicklung.
- Kenntnis der Programmiersprache C# sowie objektorientierter Softwareentwicklung, kein Wissen über die Unterschiede zwischen C# und Java.

B.1 Text für Programmieranfänger

Erforderliche Kenntnisse	Angenommene Unkenntnisse
Eingeben von Texten	Jegliche Programmiersprache

Java ist eine Programmiersprache. Mit solchen Sprachen kann man Computern beibringen, Dinge zu tun die wir gerne wollen, dass sie sie tun. Computer haben keinen eigenen Willen und brauchen immer jemanden, der ihnen sagt, was sie tun sollen. Hierfür gibt es viele verschiedene Sprachen und mit den richtigen Programmen versteht ein Computer diese auch. Wir wollen hier aber zunächst nur eine Sprache benutzen, nämlich Java.

Zunächst fangen wir mit einem ganz einfachen Beispiel an: Ein Programm, welches `Hallo Welt!` ausgibt. Das sieht dann folgendermaßen aus:

```
public class Hallo {

    public static void main(String[] args) {
        System.out.println("Hallo Welt!");
    }

}
```

Das sieht nun erst mal ziemlich beängstigend aus. Tatsächlich brauchen wir nur die Zeile mit `System.out.println` dort, um `Hallo Welt` anzuzeigen. Der Rest darum herum ist erst einmal nur dazu da, dass Java unser Programm überhaupt mag. Später kann man mit diesen Teilen noch tolle Dinge machen, aber erst einmal sehen sie für fast jedes Programm gleich aus.

Was das dort oben genau macht? Oh, das ist einfach. `System.out` ist quasi die »Systemausgabe« – das, wo wir den Text hinschreiben wollen. `println` steht für »print line« was so viel bedeutet wie »drucke Zeile«. Der Teil, der sich in Klammern anschließt, beinhaltet den Text, den wir ausgeben wollten. Nach der Klammer folgt ein Semikolon »`;`«, welches bedeutet, dass die Anweisung hier zu Ende ist.

Übung 1. Schreibe ein Programm, welches statt `Hallo Welt!` den Text `Guten Abend, Welt.` ausgibt.

Wir wissen nun, wie wir Text ausgeben können. Das gleiche funktioniert auch mit Zahlen. Ersetzen wir in obigem Programm das `Hallo Welt!` durch eine `5172`, dann wird eben jene Zahl ausgegeben.

Übung 2. Schreibe ein Programm, welches statt der Zahl `5172` die Zahl `42` ausgibt.

Mit Zahlen kann man natürlich auch rechnen; die Grundrechenarten sollten sicher noch ein Begriff sein. Die Operatoren `+`, `-`, `*` und `/` repräsentieren jeweils Addition, Subtraktion, Multiplikation und Division. Schreiben wir in Java `5 * 7`, so ist das Ergebnis `35`.

Wir können solche Berechnungen auch dort durchführen, wo wir vorhin die Zahl zum Ausgeben hingeschrieben haben. Das folgende Programm verdeutlicht dies:

```
public class Rechner {

    public static void main(String[] args) {
        System.out.println(1 + 7 * 3 - 4);
    }

}
```

Hierbei wird $1 + 7 \cdot 3 - 4 = 20$ berechnet und ausgegeben. Java könnte also als Taschenrechner verwendet werden.

Übung 3. Schreibe ein Programm, welches das Ergebnis der Berechnung

$$\big((7-2)\cdot 14\big) \div 10 + 5$$

ausgibt. Beachte, dass das Divisionszeichen in Java / ist und nicht ÷.

B.2 Text für C-Programmierer

Erforderliche Kenntnisse	Angenommene Unkenntnisse
Die Programmiersprache C	Objektorientierte Programmierung und Konzepte

Java ist von der Syntax her ähnlich der Programmiersprache C, allerdings nur auf den ersten Blick. Da Java eine objektorientierte Sprache ist, gab es einige Anleihen bei C++ und auch viel neu entwickelte Syntax, da das Ziel war, die Sprache und den Compiler möglichst einfach zu halten.

Als Einstieg zunächst das klassische „Hello World“:

```
public class Hallo {

    public static void main(String[] args) {
        System.out.println("Hallo Welt!");
    }

}
```

Es illustriert die ersten grundlegenden Unterschiede schon recht gut.

In Java ist, da es eine objektorientierte Sprache ist, alles in eine Klasse gekapselt. Klassen erlauben, in erster Näherung, Daten in gewisser Form und Code zur Manipulation derselben an einer Stelle zu haben. Objektorientierte Softwareentwicklung geht weiter als das, aber das soll hier zunächst einmal genügen. Ganz grob betrachtet kann man eine Klasse als eine Struktur nebst Funktionen in einer Einheit sehen.

Was weiterhin auffällt, ist, dass man keine Bibliothek für Ein- und Ausgabe einbinden muss (wie in C `stdio.h`). Allgemein gibt es keine Headerdateien. Code und Deklaration leben an der gleichen Stelle in Java.

Damit ein C-Programm ausführbar ist, muss eine `main`-Funktion irgendwo existieren. Analog verhält es sich mit Java, welches eine `main`-Methode benötigt, damit eine Klasse ausführbar ist (die kleinste zusammenhängende Codeeinheit in Java ist eine Klasse, im Gegensatz zu einer Datei in C).

Glücklicherweise bleibt bei den meisten Kontrollstrukturen und damit dem prozeduralen Teil der Sprache das meiste beim Alten. Nehmen wir das folgende C-Programm, welches seine Argumente ausgibt:

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    int i;
    for (i = 1; i < argc; i++) {
        printf("%s\n", argv[i]);
    }

    return 0;
}
```

so übersetzt es sich folgendermaßen nach Java:

```java
public class Args {

    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            System.out.println(args[i]);
        }
    }

}
```

Da Java von sich aus Unterstützung für Arrays fester Länge mitbringt, ist auch das zweite Argument an `main()` unnötig. Arrays haben ein Feld `length`, welches besagt, wie viele Einträge sie haben.

Eine Sache, die noch zu bemerken ist, ist, dass `args` in Java tatsächlich nur die Argumente an das Programm beinhaltet, nicht jedoch den Programmaufruf selbst. Deshalb fängt die Schleife in Java auch bei `0` statt wie in C bei `1` an.

Übung 1. Übersetze folgendes C-Programm nach Java:

```c
int main(int argc, char* argv[]) {
    switch (argc) {
        case 0:
            printf("Kein Argument.\n");
            break;
        case 1:
            printf("Nur ein Argument.\n");
            break;
        case 2:
        case 3:
            printf("Zwei oder drei Argumente.\n");
            break;
        default:
            printf("Viele Argumente.\n");
            break;
    }
    int x = 2;
    x *= argc;
    if (x > 7) {
```

