dodl2html

Ein Generator zum Erzeugen von graphspezifizierten Hyperdokumenten

Diplomarbeit

Wintersemester 1999/2000

Autor
Jörg Pleumann

Gutachter
Prof. Dr. E.-E. Doberkat
Dipl.-Inform. A. Fronk

Universität Dortmund
Lehrstuhl Informatik X
44221 Dortmund

Die Diplomarbeit ist im Kontext der *Document Description Language (DoDL)* angesiedelt, einer objektorientierten Spezifikationssprache für Hyperdokumente, die am Lehrstuhl für Software-Technologie der Universität Dortmund entwickelt wurde. Die Sprache *DoDL* propagiert eine strikte Trennung von Inhalt, Verknüpfungsstruktur und Laufzeitverhalten eines Hyperdokumentes, wobei sich die Diplomarbeit schwerpunktmäßig im Bereich der Verknüpfungsstruktur bewegt. Diese liegt in einer *DoDL*-Spezifikation als Konstruktionsvorschrift für einen attributierten Graphen vor, in dem die Knoten den Ankern und die Kanten den Verweisen eines Hyperdokumentes entsprechen. Die Spezifikation wird von einem Compiler in ein ausführbares Programm übersetzt, das – wenn es gestartet wird – den beschriebenen Graphen tatsächlich erzeugt und in eine Datei schreibt.

Hauptgegenstand der Diplomarbeit ist der Entwurf und die Implementierung eines Generatorwerkzeugs, das aus diesem Strukturgraphen, seiner Attributierung und dem eigentlichen Inhalt des Hyperdokumentes eine Menge von verknüpften Seiten erzeugt, die in *HyperText Markup Language (HTML)* abgefaßt sind. Dazu liest das Werkzeug den Graphen ein, traversiert ihn geeignet und erzeugt währenddessen die Ausgabedateien. Die Implementierung bemüht sich, geeignete Automatismen für spezielle Probleme des Generierungsprozesses zu finden, so etwa ein ansprechendes Minimal-Layout des Hyperdokumentes oder eine geeignete Aufteilung des gesamten Inhalts auf einzelne HTML-Seiten. Die Architektur ist zudem durch die durchgehende Verwendung objektorientierter Techniken so flexibel gehalten, daß der Generator mit geringem Aufwand an weitere Zielsysteme angepaßt werden kann.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Die vorliegende Diplomarbeit wurde im Wintersemester 1999/2000 am Lehrstuhl für Software-Technologie der Universität Dortmund durchgeführt. Sie ist im Kontext der *Document Description Language (DoDL)* angesiedelt, einer objektorientierten Spezifikationssprache für Hyperdokumente, die am gleichen Lehrstuhl entwickelt wurde und die in [Dobe95, Dobe96a, Dobe96b] beschrieben ist.

Der Titel `dodl2html` steht – der üblichen Nomenklatur von Konvertierungswerkzeugen unter Unix folgend – für ein Programm, das die Brücke zwischen der Welt von *DoDL* und der *Hypertext Markup Language (HTML)* [W3C99a] schlägt. Er sollte also „*DoDL to HTML*“ ausgesprochen werden. Die weiteren Details des Titels sollten sich dem Leser im weiteren Verlauf der Arbeit erschließen, die wie folgt gegliedert ist:

- Kapitel 2 erläutert das Thema der Arbeit, also insbesondere die Aufgabenstellung und den vom Autor gewählten Lösungsansatz.
- Aus dem Lösungsansatz ergeben sich drei Teilaufgaben. Die Kapitel 3 bis 5 beschreiben diese Teilaufgaben und die entsprechenden Teillösungen im Detail.
- Kapitel 6 zeigt einige Testläufe des Systems mit größeren Beispiel-Hyperdokumenten und bewertet die erreichten Ergebnisse.
- Kapitel 7 gibt einen Ausblick auf weitere Möglichkeiten und alternative Ansätze.

Im Anschluß an das letzte Kapitel folgt eine ausführliche Liste der verwendeten Literatur. Dabei handelt es sich sowohl um Literatur im traditionellen Sinne als auch um Literatur, die nur online verfügbar ist. Der Literaturliste folgt ein kurzer Anhang. Dort finden sich Informationen, die entweder nur am Rande von Interesse sind oder den allgemeinen Lesefluß gestört hätten und deshalb an das Ende des Textes verbannt wurden.

2. Thema der Arbeit

Dieses Kapitel erläutert das Thema der vorliegenden Arbeit, also insbesondere die eigentliche Aufgabenstellung und den vom Autor gewählten Lösungsansatz. Damit Aufgabe und Lösung jedoch nicht völlig vom Himmel fallen, soll zunächst das allgemeine Umfeld der Diplomarbeit etwas beleuchtet werden. Dies umfaßt sowohl eine Vorstellung der verwendeten Begriffe aus Bereich Hypermedia als auch einen kurzen Abriß über den Ansatz von *DoDL* und die Konzepte, die diesem Ansatz zugrunde liegen.

2.1. Hypertext, Hypermedia und Hyperdokumente

Einer der fundamentalen Begriffe, auf denen diese Diplomarbeit aufbaut, ist der des *Hyperdokumentes*. Eine naive Definition des Begriffes, die sich in dieser Form sicherlich in keinem Lehrbuch wiederfinden wird, könnte etwa wie folgt aussehen:

> Ein Hyperdokument ist ein Dokument, das nicht-linear gelesen werden kann.

Die Möglichkeit des nicht-linearen Lesens allein qualifiziert ein Dokument jedoch nicht wirklich zu einem Hyperdokument, denn es soll ja Leute geben, die das Ende eines Kriminalromanes zuerst lesen, um vorab herauszufinden, wer der Mörder ist. Niemand würde jedoch deshalb auf die Idee kommen, Kriminalromane generell als Hyperdokumente zu bezeichnen. Tatsächlich ist es die *Unterstützung* des nicht-linearen Lesens, die ein einfaches Dokument zu einem Hyperdokument werden läßt. Diese Unterstützung besteht in einem Netz von *Verweisen*, mit denen Verbindungen zwischen *Positionen* innerhalb des Dokumentes geschaffen werden, die aus Sicht des Autors „verwandt" sind. Aus diesen Verweisen ergeben sich für den Leser viele verschiedene mögliche Wege durch das Dokument, von denen er einen (oder mehrere) anhand seines Vorwissens, seiner Interessen oder beliebiger anderer Kriterien wählen kann.

Die Idee des Hyperdokumentes ist nicht neu. Legt man den Begriff sehr großzügig aus, dann könnte man sogar einen noch in Fraktur abgefaßten, mehrbändigen Brockhaus aus dem Jahr 1911 als Hyperdokument bezeichnen, wie der folgende Ausschnitt aus [Broc11] zeigt:

> Lexikon (grch.), Wörterbuch, alphabetisch geordnetes Verzeichnis eines Sprachschatzes oder eines wissenschaftlichen Stoffes. (...) Siehe auch Enzyklopädie und Konversationslexikon.

Das manuelle Blättern zur entsprechenden Stelle fällt zwar etwas aufwendiger aus als der heute übliche Mausklick auf dem Computer, aber dennoch kann man feststellen, daß die Querverweise zu den Begriffen *Enzyklopädie* und *Konversationslexikon* das nichtlineare Lesen prinzipiell unterstützen. In diesem Sinne wäre das alte Buch bereits ein Hyperdokument. Allgemein gilt jedoch der Artikel von Bush aus dem Jahr 1945 als Geburtsstunde der Idee des Hyperdokumentes, das er auf den Namen *Memex* [Bush45] taufte. In den sechziger Jahren begann Nelson seine Arbeit an einem System namens *Xanadu* [Wolf98]. Aus dieser Zeit stammt auch der Begriff *Hypertext*, aus dem sich dann später die Begriffe *Hypermedia* (für die multimediale Variante) und *Hyperdokument* (für die konkreten Vertreter dieser Spezies) ableiteten.

2.2. Einige Hypermedia-Systeme

Obwohl sie schon lange ein Forschungsgegenstand auch großer Unternehmen wie Xerox oder IBM waren, führten Hyperdokumente bis zum Ende der achziger Jahre eher ein Schattendasein. Einer breiteren Öffentlichkeit zugänglich wurden sie erst zu Beginn der neunziger Jahre in Form der hypermedialen Hilfesysteme von Apple MacOS, Microsoft Windows, IBM OS/2 und dem GNU-System der Free Software Foundation. Das größte und zweifellos bekannteste Beispiel für ein Hyperdokument ist jedoch das seit 1992 unaufhörlich wachsende World Wide Web (WWW). An seinem Beispiel treten auch die Probleme, mit denen Autoren von Hyperdokumenten konfrontiert werden, am deutlichsten zutage:

- Das WWW ist unüberschaubar groß.
- Es ist ständigen Änderungen unterworfen.
- Eine unbekannte Anzahl von Autoren ist mit seiner „Pflege“ beschäftigt.

Die technische Infrastruktur des WWW, deren wesentliches Element HTML ist, steht in keinem Verhältnis zu diesen drei Punkten. HTML ist eine seitenorientierte Beschreibungssprache, die alle wesentlichen Aspekte eines Hyperdokumentes in einer einzigen Datei vermengt. Entsprechend kompliziert ist die Erstellung und Wartung von HTML-basierten Hyperdokumenten, sobald diese den Umfang einer einfachen Homepage überschreiten.

Speziell die Konsistenzwahrung der Verweise ist ein großes Problem, wie man Tag für Tag im WWW bei dem Versuch feststellen kann, Verweisen auf Seiten zu folgen, die entweder nicht mehr existieren oder nie existiert haben. An diesem Umstand ändern auch die zahlreich verfügbaren visuellen Autorensysteme nicht viel, denn sie unterstützen den Autor im wesentlichen beim Layout seines Hyperdokumentes, nicht so sehr bei dessen Organisation.

Ausnahmen bilden hier Systeme wie *HyperWave* [Maur96] oder *HyperForm* [WiLe97], die ihren Inhalt in einer zentralen Datenbank ablegen und dementsprechend in der Lage sind, die Konsistenz der Verweise (zumindest innerhalb des Systems) zu wahren. Da sie zudem die Verwaltung von Benutzern und Zugriffsrechten ermöglichen, lassen sich mit ihnen auch sehr große Hyperdokumente in Kooperation von mehreren Autoren erzeugen. Andere Systeme wie *HyTime* [Newc91] und *Trellis* [StFu89] berücksichtigen neben Inhalt und Verknüpfungsstruktur auch den Gedanken, daß ein Hyperdokument ein Verhalten zur „Laufzeit“, also zur Zeit seiner Betrachtung besitzt. Im Fall von *Trellis* wird dies über Petri-Netze realisiert, die den allgemein üblichen Graph-Formalismus zur Beschreibung der Verknüpfungsstruktur um die benötigte dynamische Komponente erweitern.

Mit dem Anspruch, die Ideen, die den verschiedenen bis zu diesem Zeitpunkt bekannten Hypermedia-Systemen zugrunde lagen, auf eine gemeinsame Basis zu stellen, entstand 1990 das Dexter-Referenzmodell für Hypermedia-Systeme [HaSc94]. An diesem Modell, das einen dreischichtigen Aufbau eines Hypermedia-Systems vorsieht und verschiedene grundlegende Begriffe definiert, orientieren sich auch heute noch viele Arbeiten im Hypermedia-Kontext.

2.3. Die *Document Description Language*

Der Ansatz von *DoDL* basiert auf der Feststellung, daß die Probleme beim Entwurf eines Hyperdokumentes durchaus mit denen zu vergleichen sind, die beim Entwurf eines komplexen Software-Systems auftreten. Die drei in Abschnitt 2.2 genannten Punkte zeigen dies deutlich. Ausgehend von dieser Feststellung propagiert *DoDL* die Übertragung bewährter softwaretechnischer Vorgehensweisen auf den Bereich Hypermedia. Dazu gehört insbesondere das Prinzip, das als *separation of concerns* bekannt ist. Aus ihm folgt, daß Inhalt und Struktur – offensichtlich verschiedenartige Bestandteile eines Hyperdokumentes – eine getrennte Betrachtung und Beschreibung erfordern.

Abbildung 2.1 zeigt ein einfaches Hyperdokument. Der *Inhalt* dieses Hyperdokumentes setzt sich aus sogenannten *Medienobjekten* zusammen. Dabei kann es sich um simple Texte oder Grafiken, aber auch um komplexere multimediale Daten (etwa Musikstücke oder Videos) handeln. Die Medienobjekte werden in einer Datenbank abgelegt.

Mit *Struktur* ist die aus Positionen und Verweisen bestehende Verknüpfungsstruktur gemeint, die zwischen den Medienobjekten besteht. Wie der vorangehende Abschnitt gezeigt hat, ist es gerade diese Struktur, die ein einfaches Dokument zu einem Hyperdokument aufwertet, also zu einem zusammengesetzten Dokument, das den Leser beim nicht-linearen Durchlaufen unterstützt.

Zusätzlich zu Inhalt und Struktur berücksichtigt *DoDL* auch das *Verhalten* des Hyperdokumentes zur Laufzeit. Dieses Verhalten ist im Falle von *DoDL* ein aus Attributen und Regeln bestehendes System, das festlegt, wie sich das Hyperdokument bestimmten Benutzergruppen darstellt. Es kann im einfachsten Fall dazu verwendet werden, Benut-

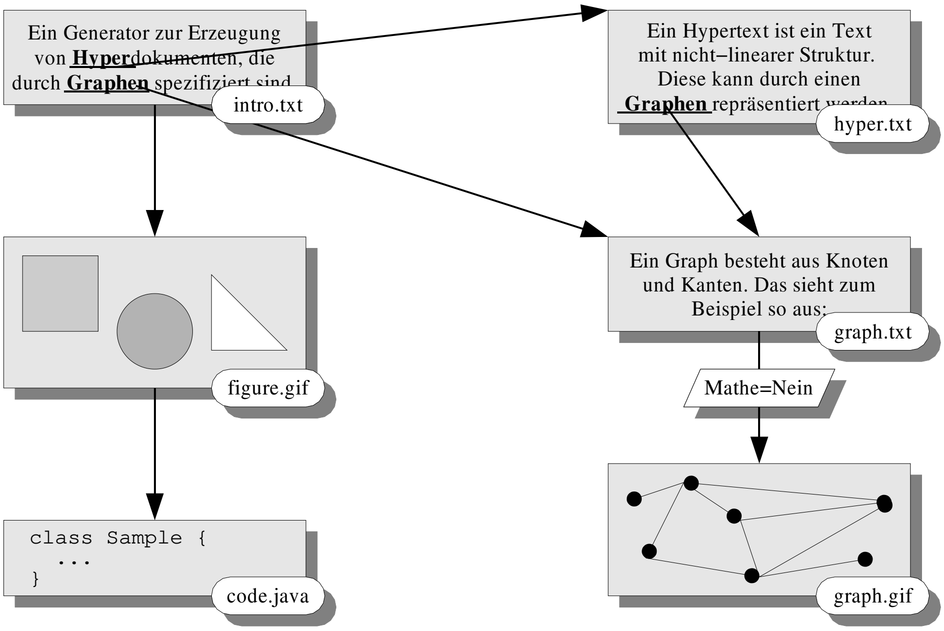

Abbildung 2.1.: Ein einfaches Hyperdokument

zern in Abhängigkeit von Merkmalen spezielle Sichten auf das gesamte Hyperdokument zu liefern.