```
            printf("x ist > 7\n");
        } else {
            printf("x ist <= 7\n");
        }

        return 0;
    }
```

Kontrollstrukturen und andere Syntaxkonstrukte, die hier verwendet wurden, verhalten sich sehr ähnlich wie in C.

B.3 Text für C#-Programmierer

Erforderliche Kenntnisse	Angenommene Unkenntnisse
Die Programmiersprache C#	Unterschiede zwischen Java und C#

Wie auch C# ist Java eine objektorientierte Programmiersprache. Beiden Sprachen lagen ähnliche Zielstellungen zugrunde: Eine universelle Programmiersprache, die objektorientiert und einfach zu erlernen ist. Beide Sprachen nutzen sehr ähnliche Syntax und haben ähnliche Möglichkeiten.

Während sie sich oberflächlich recht ähnlich sind (beispielsweise in dem Sinne, dass Mehrfachvererbung verboten ist, außer bei Interfaces), so gibt es doch gravierende Unterschiede.

Nehmen wir ein sehr einfaches Beispielprogramm:

```
using System;

public class Beispiel
{

    public static void Main(string[] args)
    {
        for (int i = 0; i < args.Length; i++)
        {
            Console.WriteLine(args[i]);
        }
    }

}
```

Dieses gibt einfach die Befehlszeilenargumente der Reihe nach aus. Übersetzt nach Java sieht es folgendermaßen aus:

```
public class Beispiel {

    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            System.out.println(args[i]);
        }
    }

}
```

Es ist recht einfach wiederzuerkennen, aber einige Unterschiede werden schon deutlich. Zunächst einmal hat Java die Konvention, sämtliche Namen von Variablen und Methoden mit kleinem Anfangsbuchstaben zu schreiben (außer Klassennamen). Dies wird bei der `main`-Methode erzwungen, da das Programm anderenfalls nicht ausführbar ist. Weiterhin hat Java keine Aliase für Typen, so dass man `String` statt `string` schreiben muss. Der letzte wirkliche Unterschied ist die Ausgabe von Text auf der Konsole; dies geschieht nicht durch statische Methoden einer Klasse, sondern durch eine `PrintStream`-Instanz die in der `System`-Klasse vordefiniert ist.

Ein wesentlicher Unterschied zwischen Java und C# ist, dass letzteres in der Sprache selbst Unterstützung für Eigenschaften (Properties) von Objekten hat. Java kennt dieses nicht, hat dafür aber die Konvention von `get`- und `set`-Methoden. Die folgende C#-Klasse

```
public class Person
{
    private string _name;

    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (string.IsNullOrEmpty(value))
               throw new ArgumentException("Name must not be empty");
            _name = value;
        }
    }

    public int Age { get; set; }
}
```

sieht dann folgendermaßen in Java aus:

```
public class Person {
    private String _name;
    private int _age;

    public String getName() {
        return _name;
    }

    public void setName(String name) {
        if (name == null || "".equals(name))
            throw new IllegalArgumentException(
                "Name must not be empty");

        _name = name;
    }
```

```
        public int getAge() {
            return _age;
        }

        public void setAge(int age) {
            _age = age;
        }
    }
```

Wie zu erkennen ist, werden aus jeder Eigenschaft zwei Methoden (oder auch nur eine, wenn die Eigenschaft nur les- oder schreibbar ist). Automatisch implementierte Eigenschaften wie `Age` im C#-Beispiel gibt es in Java gar nicht. Hier muss das zugrundeliegende private Feld sowie der Zugriff darauf manuell implementiert werden.

Anhang C. Quelltextbeispiel aus der Umsetzung der Konzepte

Die meisten Klassen, die sich in der in Kapitel 6 beschriebenen prototypischen Umsetzung finden, sind lediglich Datenstrukturen ohne eigene Logik. Aus diesem Grunde ist der jeweilige Quelltext nur bedingt von Interesse. Stattdessen soll hier exemplarisch die Methode, die ein Modell auf Fehler und Modellierungsprobleme überprüft, dargestellt werden.

Fast alle Überprüfungen wurden mittels Language Integrated Query (LINQ) umgesetzt, welches eine SQL-ähnliche Anfragesprache in C# ist, die sich auf Objektmengen oder auch Datenbanken anwenden lässt. Im Grunde lässt sich damit Code schreiben, der Konzepte der funktionalen Programmierung nutzt. Dies ist im Wesentlichen eine Hilfe dabei, die Überprüfungen kurz und leicht lesbar zu halten. Effizienz war hier weniger das Ziel als Korrektheit.

```
/// <summary>
///     Helper class that defines a CheckModel() extension method on a
///     DynamicTextModel that checks for a range of questionable or
///     invalid conditions within the model.
/// </summary>
/// <remarks>
///     A model that is considered final and complete should not have
///     any of the problems checked for.
/// </remarks>
public static class Check
{
    public static IEnumerable<IWarning> CheckModel(this DynamicTextModel model)
    {
        var l = new List<IWarning>();

        // cache the document nodes
        var nodes = model.DocumentNode.GetNodes().ToList();

        /*************************************************************
         * Check for orphaned elementary criteria, i.e. elementary
         * criteria that are not referenced by any reader criterion.
         *************************************************************/
        l.AddRange(
            // take all elementary criteria
            model.ElementaryCriteria
                // remove those that are referenced
                .Except(model.ReaderCriteria.SelectMany(r => r.Criteria))
                .Select(e => new OrphanedElementaryCriterionWarning(e)));
```

```
/*****************************************************************
 * Check for unused elementary criteria, i.e. elementary criteria
 * that are never referenced by a condition in the text model
 * graph.
 *****************************************************************/
l.AddRange(
    // take all elementary criteria
    model.ElementaryCriteria
        // reduce them to their variable name
        // (as that's the only thing left in the node conditions)
        .Select(e => e.VariableName)
        // remove those that are referenced in the graph
        .Except(
            // get all nodes
            nodes
                // reduce them to the variables contained in the
                // conditions
                .SelectMany(n => n.Condition.GetVariables()),
            StringComparer.InvariantCultureIgnoreCase)
        .Select(e => new UnusedElementaryCriterionWarning(e)));

/*****************************************************************
 * Check for reader criteria that enclose elementary criteria not
 * in the model.
 *****************************************************************/
l.AddRange(
    from r in model.ReaderCriteria
    // filter out the elementary criteria defined in the model
    from e in r.Criteria.Except(model.ElementaryCriteria)
    select new UndefinedElementaryCriterionWarning(r, e));