In Abbildung 2.1 ist einer der Verweise mit einem Attribut annotiert, das den Namen *Mathe* und dem Wert *Nein* besitzt. Dieses Attribut könnte zur Laufzeit so interpretiert werden, daß der Verweis (und damit auch die Grafik, zu der er führt) nur für einen Leser angezeigt wird, der mit Mathematik nicht vertraut ist. Das Attribut würde also dazu dienen, unter allen vorhandenen Verweisen die für einen Benutzer sichtbaren zu selektieren. Tatsächlich gehen die Möglichkeiten – ein geeignetes Laufzeitsystem vorausgesetzt – jedoch weit über diesen simplen Selektionsmechanismus hinaus: Wenn das Laufzeitsystem zum Beispiel in der Lage ist, die Merkmale des Benutzers während des Betrachtens zu ändern, dann könnte anhand der bisher besuchten Seiten auf die Interessen oder das Wissen des Benutzers geschlossen werden. Weitere Seiten könnten den geänderten Merkmalen entsprechend automatisch aufbereitet werden. Damit ließen sich Systeme im Bereich des *Computer Based Training (CBT)* realisieren, wie etwa das *Kolibri*-System [Bahn98], das am Lehrstuhl für Automaten- und Schaltwerktheorie der Universität Dortmund entwickelt wurde.

Inhalt und Struktur bilden den statischen Anteil eines Hyperdokumentes, während das Verhalten offensichtlich eher dynamischen Charakter besitzt. Abbildung 2.2, die eine Erweiterung von [Fron99] darstellt, verdeutlicht diese Gedanken anhand eines Schichtenmodells. Das Modell zeigt, wie sich das Hyperdokument in einen statischen und einen dynamischen Anteil gliedert, welche sich wiederum zu Inhalt, Struktur und Verhalten

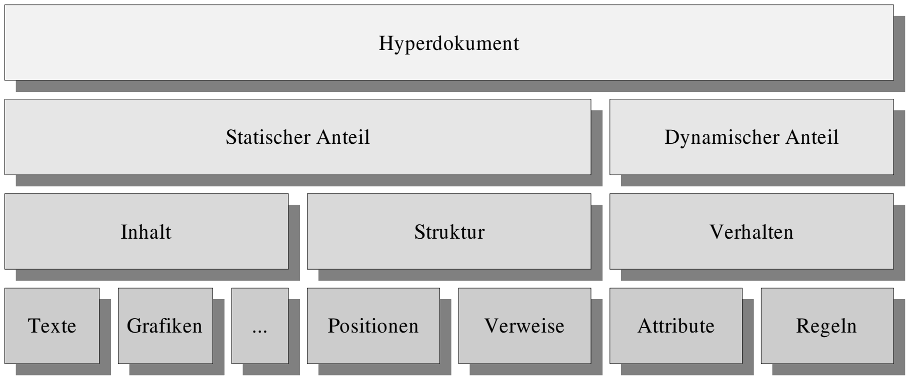

Abbildung 2.2.: Schichtenmodell eines Hyperdokumentes

verfeinern lassen. Auf der untersten Ebene befinden sich die Elemente oder Hilfsmittel, die dem jeweiligen Anteil eines Hyperdokumentes zugrunde liegen.

Dem Autor ist durchaus bewußt, daß zwischen den einzelnen Elementen komplexere Beziehungen bestehen. So beziehen sich zum Beispiel die Positionen auf die Medienobjekte, in denen sie verankert sind. Ihre konkrete Realisierung ist sogar – wie der weitere Verlauf dieser Arbeit zeigen wird – vom Typ des jeweiligen Medienobjektes abhängig. Die Attribute, um ein weiteres Beispiel zu nennen, dienen zum Annotieren der Struktur. Es scheint so, als bestünden zwischen den Elementen der zweituntersten Schicht (und damit natürlich auch implizit der anderen Schichten) weitere Beziehungen derart, daß weiter rechts stehende Elemente auf weiter links stehenden aufbauen. Die Grafik hat jedoch nicht den Anspruch, all diese Beziehungen vollständig zu beschreiben. Es soll lediglich eine einfache Hierarchie der Bestandteile eines Hyperdokumentes dargestellt werden, um grundsätzliche Begrifflichkeiten zu klären, die im weiteren Verlauf der Arbeit Verwendung finden.

2.4. Aufbau einer Spezifikation

Eine *DoDL-Spezifikation* nutzt eine Reihe von verschiedenen Techniken zur Beschreibung der verschiedenen Teile eines Hyperdokumentes. Den Rahmen der Spezifikation, die textuell durchgeführt wird, bildet ein objektorientiertes Modell, das mit Hilfe von Spezialisierung und Aggregation sehr flexibel den Aufbau von wiederverwendbaren Dokumentklassen ermöglicht. Außerdem existieren mit generischen (parametrisierten) Klassen und einem allgemeinen Listenkonstrukt sprachliche Mittel, die in speziellen Anwendungsfällen hilfreich sein können.

Eine Dokumentklasse enthält eine beliebige Anzahl von Attributen, die als Platzhalter entweder für Medienobjekte oder für Instanzen von Dokumentklassen dienen. Letzteres erlaubt eine Schachtelung von Dokumentinstanzen, die zu beliebig tiefen, baumartig strukturierten Spezifikationen führt. Der konkrete Inhalt eines Hyperdokumentes wird in einem gesonderten Abschnitt der Spezifikation festgelegt, den sogenannten *Bindungen*. Dort wird jedem Attribut, das ein Medienobjekt repräsentiert, die eindeutige Kennung eines Medienobjektes in der Datenbank (momentan ist dies der Name einer externen Datei) zugewiesen.

Zur Spezifikation der Struktur des Hyperdokumentes besitzt jede Dokumentklasse einen Abschnitt, in welchem die Positionen und Verweise mit Hilfe einer Programmiersprache konstruiert werden. Bei dieser Sprache handelt es sich derzeit um ANSI-C, wie es etwa in [KeRi90] beschrieben ist. Zusätzlich besteht aber innerhalb des Codes die Möglichkeit, auf die definierten Dokumentklassen und deren Instanzen zuzugreifen bzw. diese zu manipulieren. Die ursprüngliche Variante von *DoDL* sah hier die Verwendung von PROLOG [Brat86] anstelle von C vor und erzeugte den Graphen entsprechend durch die Verwendung von Prädikaten.

Der Vollständigkeit halber soll erwähnt sein, daß die Festlegung des Verhaltens eines Hyperdokumentes auf der Basis von Feature-Termen [Smol92] geschieht. Diese werden zur Bildung von Attributen verwendet, mit denen die Verweise des Hyperdokumentes annotiert werden, sowie zum Beschreiben von Selektionsregeln, die auf diesen Attributen aufbauen. Ein Browser für das aus der Spezifikation entstehende Hyperdokument muß die Attribute und Regeln geeignet interpretieren und das Hyperdokument entsprechend anzeigen. Um die Möglichkeiten, die sich aus dem Ansatz von *DoDL* ergeben, vollständig zu nutzen, wird ein spezielles Laufzeitsystem benötigt, daß in der Lage ist, die Terme zur Laufzeit, also zur Betrachtungszeit des Hyperdokumentes, auszuwerten. Die Feature-Terme spielen jedoch innerhalb der Diplomarbeit keine Rolle, da sich die Arbeit hauptsächlich in den Bereichen Inhalt und Struktur bewegt. Die für eine spätere Implementierung der Feature-Terme nötige Infrastruktur soll allerdings – sofern dies möglich ist – schon jetzt berücksichtigt werden.

2.4.1. Beispiel

Abbildung 2.3 zeigt eine einfache Spezifikation, die das Hyperdokument aus Abbildung 2.1 realisieren soll. Abbildung 2.4 enthält die zugehörigen Bindungen. Zwei Eigenschaften der Spezifikation fallen besonders ins Auge: Zunächst wurde der Code der Methode `main()` durch „...“ abgekürzt. Grund dafür ist die Tatsache, daß die dort benötigten Funktionen erst innerhalb dieser Arbeit entwickelt werden. Der ausführliche Kommentar sollte jedoch verdeutlichen, worin der Sinn der Methode liegt. Außerdem fällt auf, daß die Spezifikation neben den beiden aus [Dobe95] bekannten Medientypen `Text` und `Graphics` einen neuen Medientyp `Listing` (Zeile 5) verwendet. Dieser soll zur Repräsentation von bereits formatiertem Text dienen, speziell Quellcode, der ja in einem HTML-Browser nur dann anders behandelt werden kann als Fließtext, wenn er speziell kenntlich gemacht wird. Anhand von `Listing` soll im weiteren Verlauf der Arbeit verdeutlicht werden, wie dem System neue Medientypen hinzugefügt werden können.

```
class Simple is Document with
  documents
    introTxt :   Text;
    figurePic :  Graphics;
    samplePrg :  Listing;
    hyperTxt :   Text;
    graphTxt :   Text;
    graphPic :   Graphics;

  construct
    void main(void)
    {
      // Erzeuge hier folgende Verweise:
      //
      // - Von allen Vorkommen von "Hyper" in "introTxt"
      //   auf den Beginn von "hyperTxt".
      // - Von allen Vorkommen von "Graphen" in "introTxt"
      //   auf den Beginn von "graphTxt".
      // - Von allen Vorkommen von "Graphen" in "hyperTxt"
      //   auf den Beginn von "graphTxt".
      // - Vom Ende von "introTxt" zum Beginn von "figurePic".
      // - Vom Ende von "figurePic" zum Beginn von "samplePrg".
      // - Vom Ende von "graphTxt" zum Beginn von "graphPic".
      //
      // Erzeuge außerdem für den zuletzt genannten Verweis
      // ein Attribut mit dem Namen "Mathe" und dem Wert "Nein".

      ...
    }

end Simple;
```

Abbildung 2.3.: Spezifikation des Beispiels

```
binding Simple is
  mainTxt   = "intro.txt";
  mainPic   = "figure.gif";
  mainPrg   = "code.java";

  hyperTxt = "hyper.txt";
  graphTxt = "graph.txt";
  graphPic = "graph.gif";
end;
```

Abbildung 2.4.: Bindungen des Beispiels

2.4.2. Übersetzung einer Spezifikation

Es existiert bereits ein Compiler für *DoDL* [FrPl99], an dessen Entwicklung der Autor im Rahmen seiner Tätigkeit als wissenschaftliche Hilfskraft am Lehrstuhl beteiligt war. Der Compiler ist in der Lage, die Spezifikation eines Hyperdokumentes mit den dazugehörigen Bindungen in ein ausführbares Programm zu übersetzen. Der *DoDL*-Compiler dient hier als eine Art erweiterter Präprozessor, der die Klassenstruktur und den eingebetteten „objektorientierten" C-Code in reinen ANSI-C-Code umwandelt. Dieser ANSI-C-Code wird dann mit einem entsprechenden Compiler in ein lauffähiges Programm übersetzt. Beim Starten des Programms wird der die Positionen und Verweise generierende Code der einzelnen Dokumentinstanzen ausgeführt. Als Ergebnis entsteht ein attributierter Graph, der das Hyperdokument auf eine abstrakte Weise repräsentiert.

Dem *DoDL*-Projekt fehlt derzeit eine Möglichkeit, eventuell mit Hilfe eines weiteren Werkzeuges, diesen Graphen in konkrete Zielformate umzuwandeln, die mit einem geeigneten Browser betrachtet werden können. Es liegt auf der Hand, daß ein solches Werkzeug von großem Interesse ist, da es eine Visualisierung und somit auch Validierung sowohl der spezifizierten Hyperdokumente als auch des gesamten *DoDL*-Konzeptes erlaubt.

2.5. Aufgabenstellung

Die Aufgabe der Diplomarbeit besteht darin, die fehlende Brücke zwischen dem bestehenden *DoDL*-Compiler und mindestens einem konkreten Zielformat für Hyperdokumente zu schlagen. Dieses Zielformat soll naheliegenderweise HTML sein, da auf jeder Plattform Werkzeuge zur Anzeige von HTML-Dokumenten zur Verfügung stehen. Zusätzlich bietet sich durch die Wahl von HTML als Zielformat die Möglichkeit, mit Hilfe von *DoDL* unmittelbar Dokumente für das WWW zu erzeugen.

Abbildung 2.5 gibt einen Überblick über die Aufgabenstellung. Die abgerundeten Rechtecke repräsentieren die einzelnen Schritte des gesamten Prozesses, spitze Rechtecke zeigen die am Prozeß beteiligten Dateien, und Pfeile stehen für Zugriffe auf diese Dateien bzw. für Informationen innerhalb der Dateien, die von außen referenziert werden. Diese Grafik wird in den folgenden Abschnitten, die den Lösungsansatz und die Vorgehensweise erläutern, schrittweise verfeinert.