/*****************************************************************
 * Check for groups that enclose reader criteria not in the model.
 *****************************************************************/
l.AddRange(
    from g in model.Groups
    // filter out the reader criteria defined in the model
    from r in g.Criteria.Except(model.ReaderCriteria)
    select new UndefinedReaderCriterionWarning(g, r));

/*****************************************************************
 * Check for conditions that use elementary criteria that do not
 * exist in the model.
 *****************************************************************/
l.AddRange(
    from n in nodes
    // filter out those variables that are defined in the model
    let vars = n.Condition.GetVariables().Except(
        model.ElementaryCriteria.Select(e => e.VariableName),
        StringComparer.InvariantCultureIgnoreCase)
    from v in vars
    select new UndefinedVariableWarning(n, v));

/*****************************************************************
 * Check for cycles in the graph.
 *****************************************************************/
l.AddRange(
    from n in nodes
    where n.GetNodes().Contains(n)
    select new GraphCycleWarning(n));
```

```
/*****************************************************************
 * Check for graph nodes with empty text fragments.
 *****************************************************************/
l.AddRange(
    from n in nodes
    where string.IsNullOrEmpty(n.Fragment)
    select new EmptyFragmentWarning(n));

/*****************************************************************
 * Check for inconsistent conditions on nodes.
 * For this we have to emulate the layout algorithm for each
 * configuration, essentially.
 *****************************************************************/
var configurations =
    GetPossibleReaderCriteriaConfigurations(
        new List<ReaderCriterion>(),
        model.Groups,
        model.ReaderCriteria.Except(
            model.Groups.SelectMany(g => g.Criteria)))
    .Select(rcc => new
    {
        ReaderCriteria = rcc,
        Configuration =
            DynamicTextModel.GetCriteriaConfiguration(rcc)
    });

foreach (var c in configurations)
{
    // start with the document node
    var node = model.DocumentNode;
    // set of visited nodes to prevent running into cycles
    var visitedNodes = new HashSet<Node>();

    while (node.Successors.Count > 0)
    {
        if (visitedNodes.Contains(node))
            break;

        if (node.Successors.Count == 1)
        {
            visitedNodes.Add(node);
            node = node.Successors[0];
        }
        else
        {
            var possibleSuccessors =
                node.Successors
                    .Where(s => s.Condition.GetValue(c.Configuration));

            if (possibleSuccessors.Count() == 0)
                l.Add(new NoPossibleSuccessorsWarning(
                    node, c.ReaderCriteria));
            else if (possibleSuccessors.Count() > 1)
            {
                l.Add(new MultiplePossibleSuccessorsWarning(
                    node, possibleSuccessors.ToList(),
                    c.ReaderCriteria));
                break;
            }
```

```
                else
                {
                    visitedNodes.Add(node);
                    node = possibleSuccessors.Single();
                }
            }
        }
    }

    return l;
}

/// <summary>
///     Enumerates all possible configurations of elementary criteria
///     that can occur given the groups and reader criteria of a
///     model.
/// </summary>
/// <param name="model">The text model.</param>
/// <returns>
///     An enumeration of all elementary criteria configurations
///     that are possible under the constraints of the model.
/// </returns>
public static IEnumerable<CriteriaConfiguration>
    GetPossibleConfigurations(DynamicTextModel model)
{
    return GetPossibleReaderCriteriaConfigurations(
            new List<ReaderCriterion>(),
            model.Groups,
            model.ReaderCriteria.Except(
                model.Groups.SelectMany(g => g.Criteria)))
        .Select(rcc => DynamicTextModel.GetCriteriaConfiguration(rcc));
}

/// <summary>
///     Enumerates all possible configurations of reader criteria
///     given a set of groups and ungrouped reader criteria.
/// </summary>
/// <param name="pickedCriteria">
///     The criteria picked so far. This is initially empty but
///     will be populated on subsequent recursive calls.
/// </param>
/// <param name="groups">
///     The groups from the model. Groups enforce that only exactly
///     one criterion per group can be chosen.
/// </param>
/// <param name="criteria">
///     The ungrouped reader criteria from the model. Ungrouped
///     criteria may either be chosen or not.
/// </param>
/// <returns>
///     An enumeration of all possible sets of chosen reader criteria
///     given the group and ungrouped criteria constraints.
/// </returns>
public static IEnumerable<ISet<ReaderCriterion>>
    GetPossibleReaderCriteriaConfigurations(
        IEnumerable<ReaderCriterion> pickedCriteria,
        IEnumerable<Group> groups,
        IEnumerable<ReaderCriterion> criteria)
```

```
{
    // Start enumerating all possible selections from groups.
    // Exactly one criterion must be chosen from each group.
    if (groups.Any())
    {
        // Choose group to process in this call
        var group = groups.First();
        // Filter collection of groups accordingly to omit it
        var newGroups = groups.Skip(1);

        foreach (var criterionInGroup in group.Criteria)
        {
            // Add the criterion to the picked collection
            var newPickedCriteria =
                pickedCriteria.Concat(
                    new List<ReaderCriterion> { criterionInGroup });

            // Gather results from the recursive calls
            var results = GetPossibleReaderCriteriaConfigurations(
                    newPickedCriteria, newGroups, criteria);

            // Let them bubble upwards
            foreach (var r in results) yield return r;
        }
    }
    // Continue with criteria if no more groups are to process
    else if (criteria.Any())
    {
        // Choose criterion to process in this call
        var criterion = criteria.First();
        // Filter collection accordingly
        var newCriteria = criteria.Skip(1);

        // Make one call with that criterion
        var results = GetPossibleReaderCriteriaConfigurations(
                pickedCriteria.Concat(
                    new List<ReaderCriterion> { criterion }),
                groups, newCriteria)
            // and one without it
            .Concat(GetPossibleReaderCriteriaConfigurations(
                pickedCriteria, groups, newCriteria));

        foreach (var r in results) yield return r;
    }
    // If there are neither groups nor criteria to pick from we can
    // return the currently picked criteria as one possible
    // configuration.
    else
        yield return new HashSet<ReaderCriterion>(pickedCriteria);
}

/// <summary>
///     Retrieves the nodes of a graph as a enumerable list.
/// </summary>
/// <param name="n">The node to start from.</param>
/// <returns>
///     A list of all nodes reachable from the given start node.
/// </returns>
private static IEnumerable<Node> GetNodes(this Node n)
{
    // set to avoid crawling duplicates in case of cycles
    ISet<Node> s = new HashSet<Node>();
```