Die einzelnen Schritte bei der Erzeugung eines Hyperdokumentes sehen etwa wie folgt aus: Zunächst existieren in einer Datenbank Ⓐ nur die einzelnen Medienobjekte, aus denen sich das Hyperdokument zusammensetzen wird. Aufbauend auf diesen Medienobjekten wird eine textuelle Spezifikation Ⓑ des Hyperdokumentes entworfen, innerhalb welcher die Medienobjekte referenziert werden. Aus der Spezifikation wird mit Hilfe des bestehenden *DoDL*-Compilers und eines ANSI-C-Compilers ein ausführbares Programm Ⓒ erzeugt. Der Compilierungsvorgang wird durch ① repräsentiert. Ein weiterer Vorgang ist nötig, der den beim Ausführen des Programms entstehenden attributierten Graphen

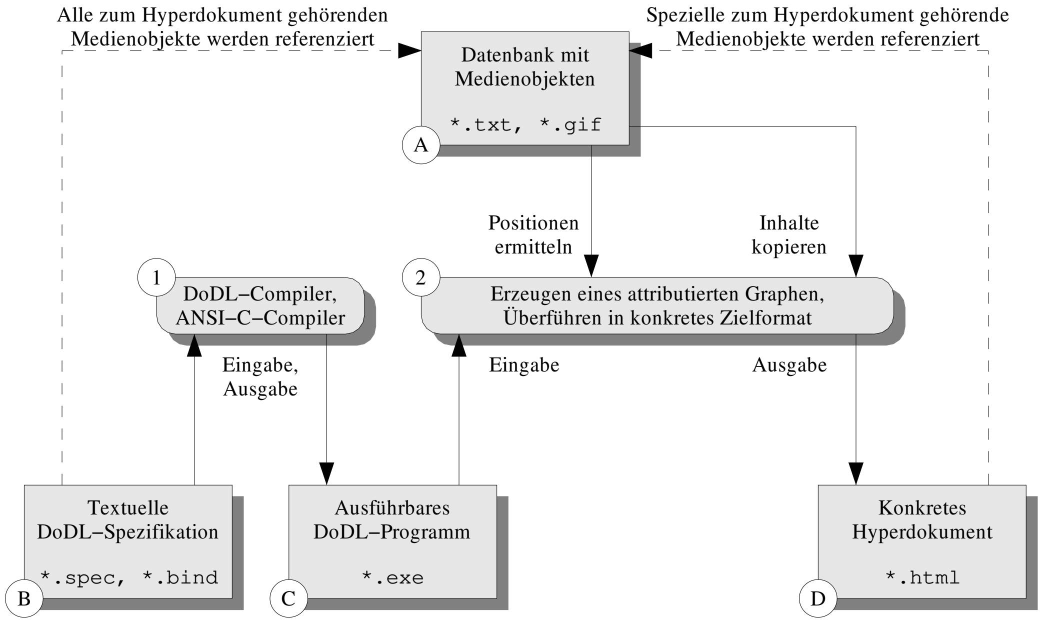

Abbildung 2.5.: Aufgabenstellung der Diplomarbeit

in das endgültige Hyperdokument (D) umwandelt. In diesem Schritt, der durch (2) repräsentiert wird, fließen erstmals auch die Medienobjekte selbst in den Prozeß ein: Zum einen werden sie zum Auffinden von Positionen benötigt, zum anderen werden selbstverständlich die Medienobjekte bzw. deren Inhalte in das erzeugte Hyperdokument übernommen. Ob dabei sämtliche Medienobjekte in das Zieldokument kopiert werden, hängt vom verwendeten Ausgabeformat ab. Im konkreten Fall von HTML werden zwar Texte komplett in die Zieldatei(en) übernommen, Grafiken und andere multimediale Objekte aber weiterhin als externe Dateien referenziert. Diesen Zusammenhang soll der rechte gestrichelte Pfeil andeuten.

2.6. Lösungsansatz

Die Grundidee zur Lösung der gestellten Aufgabe beruht auf einer Aufteilung des zweiten Schrittes, so daß sich ein insgesamt dreischrittiges Verfahren ergibt. Alle drei Schritte werden von verschiedenen Programmen durchgeführt:

1. Der *DoDL*-Compiler liest die Spezifikation eines Hyperdokumentes sowie die zugehörigen Bindungen und erzeugt daraus unter Verwendung eines ANSI-C-Compilers ein ausführbares *DoDL*-Programm.
2. Das ausführbare Programm wird gestartet. Es erzeugt mit Hilfe des (compilierten) C-Codes der einzelnen Dokumentklassen eine Datenstruktur, die das gewünschte

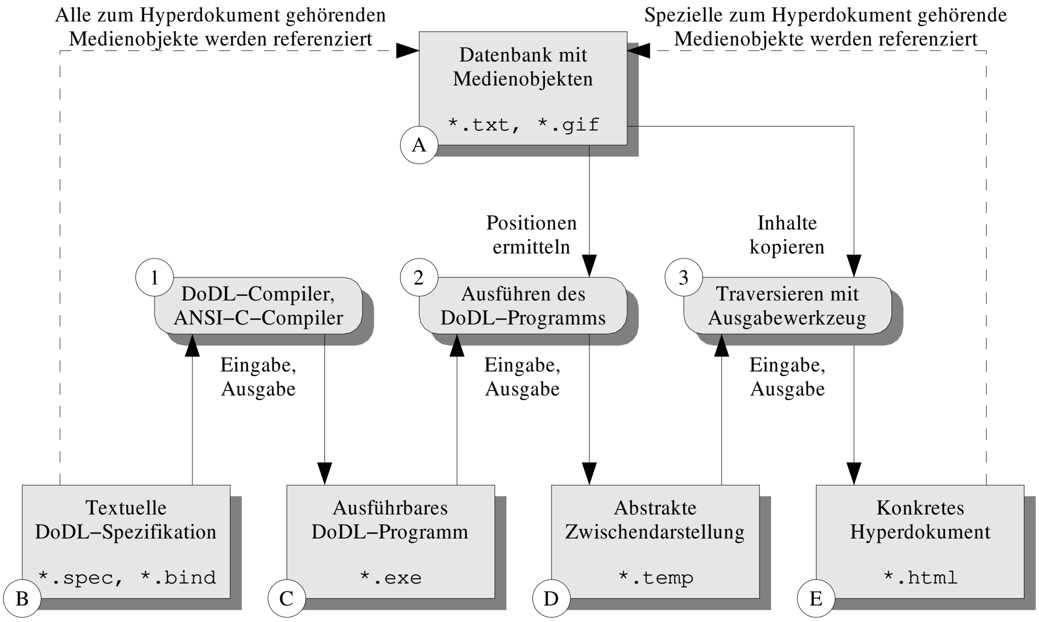

Abbildung 2.6.: Lösungsansatz der Diplomarbeit

Hyperdokument abstrakt repräsentiert, wobei *abstrakt* bedeutet, daß die Datenstruktur noch nicht an ein spezielles Zielformat gebunden ist. Die Datenstruktur, die unter anderem den attributierten Graphen umfaßt, wird in eine externe Datei geschrieben, die als *Zwischendarstellung* bezeichnet werden soll.

3. Ein weiteres Programm, das eigentliche Ausgabewerkzeug, liest diese Zwischendarstellung, traversiert die Datenstruktur geeignet und erzeugt als Ausgabe ein – möglicherweise über mehrere Dateien verteiltes – konkretes Hyperdokument. Mit *konkret* ist an dieser Stelle gemeint, daß das Ergebnis im gewünschten Zielformat, hier also HTML, vorliegt und mit einem entsprechenden Browser betrachtet werden kann.

Abbildung 2.6 zeigt, wie sich die drei Schritte in die Aufgabenstellung einfügen und wie das Zusammenspiel der einzelnen Schritte und der beteiligten Dateien aussieht.

Es sei an dieser Stelle darauf hingewiesen, daß sich aus der Abstützung auf die Zwischendarstellung eine Reihe von Vorteilen ergibt: So ist es zum Beispiel möglich, Hyperdokumente mit Hilfe des *DoDL*-Compilers aus Spezifikationen zu gewinnen, dies ist aber nicht notwendigerweise der einzige Weg. Es ist auch möglich, mit einem Programm (unter Umgehung des *DoDL*-Compilers) direkt die Zwischendarstellung zu erzeugen, wenn dies in einem speziellen Anwendungsfall gewünscht wird. Man stelle sich etwa ein Hyperdokument vor, das regelmäßig komplett von einer Software produziert wird, z.B. eine Darstellung der Topologie und Auslastung eines Rechnernetzes oder die Auflistung des

Inhalts einer Datenbank. Die Software besitzt bereits alle nötigen Daten zum Erzeugen des Hyperdokumentes. Die Spezifikation wäre hier zwar ein möglicher, aber dennoch überflüssiger Schritt.

Zusätzlich kann die Zwischendarstellung als plattformunabhängiges, austauschbares Zielformat für alle zukünftigen *DoDL*-Compiler dienen, etwa solche, bei denen Ziel- und/oder Gastsprache nicht ANSI-C sind, sondern z.B. Pascal, Modula oder Java. Weiterhin können auf der Zwischendarstellung andere Werkzeuge aufbauen, zum Beispiel spezielle Debugger für Hyperdokumente. Es ist klar, daß eine sorgfältige Definition der Zwischendarstellung bzw. des verwendeten Dateiformates von hoher Bedeutung ist.

2.7. Vorgehensweise

Betrachtet man das dargestellte System, so ergeben sich drei größere Teilaufgaben für die Diplomarbeit:

1. Entwurf einer *Laufzeitbibliothek* für den *DoDL*-Compiler. Diese Laufzeitbibliothek erweitert den Compiler um die benötigte Funktionalität zum Erzeugen des Graphen, der die Verknüpfungsstruktur eines Hyperdokumentes repräsentiert.

2. Entwurf einer *Zwischendarstellung* für Hyperdokumente. Die Datenstruktur, die das Hyperdokument im ausführbaren Programm repräsentiert, muß in diese abstrakte Zwischendarstellung überführt werden.

3. Entwurf eines *Ausgabewerkzeugs* für HTML. Das Werkzeug muß die Zwischendarstellung einlesen, die Datenstruktur geeignet traversieren und daraus ein konkretes HTML-Dokument generieren.

Abbildung 2.7 zeigt eine um die drei Teilaufgaben ergänzte Sicht des gesamten Systems. Jeder Teilaufgabe ist eines der folgenden Kapitel gewidmet, die Abfolge entspricht aber nicht der Reihenfolge, in der die Teilaufgaben tatsächlich bearbeitet wurden. Aufgrund der zentralen Bedeutung der Zwischendarstellung wurde diese definiert, bevor mit der Implementierung von Bibliothek und Ausgabewerkzeug begonnen wurde. Um jedoch das gesamte System leichter darstellen zu können, entsprechen die Kapitel in der Arbeit der Reihenfolge, in der ein Hyperdokument das *DoDL*-System durchläuft.

Der rote Faden, der sich durch die einzelnen Kapitel zieht, soll dabei das Hyperdokument aus Abbildung 2.1 sein. Es wird entweder komplett oder in Ausschnitten immer wieder Verwendung finden, so daß am Ende von Kapitel 5 schließlich die Darstellung dieses Hyperdokumentes in einem HTML-Browser zu sehen sein wird.

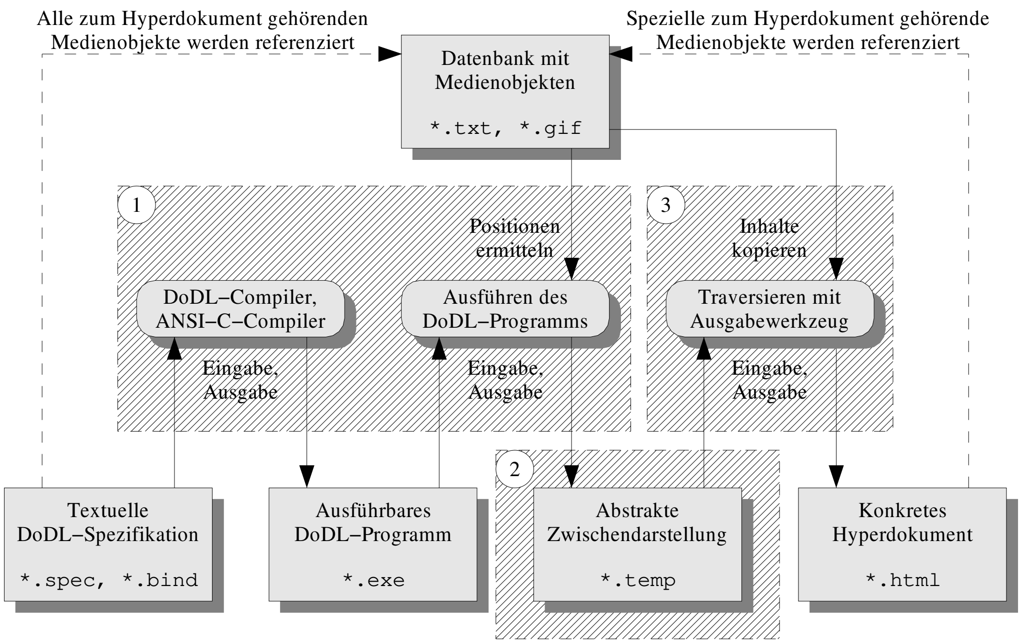

Abbildung 2.7.: Teilaufgaben der Diplomarbeit

3. Eine Laufzeitbibliothek für den *DoDL*-Compiler

Die erste Teilaufgabe der Diplomarbeit umfaßt Erweiterungen des bestehenden *DoDL*-Compilers. Der Compiler, wie er in [FrPl99] beschrieben ist, implementiert nur die „nackte“ Sprache *DoDL*, also die Übersetzung einer Spezifikation und der dazugehörigen Bindungen zunächst in ANSI-C-Code und anschließend in ein ausführbares Programm. Es fehlt noch eine Reihe von speziellen Datentypen und Funktionen, durch welche die Verknüpfungsstruktur eines Hyperdokumentes repräsentiert und konstruiert werden kann. Ohne diese Hilfsmittel kann ein *DoDL*-Programm zwar eine Menge verschiedenartiger (Bildschirm-) Ausgaben erzeugen, die gewünschten Hyperdokumente gehören jedoch nicht dazu.

3.1. Anforderungen

Damit mit *DoDL*-Programmen Hyperdokumente erzeugt werden können, ist es notwendig, den *DoDL*-Compiler um eine *Laufzeitbibliothek* zu erweitern. In dieser müssen die fehlenden Datentypen zur Repräsentation von Medienobjekten, Positionen, Verweisen und Attributen bereitgestellt werden. Aufbauend auf diesen Typen können Funktionen zum Auffinden von Positionen in Medienobjekten, zum Erzeugen von Verweisen zwischen Positionen und zum Attributieren dieser Verweise implementiert werden. Aus der dabei entstehenden Datenstruktur wird unmittelbar vor Beendigung des *DoDL*-Programms automatisch die externe Zwischendarstellung erzeugt. Mit anderen Worten, die zu entwickelnde Bibliothek muß mindestens die folgenden Elemente enthalten:

- Einen Datentyp `DBUnit`, der ein Medienobjekt repräsentiert. Dieser Typ muß in der Lage sein, zwischen den verschiedenen spezielleren „Untertypen“ `Text`, `Graphics` und `Listing` zu unterscheiden.
- Einen Datentyp `Position`, der eine Position repräsentiert.
- Einen Datentyp `Link`, der einen Verweis repräsentiert.
- Einen Datentyp `Attribute`, der ein Attribut repräsentiert.