```
            // list that will be handed out
            IList<Node> l = new List<Node>();
            // queue of nodes yet to crawl
            Queue<Node> q = new Queue<Node>(); q.Enqueue(n);

            while (q.Count > 0)
            {
                var node = q.Peek();
                foreach (var n2 in node.Successors)
                {
                    if (!s.Contains(n2))
                    {
                        q.Enqueue(n2);
                        l.Add(n2);
                        s.Add(n2);
                    }
                }
                q.Dequeue();
            }

            return l;
        }

        /// <summary>
        ///     Retrieves the names of all variables in a given expression
        ///     tree.
        /// </summary>
        /// <param name="c">The term to find all variables in.</param>
        /// <returns>
        ///     An enumerable list of all variables used in the expression.
        /// </returns>
        private static IEnumerable<string> GetVariables(this ITerm c)
        {
            if (c is Variable)
                yield return ((Variable)c).Criterion;
            else if (c is NotCondition)
                foreach (var v in GetVariables(((NotCondition)c).Term))
                    yield return v;
            else if (c is AndCondition)
                foreach (var v in GetVariables(((AndCondition)c).LeftTerm)
                        .Union(GetVariables(((AndCondition)c).RightTerm)))
                    yield return v;
            else if (c is OrCondition)
                foreach (var v in GetVariables(((OrCondition)c).LeftTerm)
                        .Union(GetVariables(((OrCondition)c).RightTerm)))
                    yield return v;
            else yield break;
        }
    }
```

Anhang D. Fallbeispiel aus Kapitel 7

In diesem Anhang findet sich der für das Fallbeispiel in Kapitel 7 modellierte Text in mit Kenntlichmachung von Einschüben und Alternativen. Als Grundlage hierfür wurde eine Einführung in LaTeX verwendet, die freundlicherweise von Andreas Dähn zur Verfügung gestellt wurde (22).

Es werden die schon in Kapitel 7 besprochenen elementaren Kriterien der Metaebene verwendet, mit folgenden Kürzeln:

Betriebssystem: Windows	*win*
Betriebssystem: Linux	*linux*
Betriebssystem: BSD	*bsd*
Betriebssystem: Mac OS X	*osx*
Betriebssystem: anderes	*anderesos*
LaTeX ist schon installiert	*installiert*
Leser hat nur Erfahrung mit Schriftformatierung in Word	*formatierung*
Leser hat Erfahrung mit Formatvorlagen in Word	*vorlagen*
Dokumentenklasse: book	*buch*
Dokumentenklasse: article	*artikel*
Dokumentenklasse: beamer	*folien*
BibTeX	*bibtex*
Abbildungen	*bilder*
Tabellen	*tabellen*
Spalten	*spalten*
gespiegelte Seitenanordnung	*seitenspiegel*
Formelsatz	*formeln*
Chemische Formeln	*chemie*
Quelltext-Listings	*listings*

Im Folgenden wird der Text mit Einschüben und Alternativen modelliert. Statt der bisher verwendeten grafischen Darstellung des entsprechenden Graphen wurde eine alternative Art der Kenntlichmachung gewählt.

Alternativen erscheinen grün hinterlegt, wobei am Rand jeweils auf der gleichen Ebene die Bedingungen abgetragen sind, die für das jeweilige Fragment gelten.	Alternative
Einschübe sind violett hinterlegt. Hier wird ebenfalls am Rand die jeweilige Bedingung kenntlich gemacht.	Einschub

Sowohl Einschübe als auch Alternativen können geschachtelt sein. Aus Platzgründen werden Alternativen nicht nebeneinander, sondern übereinander dargestellt. Weiterhin sind nicht alle Teile des Textes komplett ausformuliert bzw. nachträglich gekürzt worden, um diesen Anhang nicht zu lang werden zu lassen. Dieses Beispiel dient im Wesentlichen zur Illustration der Modellierung von Texten mit komplexeren Anforderungen. Abschnitte, die also in allen Varianten identisch sind, sind daher weniger interessant und gekürzt.

1 LaTeX – erster Kontakt

1.1 Was ist LaTeX?

LaTeX ist eine Sammlung von Makros für das Textsatzsystem TeX. Diese Sammlung wurde von Leslie Lamport im Jahr 1984 begonnen und LaTeX steht für „Lamports TeX“.

1.2 Was unterscheidet LaTeX von einer Textverarbeitung?

Textverarbeitungen gehen nach einem radikal anderen Konzept vor als LaTeX dies tut.

In LaTeX schreibt man nicht direkt vor, wie der Text auszusehen hat, sondern weist stattdessen die Semantik zu. Überschriften werden nicht groß und fett formatiert, sondern als Überschriften kenntlich gemacht. LaTeX macht dann den Rest automatisch.	formatierung
In LaTeX arbeitet man ähnlich wie mit Formatvorlagen in Word: Das konkrete Aussehen von Überschriften, etc. wird an anderer Stelle definiert (oder auf LaTeXs Standardeinstellungen belassen), so dass das gesamte Dokument einheitlich aussieht.	vorlagen
Textverarbeitungen machen es häufig einfach, die direkte Schriftformatierung zu ändern, vernachlässigen dabei aber eine semantische Einordnung. Benutzer formatieren damit häufig jede Überschrift einzeln größer und fett, obwohl sie auch direkt als Überschrift gekennzeichnet werden könnte. LaTeX erzwingt letztere Arbeitsweise durch semantisches Markup.	¬formatierung ∧ ¬vorlagen

1.3 Bezug von LaTeX, Installation

LaTeX kann man kostenlos herunterladen.

Für Windows gibt es die Distribution MiKTeX. [...]	win
Linux-Benutzer müssen sich an ihrer Distribution orientieren – die meisten Distributionen bieten ein Paket namens „tetex“, welches installiert werden sollte.	linux
Unter BSD kann man LaTeX aus den Ports selbst kompilieren oder als binäres Paket installieren.	bsd
Benutzer von Apples Mac OS X können LaTeX mittels „Fink“ installieren. Außerdem gibt es das Paket „MacTeX“, welches einen grafischen Installer bietet.	osx

Um zu überprüfen, ob die Installation von LaTeX erfolgreich war, gebe man in

einer Shell	¬win
einer Eingabeaufforderung	win

den Befehl `latex` ein. Wenn keine Fehlermeldung, sondern ein Text im Stile von `This is *TeX. Version ***` ausgegeben wird, so war die Installation erfolgreich.

¬installiert

2 Der erste LaTeX-Lauf

2.1 Dokument erstellen

Das Dokument wird zunächst als Quelltext erstellt – also ohne jede Formatierung. Hierzu kann ein beliebiger Editor

, beispielsweise Notepad oder Notepad2,	win
, beispielsweise KWrite, gedit oder vi,	linux ∨ bsd
, beispielsweise TextEdit,	osx
	anderesos

benutzt werden. Wichtig ist, dass eine Datei erzeugt wird, die selbst keine Formatierung sondern nur den eingegeben Text enthält. Diese wird mit einem Dateinamen wie `eins.tex` erzeugt.

Zunächst einmal benötigt LaTeX eine Information darüber, welche Art Dokument in der gegebenen Datei enthalten ist. Dies geschieht durch den Befehl `\documentclass`. Nach dieser Information kann der eigentliche Inhalt des Dokumentes beginnen. Im Code sieht das so aus:

`\documentclass{book}`	buch
`\documentclass{article}`	artikel
`\documentclass{beamer}`	folien