- Eine Funktion `begin()`, welche die ausgezeichnete Position ermittelt, die dem Beginn eines Medienobjektes entspricht.
- Eine Funktion `end()`, welche die ausgezeichnete Position ermittelt, die dem Ende eines Medienobjektes entspricht.
- Eine Funktion `occ()`, welche sämtliche Positionen ermittelt, an denen ein vorgegebener Suchparameter in einem Medienobjekt (etwa eine Zeichenkette in einem Text) vorkommt.
- Eine Funktion `link()`, die einen Verweis zwischen zwei Positionen erzeugt.
- Eine Funktion `bindAll()`, die Verweise zwischen einer Liste von (Quell-) Positionen und einer einzelnen (Ziel-) Position erzeugt.
- Eine Funktion `setAttribute()`, die einem Verweis ein Attribut hinzufügt.
- Eine Funktion `setValue()`, die den Wert eines Attributes ändert.
- Eine Funktion `save()`, die aus der im Speicher entstandenen Datenstruktur die externe Zwischendarstellung erzeugt.

Die benötigten Datentypen ergeben sich aus [Dobe96a], wobei die Unterscheidung zwischen den verschiedenen Untertypen von `DBUnit` dem entspricht, was das Prolog-Prädikat `extType` leistet. Die ersten fünf Funktionen ergeben sich ebenfalls aus [Dobe96a]. Dabei wurde die Arbeitsweise von `bindAll()` von einer Liste von Zielpositionen auf eine einzelne Zielposition eingeschränkt, da dies der praktisch einzig relevante Fall ist. Die beiden folgenden Funktionen entstammen dem ersten Kapitel von [FrPl99], und der Name der letzten Funktion leitet sich schlicht aus der Tatsache ab, daß diese die Datenstruktur in eine Datei „sichert".

Um dem bislang unentdeckt gebliebenen Konflikt zwischen der Funktion `end()` und dem gleichnamigen reservierten Wort der Sprache *DoDL* zu entgehen und um die Namensgebung insgesamt konsistenter zu gestalten, werden einige der Funktionen im weiteren Verlauf des Kapitels eine Umbenennung erfahren. Ihre Semantik bleibt davon jedoch unberührt. Die genannten Datentypen und Funktionen stellen also den minimalen Umfang einer Laufzeitbibliothek für den *DoDL*-Compiler dar.

3.2. Eine imperative Lösung

Nachdem geklärt ist, welche Datentypen und Funktionen innerhalb der Laufzeitbibliothek benötigt werden, stellt sich nun die Frage, wie diese dem bestehenden System am besten hinzugefügt werden können. Eine Lösung, die in der Anfangsphase der Diplomarbeit vorgeschlagen und diskutiert wurde, beruht auf den beiden folgenden Ideen:

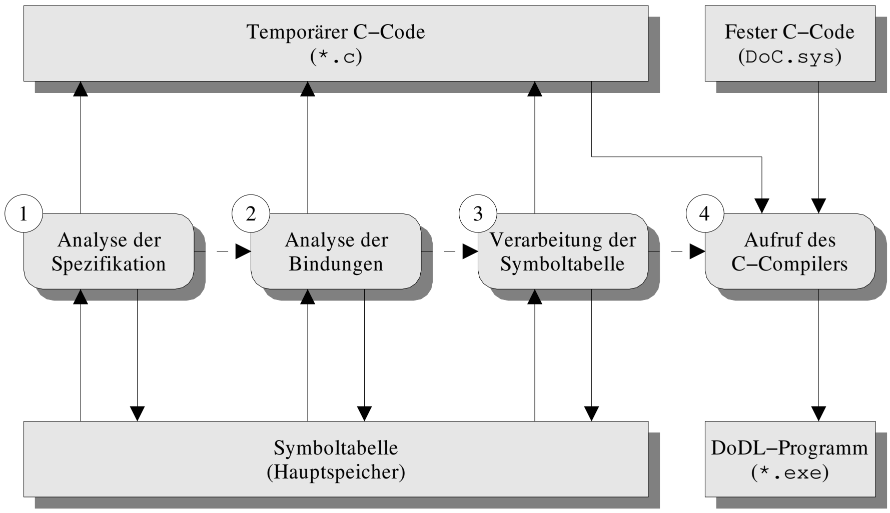

Abbildung 3.1.: Vorgänge im *DoDL*-Compiler

- Die Verknüpfungsstruktur, die von dem Programm erzeugt wird, ist ein attributierter Graph. Positionen und Verweise entsprechen unmittelbar Knoten und Kanten. Es liegt nahe, diesen Graphen als zentrale Datenstruktur des ausführbaren *DoDL*-Programms zu betrachten und die restlichen Elemente entweder „um den Graphen herum“ anzuordnen oder in diesen zu integrieren.

- Zur Implementierung wird die Datei `DoC.sys` benutzt, die bereits Bestandteil des *DoDL*-Compilers ist. Diese Datei enthält festen, von der übersetzten Spezifikation unabhängigen ANSI-C-Code, der über eine `#include`-Anweisung in jedes erzeugte *DoDL*-Programm eingebunden wird. In `DoC.sys` bereitgestellte Funktionen stehen allen Spezifikationen automatisch zur Verfügung. Datentypen müssen zusätzlich der Symboltabelle während der Initialisierung bekannt gemacht werden, damit sie in der `documents`-Sektion einer Spezifikation nutzbar sind. Dieses Verfahren wird zum Beispiel für die Basisklasse `Object` und den primitiven Typ `string` verwendet.

Abbildung 3.1 zeigt anhand eines sehr groben Überblicks über die Vorgänge im *DoDL*-Compiler, wo sich die Datei `DoC.sys` in diesem System wiederfindet. Die durchgezogenen Pfeile stellen Datenfluß dar, während die gestrichelten Pfeile Kontrollfluß symbolisieren.