```
\begin{document}
```

`  \begin{frame}`	folien

```
    \frametitle{Toller Titel}
    Toller Text.
  \end{frame}
  Toller Text.
\end{document}
```

¬folien

2.2 Von der Eingabe zur Ausgabe

Um diese Datei nun in eine „ansehbare“ Form zu bringen, ist sie einem LaTeX-Lauf zu unterwerfen. Hierzu wird die Datei dem Interpreter übergeben, der sie dann weiterverarbeitet. In diesem Beispiel wäre der Befehl dazu `latex eins.tex`. [...] Der LaTeX-Lauf erzeugt eine Handvoll Dateien im aktuellen Verzeichnis. Eine Testausgabe:

```
ad001@ATP:~/latex$ ls eins.*
eins.aux      eins.dvi      eins.tex      eins.log      eins.toc
```

¬win

```
H:\LaTeX> dir eins.*
eins.aux
eins.dvi
eins.tex
eins.log
eins.toc
```

win

Zur Erklärung: In der Datei `eins.aux` befinden sich erst einmal nicht weiter relevante Hilfsdaten; in der Datei `eins.log` [...]. Nun bleibt noch `eins.dvi`. Dies ist das Ziel der Bemühungen gewesen. Die Datei kann man sich nun als Ergebnis ansehen:

```
xdvi eins.dvi
```

¬win

```
yap eins.dvi
```

win

Dies ist der generelle Ablauf, um eine LaTeX-Datei zur Anzeige zu bringen. Alternativ kann auch `pdflatex` anstelle von `latex` als Interpreter benutzt werden, in dem Falle wird gleich eine PDF-Datei erstellt.

3 Der Grundaufbau eines LaTeX-Dokumentes

Was im letzten Kapitel an einem Beispiel demonstriert wurde, soll hier nun theoretisch unterfüttert werden. Hierzu soll zunächst die Testdatei noch einmal in abstrahierter Form angesehen werden:

```
\documentclass[Optionen]{Klasse}
\usepackage[Optionen]{Paket}
\begin{document}
  Inhalt
\end{document}
```

3.1 Präambel

Die Zeilen 1 und 2 bilden den Teil, der als Präambel bezeichnet wird. [...] Die erste Zeile ist unabdingbar, hier wird LaTeX vorgegeben, um was für eine Art von Dokument es sich im Folgenden handelt. In der Folge wird das Grundlayout entsprechend für

ein Buch	buch
einen Artikel	artikel
Vortragsfolien	folien

gesetzt. Der Dokumentenklasse können optionale Argumente mitgegeben werden, wie beispielsweise die Papiergröße.
Der Einstellung des Dokumenttyps folgt der Import von benötigten Paketen, die mittels `\usepackage` eingebunden werden. [...] Pakete bieten Modularisierung und sind ein Grund für die Beliebtheit von LaTeX.

3.2 Inhalt

Auf die Präambel folgt der eigentliche Inhalt des Dokumentes (ab Zeile 3). Hier ist der Text, hier werden die aus den Paketen importierten Funktionalitäten genutzt.

4 LaTeX-Syntax

[...]

5 LaTeX in der Anwendung

Wenn man beginnt, sich mit LaTeX zu befassen, wird man relativ schnell auf eine Handvoll Probleme stoßen. Ein paar davon sollen im Folgenden abgehandelt werden.

5.1.1 Geschütztes Leerzeichen

[...]

5.1.2 Anführungszeichen

[...]

5.1.3 Umlaute und Kodierung

Möchte man Zeichen außerhalb des ASCII-Zeichensatzes verwenden, so muss man dies LaTeX erst mitteilen. Hierfür gibt es das Paket `inputenc`. Dem Paket muss lediglich die genutzte Zeichencodierung mitgegeben werden; ein Aufruf könnte also

`\usepackage[latin1]{inputenc}`	linux ∨ bsd ∨ anderesos

`\usepackage[ansinew]{inputenc}`	win
`\usepackage[macroman]{inputenc}`	osx

lauten. Alternativ, falls UTF-8 verwendet wird, auch `\usepackage[utf8x]{inputenc}`.

5.2 Gliederungsbefehle

5.2.1 Inhaltsverzeichnis

Ein Inhaltsverzeichnis kann eingefügt werden indem der Befehl `\tableofcontents` benutzt wird. Dies hat außerhalb der Gliederung zu geschehen.

Dies hat außerhalb der Gliederung zu geschehen. Dies muss innerhalb einer `frame`-Umgebung geschehen.	¬folien folien

5.2.2 Gliederung erstellen

Um eine Gliederung zu erstellen, benötigt man folgende Befehle:

`\chapter{`*Kapitel*`}`
`\subchapter{`*Unterkapitel*`}`
`\subsubchapter{`*Unter-Unterkapitel*`}`
	buch

`\section{`*Abschnitt*`}`
`\subsection{`*Unterabschnitt*`}`
`\subsubsection{`*Unter-Unterabschnitt*`}`
	artikel

`\begin{frame}{`*Folientitel*`} ... \end{frame}`
`\section{`*Abschnitt*`}`
`\subsection{`*Unterabschnitt*`}`
	folien

5.2.3 Titelseite

Die Titelseite soll als allem voranstehend einmal als Teil der Gliederung aufgefasst und hier behandelt werden. Die „Variablen“ für eine Titelseite werden in der Präambel festgelegt, die Titelseite wird danach innerhalb der `document`-Umgebung mithilfe des Tags `\maketitle` ausgelöst. Es folgt ein Beispiel aus dieser Datei:

```
\title{(Kurz-)Einführung in Textsatz mit \LaTeX{}}
\author{Andreas Dähn}
\begin{document}
    \maketitle
    ...
\end{document}
```

5.3 Formelsatz

Formelsatz ist eine der Kernkompetenzen von LaTeX, weshalb ich nicht umhinkommen werde, ihn hier zumindest anzureißen.

5.3.1 Mathe-Umgebungen

Mathe-Satz muss in einer Mathe-Umgebung vorgenommen werden. Es gibt zwei verschiedene Möglichkeiten, eine Formel in den Text einzubinden: Entweder als Formel im laufenden Text $a^2 = b^2 = c^2$ oder als abgesetzte Formel:

$$x_{1,2} = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}.$$

Zum Verständnis nun der ganze Absatz als Quelltext:

```
Es gibt zwei verschiedene Möglichkeiten, eine Formel in den Text einzubinden:
Entweder als Formel im laufenden Text $a^2=b^2=c^2$ oder als abgesetzte Formel:
\[x_{1,2} = -\frac p 2 \pm \sqrt{\frac {p^2} 4 - q}.\]
```

5.3.2 Grundlegende Konstruktionen

[…]

5.3.3 Griechische Buchstaben

[…]

5.3.4 Mathematische Sonderzeichen

[…]

5.3.5 Funktionsnamen

[…]

formeln

5.4 Vielgenutzte Umgebungen

Hier folgt nur ein kleiner Ausschnitt – es gibt eine Unmenge an Umgebungen, jede für ihren Spezialfall. Da dies dem Titel nach allerdings nur eine „Einführung“ werden soll, beschränke ich mich mal auf wenige, die dafür im täglichen Gebrauch von größerer Bedeutung sind.

5.4.1 `itemize`, oder: „Liste, unnumeriert“

Für eine Liste von Items ist die `itemize`-Umgebung die einfachste Variante. Zu beachten ist: Jeder einzelne Eintrag beginnt mit `\item` – und hat als Enddefinition nur den Beginn

des nächsten Items oder das Ende der Listenumgebung. Als Parameter kann optional ein anderes Symbol für den einzelnen Punkt gesetzt werden. LaTeX kommt mit Schachtelungen ohne Probleme zurecht, hierbei werden die Symbole gewechselt, die dem einzelnen Punkt der Aufzählung vorangestellt sind.

5.4.2 `description`, oder: „Liste, eigene Überschrift“

[...]

5.4.3 `enumerate`, oder: „Liste, numeriert“

[...]

5.4.4 `verbatim`, oder: „nur Text“

[...]

tabellen

5.4.5 `tabular`, oder: „Tabellen, simpel“

Die `tabular`-Umgebung erlaubt simple Tabellen. Diese Tabellenform ist allerdings noch nicht in der Lage, mit Seitenumbrüchen umzugehen. Ein Beispiel:

Überschrift		
a	b	c
d	e	f

Und die Quelle:

```
\begin{tabular}{|c|l r|}
  \hline
  \multicolumn{3}{|c|}{Überschrift}\\
  \hline
  a & b & c\\
  d & e & f\\
  \hline
\end{tabular}
```

[...]

listings

5.4.6 `lstlisting`, oder: Programmquelltext einbinden

Mithilfe der Umgebung `lstlisting` aus dem Paket `listings` ist es ohne Probleme möglich, Quelltext aus verschiedenen Programmiersprachen formatiert auszugeben.
Der Quelltext kann natürlich auch aus einer externen Datei eingebunden werden und muss nicht wie hier gezeigt im LaTeX-Dokument stehen.
Das Ergebnis kann dann wie folgt aussehen:

listings

```
1  #include <stdio.h>
2  int main (int argc, char * argv[])
3  {
4          printf("Ich_habe_%i_Argumente_bekommen!\n\n",argc);
5          return 0;
6  }
```

Der Quelltext dafür sieht wie folgt aus:

```
\lstset{language=c}
\begin{lstlisting}[numbers=left,frame=single]
#include <stdio.h>
int main (int argc, char * argv[])
{
       printf("Ich habe %i Argumente bekommen!\n\n",argc);
       return 0;
}
\end{lstlisting}
```

bibtex

5.5 Literaturverzeichnis, einfach

Jede größere naturwissenschaftliche Arbeit stützt sich zu nicht unerheblichen Anteilen auf Zitate aus Drittliteratur (Primär- und Sekundärquellen), die entsprechend gekennzeichnet werden müssen. Dazu ist in den meisten Distributionen von LaTeX das Programm BibTeX enthalten. Für größere Projekte sollte unbedingt eine separate BibTeX-Datenbank benutzt werden, für kleinere Projekte reicht die hier vorgestellte Technik durchaus auch.
Am Ende des Dokumentes (noch vor `\end{document}`) werden die Literaturangaben folgendermaßen eingegeben:

```
\begin{thebibliography}{99}
\bibitem[Q]{W} ERZ: Mein Feind.
\end{thebibliography}
```

[...]

5.6 Literaturverzeichnis mit BibTeX

[...]

5.7 Fußnoten

[...]

6 Komplexere Formatierungen

6.1 Kopf- und Fußzeile anpassen

[...]

6.2 Gespiegelte Seitenränder

[...]

seitenspiegel

An diesem Punkt wurde der Großteil der eingangs erwähnten elementaren Kriterien verwendet. Damit dieser Anhang nicht zu lang wird, soll dies als Beispiel genügen. Die restlichen Kriterien lassen sich an geeigneten Stellen problemlos in die Gliederung mit einbringen.

Weiterhin würden an einigen Stellen (die bisher ausgelassen wurden) noch kleine Alternativen entstehen, um auf Betriebssystem-spezifische Dinge oder auf bestimmtes Vorwissen einzugehen – gerade im einleitenden Abschnitt ist dies auch schon geschehen. Dies sind kleine Details, die es allerdings dem Leser sehr erleichtern, sich in den Text hineinzufinden. Windows-Benutzer erhalten keine Linux-spezifischen Informationen, weder in Klammern irgendwo, noch als Fußnote – es gibt nichts, was überlesen werden muss.

Im größeren Maßstab werden dann einige Themen nur dann behandelt, wenn sie für die Zielstellung des Lesers wichtig und notwendig sind – dass ein Informatiker chemische Strukturformeln erstellen muss, ist eher unwahrscheinlich.

Anhang E. Glossar

Um Missverständnisse zu vermeiden werden in dieser Arbeit verwendete Begriffe hier einmal kurz erläutert. Die gleiche Definition findet sich auch bei der jeweils ersten Verwendung eines Begriffes im Text.

Inhalt bezeichnet relativ abstrakt eine Menge an Informationen, die vermittelt werden soll. Dies kann in Form eines *Textes* geschehen, ist aber nicht darauf beschränkt. Im Rahmen dieser Arbeit bezeichnen Inhalte jedoch grundsätzlich Inhalte in Textform.

Als **Text** wird jede Art von vorwiegend textueller Information bezeichnet. Dies kann auch andere Medien beinhalten, beispielsweise Bilder, die im Text eingebettet sind. Generell ist ein Text etwas, was dazu gedacht ist, gelesen zu werden und Informationen und Wissen zu vermitteln, üblicherweise aber in linearer Form.

Ein **Textfragment** ist ein zusammenhängendes Teilstück eines *Textes*. Dies kann ein Kapitel oder ein Abschnitt sein, ein Absatz oder gar nur ein einzelner Satz oder Satzteil. Diese werden in der im Rahmen dieser Arbeit vorgestellten Modellierung und Implementierung genutzt, um *Textvarianten* aufzubauen. Fragmente sind dann vorwiegend Teile, die in mehreren Varianten verwendet werden.

Eine **Textvariante** ist ein *Text*, der das gleiche Thema wie ein anderer Text behandelt, aber andere Voraussetzungen bzw. Annahmen hat. Beispielsweise kann ein Kinderbuch grob das gleiche Thema behandeln wie ein Schulbuch oder ein Artikel in einem Lexikon. In diesen Fällen sind jedoch die Voraussetzungen, die an den Leser gestellt werden, deutlich andere.

Ein **Lesepfad** bezeichnet eine Sequenz von *Fragmenten* eines *Textes*, die der Reihenfolge nach gelesen werden. In Hypertexten, die mehrere Verzweigungsmöglichkeiten durch Hyperlinks haben, ergibt sich ein solcher Lesepfad durch die jeweils gewählten Verzweigungen.

Voraussetzungen sind die Menge an *Kriterien*, die für eine bestimmte *Textvariante* gelten.

Kriterien stellen einzelne Anforderungen an eine *Textvariante* dar. Diese unterteilen sich in Kriterien, die der Leser auswählen kann, (**Leserkriterien**) sowie **elementare Kriterien**, die grundlegende Anforderungen beschreiben und zur Textgestaltung durch den Autor vorgesehen sind. Dies wird insbesondere in Kapitel 4 betrachtet.

Kohärenz ist in einem *Text* das Maß, wie gut aufeinanderfolgende Argumente ineinandergreifen und sich überlappen. Hohe Kohärenz deutet auf starke Überlappung hin und erleichtert dem Leser so zumindest theoretisch das Verständnis des Textes. Wenig Kohärenz hingegen ist ein Zeichen dafür, dass sich Argumente im Text nicht ausreichend überlappen und damit das Verständnis erschweren.

Anhang F. Verwendete Teile der Z-Notation

Die formale Spezifikation in Abschnitt 4.3.4 verwendet die Z-Notation, um das Modell und sein Verhalten zu beschreiben. In diesem Abschnitt werden kurz die verwendeten Syntaxelemente aufgelistet sowie Verweise zu ihren jeweiligen Beschreibungen im *Z Reference Manual* (ZRM) (20) oder *Using Z* (UZ) (21) gegeben. Letzteres gibt häufig eine bessere Erklärung, während ersteres schon dem Namen nach eher Referenzcharakter hat. Beide Bücher sind online frei erhältlich[18].

Aufgeführt werden hier jedoch nur Teile von Z, die nicht schon identisch in normaler mathematischer Notation verwendet werden.

Auswahl einer Schemakomponente

Variable.Komponente

ZRM S. 61
UZ S. 153

Axiomatische Definition

| *Deklaration*
|---
| *Prädikate*
| *Deklaration*

ZRM S. 48
UZ S. 77

Bedingte Ausdrücke

if *Prädikat* **then** *Ausdruck* **else** *Ausdruck*

ZRM S. 64
UZ S. —

Deklaration

Variable : *Typ*

ZRM S. 51
UZ S. 73

Funktionen, totale Funktion

$X \longrightarrow Y$

ZRM S. 105
UZ S. 100

[18] J. M. Spivey: *The Z Notation: A Reference Manual*. http://spivey.oriel.ox.ac.uk/mike/zrm/
J. Woodcock, J. Davies: *Using Z: Specification, Refinement, and Proof*. http://www.usingz.com/

Funktionen, partielle Funktion

$X \pfun Y$

ZRM S. 105
UZ S. 99

Konkrete Beschreibung (*definite description*, μ)

(μ *Deklaration* | *Prädikate* • *Ausdruck*)

ZRM S. 58
UZ S. 52

Lokale Definition

(**let** *Definition* • *Ausdruck*)

ZRM S. 59
UZ S. —

Mengen, Menge eines Typs

$\mathbb{P}$ *Typ*

ZRM S. 25, 56
UZ S. 65

Mengen, Endliche Menge eines Typs

$\mathbb{F}$ *Typ*

ZRM S. 111
UZ S. 112

Mengen, Anzahl der Elemente einer endlichen Menge

Menge

ZRM S. 111
UZ S. 113

Mengen, Mengenbildung (*set comprehension*)

{ *Deklaration* | *Prädikate* • *Ausdruck* }

ZRM S. 52, 57
UZ S. 61

Quantoren, Allquantor

$\forall$ *Deklaration* | *Prädikate* • *Prädikat*

ZRM S. 70
UZ S. 29

Quantoren, Existenz

$\exists$ *Deklaration* | *Prädikate* • *Prädikate*

ZRM S. 70
UZ S. 28

Quantoren, Existenz genau eines Elements

$\exists_1$ *Deklaration* | *Prädikate* • *Prädikat*

ZRM S. 70
UZ S. 51

Relationen

$X \rightleftarrow Y$

ZRM S. 95
UZ S. 83

Relationen, Transitive Hülle

$Relation^+$

ZRM S. 103
UZ S. 95

Relationen, Definitionsbereich

dom *Relation*

ZRM S. 96
UZ S. 85

Relationen, Wertebereich

ran *Relation*

ZRM S. 96
UZ S. 85

Schemata

Schemabezeichnung

Deklaration
Prädikate

Schemabezeichnung

Deklaration

ZRM S. 28
UZ S. 147

Sequenzen

seq *Typ*

ZRM S. 115
UZ S. 115

Sequenzen, Literale

$\langle a, b, \ldots \rangle$

ZRM S. 66
UZ S. 115

Sequenzen, Verkettung

$a \frown b$

ZRM S. 116
UZ S. 115

Typen, Basistypen

[*Typ*]

ZRM S. 25
UZ S. 70

Typen, Freie Typen

$Typ ::= Name \mid Konstruktor \langle\langle Typ \rangle\rangle \mid \ldots$

ZRM S. 82
UZ S. 133

Typen, Abkürzungsdefinition

$Typ == Typ_2$

ZRM S. 50
UZ S. 74

Literaturverzeichnis

1. **Biermann, Kai.** Die Diktatur der Relevanz. *Zeit Online.* [Online] 23. Oktober 2009. http://www.zeit.de/digital/internet/2009-10/wikipedia-streit-fefe/komplettansicht.

2. *Complex Information Processing: A File Structure for The Complex, The Changing and the Indeterminate.* **Nelson, Ted H.** Cleveland, Ohio, United States : ACM, 1965. Proceedings of the 1965 20th national conference. S. 84–100. DOI: http://doi.acm.org/10.1145/800197.806036.

3. **Foltz, Peter W.** Comprehension, Coherence and Strategies in Hypertext and Linear Text. [Hrsg.] Jean-Francois Rouet, et al., et al. *Hypertext and Cognition.* 1. Edition. Hillsdale : Lawrence Erlbaum Associates, 1996, S. 109–136.

4. World Wide Web Consortium (W3C). [Online] 2011. http://www.w3.org/.

5. *As We May Think.* **Bush, Vannevar.** July 1945, The Atlantic.

6. **Dix, Alan, et al., et al.** *Human-Computer Interaction.* 1993.

7. **MacKay, David J. C.** *Information Theory, Inference, and Learning Algorithms.* Cambridge : Cambridge University Press, 2005.

8. **Brinker, Klaus.** *Linguistische Textanalyse: Eine Einführung in Grundbegriffe und Methoden.* 7. Auflage. Berlin : Erich Schmidt Verlag, 2010. 978 3 505 12206 6.

9. **Cap, Clemens H.** *Content Neutrality for Wiki Systems.* 2011. unveröffentlicht.

10. *Software Tutors Offer Help and Customized Hints.* **Hafner, Katie.** 16. September 2004, The New York Times.

11. **Freedman, Reva.** What is an Intelligent Tutoring System? *Intelligence.* 2000, Bd. 11, 3, S. 15–16.

12. **Conklin, Jeff und Begeman, Michael L.** gIBIS: A Tool for all Reasons. *Journal of the American Society for Information Science and Technology.* 1989, Bd. 3, 40, S. 200–213.

13. JANUS*: Integrated Hypertext with a Knowledge-based Design Environment.* **Fischer, Gerhard, McCall, Raymond und Morch, Anders.** Pittsburgh, Pennsylvania, United States : ACM, 1989. Proceedings of the second annual ACM conference on Hypertext. S. 105–117. http://doi.acm.org/10.1145/74224.74233. 0-89791-339-6.

14. **Kunz, Werner and Rittel, Horst W. J.** *Issues as Elements of Information Systems.* 1970. Working Paper No. 131.

15. **Downes, Stephen.** Learning Objects: Resources for distance education worldwide. *International Review of Research in Open and Distance Learning.* Juli 2001, Bd. 2, 1.

16. *What is DocBook?* [Online] http://docbook.org/whatis.

17. **Martínez-Ortiz, Iván, et al., et al.** Using DocBook and XML Technologies to Create Adaptive Learning Content in Technical Domains. *International Journal of Computer Science & Applications.* Juni 2006, Bd. 3, 2.

18. **Wikipedia.** Seite „Turingmaschine". *Wikipedia, Die freie Enzyklopädie.* [Online] 1. Dezember 2010. http://de.wikipedia.org/w/index.php?title=Turingmaschine&oldid=82143401.

19. **ISO/IEC 13568:2002(E).** Information technology – Z formal specification notation – Syntax, type system and semantics. 1. Juli 2002.

20. **Spivey, J. M.** *The Z Notation: A Reference Manual.* Second. Oxford : Prentice Hall, 1998. ISBN 978-0139785290.

21. **Woodcock, Jim und Davies, Jim.** *Using Z: Specification, Refinement, and Proof.* s.l. : Prentice Hall, 1996. ISBN 0-13-948472-8.

22. **Dähn, Andreas.** *Einführung in Textsatz mit LaTeX.* 2006.

23. **Creative Commons.** *Creative Commons.* [Online] http://creativecommons.org/.

24. —. About the licenses. *Creative Commons.* [Online] http://creativecommons.org/licenses/.

25. **Microsoft.** *Extension Methods (C# Programming Guide).* [Online] Oktober 2010. http://msdn.microsoft.com/en-us/library/bb383977.aspx.

Sämtliche URLs in Quellen wurden am 21. September 2011 zuletzt besucht und waren zu diesem Datum erreichbar.

Abbildungsverzeichnis

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Stellen meiner Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Dasselbe gilt sinngemäß für Tabellen und Abbildungen. Diese Arbeit hat in dieser oder einer ähnlichen Form noch nicht im Rahmen einer anderen Prüfung vorgelegen.

_____

Johannes Rössel

Rostock, 13. Oktober 2011

Thesen

1. Es gibt zwei große Arten von Text – linearen Text und Hypertext. Lineare Texte werden geradlinig von Anfang bis Ende gelesen. Hypertext stellt jedoch ein Netz untereinander verknüpfter Abschnitte dar, in der die Lesefolge nicht vorher festgelegt ist.
2. Lineare Texte eignen sich insbesondere für strukturierte und aufeinander aufbauende Inhalte. Hypertext ist besonders dann von Vorteil, wenn die Inhalte nichtlinear verknüpft sind und beliebige Sprünge zwischen verknüpften Inhalten möglich sein sollen.
3. Da Hypertext nahezu beliebige Sprünge zwischen Abschnitten des Dokumentes erlaubt, kann der Autor nur wenige Annahmen darüber treffen, was der Leser bereits gelesen hat. Das Verständnis des Textes kann durch inhaltliche Brüche leiden.
4. Dynamische Texte stellen einen Mittelweg zwischen linearen und Hypertexten dar. Sie ermöglichen die Anpassung eines Textes an verschiedene Zielgruppen mit jeweils unterschiedlichen Vorkenntnissen durch Kriterien die vorher durch den Leser ausgewählt werden.
5. Dynamische Texte werden auf zwei Ebenen modelliert, von denen eine die Anforderungen des Textes an den Leser beschreibt und die andere der Modellierung der verschiedenen Textvarianten für unterschiedliche Anforderungen dient. Dies ermöglicht einem Autor feine Kontrolle über die anzusprechenden Zielgruppen.
6. In der die Anforderungen beschreibenden Metaebene werden Kriterien unterteilt in *Leserkriterien*, die vom Leser ausgewählt werden können und dem Autor zur Modellierung von Zielgruppen dienen, sowie *elementare Kriterien*, die durch Leserkriterien gesetzt werden und elementare Anforderungen modellieren, die für die Modellierung des Textes verwendet werden. Leserkriterien können gruppiert werden, um sich gegenseitig ausschließende Kriterien zu modellieren.
7. Die Ebene auf der der Text modelliert wird, wird durch einen gerichteten Graph repräsentiert, der für die Modellierung unter anderem Alternativen und Einschübe von Textfragmenten unterstützt. Dies ermöglicht eine einfache und flexible Definition der Textvarianten für die verschiedenen Anforderungen.
8. Die vorgestellten Konzepte wurden prototypisch umgesetzt und ein komplexeres Fallbeispiel unter Anwendung der Konzepte erstellt. Dies demonstriert praktische Anwendbarkeit der Konzepte.