Der Vorteil dieser Lösung besteht darin, daß sie ohne großen Aufwand zu realisieren ist. Der Typ `DBUnit` kann als C-Struktur deklariert werden, die den Namen und den konkreten Typ eines Medienobjektes enthält. Die Typen `Position`, `Link` und `Attribute` sind schnell als verzeigerte C-Strukturen implementiert, ebenso die meisten der benötigten Funktionen. Einzig `occ()` und `save()` verlangen etwas Arbeit: Die Implementierung von `occ()`, die nur für die Medientypen `Text` und `Listing` durchgeführt wird, muß sämtliche Vorkommen einer gegebenen Zeichenkette in einer Datei ermitteln. Die Funktion `save()`

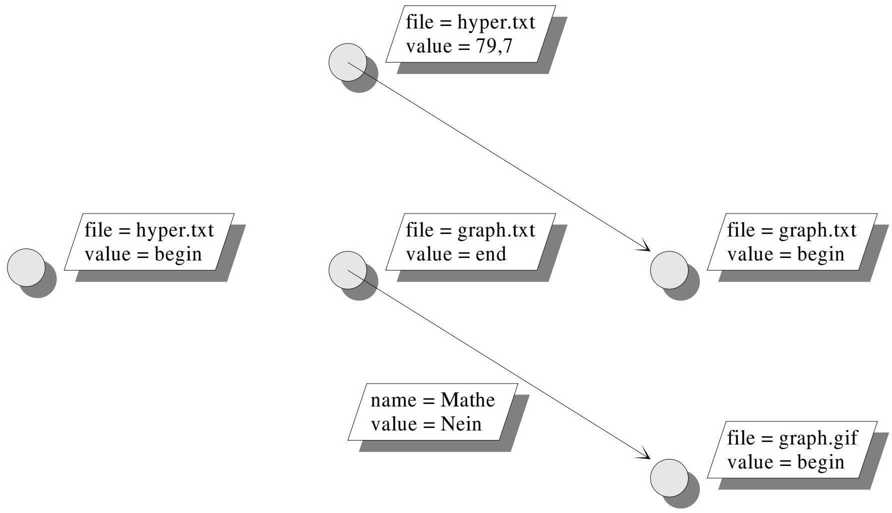

Abbildung 3.2.: Struktur des Beispiel-Hyperdokumentes als Graph

muß den Graphen geeignet traversieren und dabei die externe Zwischendarstellung erzeugen.

Versucht man jedoch, dieses „den Graphen geeignet traversieren“ etwas genauer zu fassen, dann tritt bereits ein erster Nachteil der vorgestellten Lösung zu Tage: Es stellt sich nämlich heraus, daß der Graph allein gar nicht alle Informationen enthält, die zum Erzeugen der Datei benötigt werden. Angaben zu Namen und Typen der einzelnen Medienobjekte fehlen zum Beispiel völlig. Es ist natürlich möglich, diese Informationen „nachzurüsten“, indem sie in Form von zusätzlichen Attributen an jede Position gehängt werden, wie [FrPl99] dies in Kapitel 1 in einigen Beispielen vorschlägt. Abbildung 3.2 zeigt das Ergebnis dieser Überlegungen für die rechte Hälfte des Hyperdokumentes aus Abbildung 2.1. Aber auch dies ist keine angemessene Lösung: Man denke nur an die spätere Integration der Selektionsregeln, die ja nicht Teil dieser Arbeit sind, in den Graphen. Natürlich ist es auch hier möglich, jeden Verweis (also jede Kante) zusätzlich mit allen Regeln zu attributieren, die ihn betreffen. Aber spätestens an dieser Stelle sollte klar werden, daß es keinen Sinn macht, sämtliche Informationen über das Hyperdokument mit Zusatzattributen in eine einfache Graphstruktur zu zwängen. Stattdessen sollte nach einer Datenstruktur gesucht werden, die alle Bestandteile des Hyperdokumentes – Inhalt, Struktur und Verhalten – angemessen beschreibt.

Doch nicht nur die Datenstruktur selbst weist Mängel auf, auch die Implementierung in C besitzt Nachteile. Einer davon ist die sehr aufwendige Wartung des Systems, die der einfachen Erstimplementierung gegenübersteht. Grund dafür ist die Tatsache, daß zum Beispiel ein neuer Medientyp nicht nur Änderungen an der Datei `DoC.sys` zur Folge hat, sondern auch am Quellcode des eigentlichen Compilers. Zum einen muß der neue Typ in die Symboltabelle eingetragen werden, damit er bekannt ist und genutzt werden kann. Zum anderen muß die Implementierung der Funktion `occ()` um den neuen Medientyp erweitert werden. Da `occ()` in C nur durch eine große `switch`-Anweisung zur Unterscheidung der Medientypen zu realisieren ist, verkommt die Funktion bei mehr als einer Handvoll Medientypen schnell zu einem schwer durchschaubaren Konstrukt. Erweiterungen der Laufzeitbibliothek setzen damit sehr grundlegende Kenntnisse über die Struktur des Compilers voraus, was den Kreis der Personen, die an der Bibliothek arbeiten können, unnötig einschränkt. Die allgemeine Wartungsunfreundlichkeit von fremdem C-Code tut ein übriges dazu, so daß vermutlich nur genau die zwei Personen jemals an der Bibliothek arbeiten können, die auch den Compiler selbst entwickelt haben.

Schließlich, und auch dieser Einwand sollte nicht völlig unberücksichtigt bleiben, stellt sich einem Benutzer das System *DoDL* bei der vorgestellten Lösung als Mischung aus objektorientierten und imperativen Konzepten dar. Obwohl dies eher eine Stilfrage ist, die nicht als wesentliches Argument für oder gegen eine bestimmte Lösung verwendet werden kann, stellt sich doch die Frage, ob nicht eine Lösung existiert, die die aufgezählten Nachteile vermeidet und zudem im Bezug auf das gesamte Erscheinungsbild etwas mehr Konsistenz und damit auch Eleganz besitzt.

3.3. Eine objektorientierte Lösung

Eine alternative Implementierung der Laufzeitbibliothek, die den Benutzer nicht mehr mit der unschönen Mixtur aus objektorientierter und imperativer Programmierung konfrontiert, muß vollständig objektorientiert sein, denn *DoDL* ist eine objektorientierte Sprache. Da es weder Sinn macht noch den gewünschten Effekt hat, etwa C++ anstelle von C für die Laufzeitbibliothek zu verwenden, bleibt als Implementierungssprache nur *DoDL* selbst übrig: Die einzelnen Datentypen werden dabei zu *DoDL*-Klassen, die benötigten Funktionen werden entsprechend als Methoden auf diese Klassen verteilt. Die Klassen müssen dem Compiler in Form einer Spezifikation – etwa einer Datei `DoC.spec` – zugeführt werden, die automatisch übersetzt wird, bevor die Verarbeitung der Spezifikation des Benutzers beginnt.

Abbildung 3.3 zeigt, wie sich die über eine Spezifikation realisierte Laufzeitbibliothek in den Compiler einfügt. Wie leicht einzusehen ist, hat diese Art der Implementierung keinen Einfluß auf den Quellcode der Symboltabelle, da sämtliche Klassen gewissermaßen „auf dem üblichen Weg“ in die Symboltabelle aufgenommen werden, nämlich durch die syntaktische Analyse. Folglich ist der Compiler selbst von Änderungen der Laufzeitbibliothek nicht betroffen, wodurch sich deren Wartung und Erweiterung im Vergleich zur vorherigen Lösung erheblich vereinfacht.

Der Graph, die zentrale Datenstruktur der imperativen Lösung, läßt sich ohne großen Aufwand in die objektorientierte Welt übersetzen: Die Datentypen `Position`, `Link` und `Attribute` werden zu Klassen, deren Beziehungen zueinander sich dann sehr einfach durch Assoziation ausdrücken lassen: Eine Instanz von `Position` ist mit den Instanzen von `Link` assoziiert, die von der Position ausgehen. Entsprechend ist eine Instanz von `Link` mit den Instanzen von `Attribute` assoziiert, die den Verweis „schmücken“. Zusätzlich

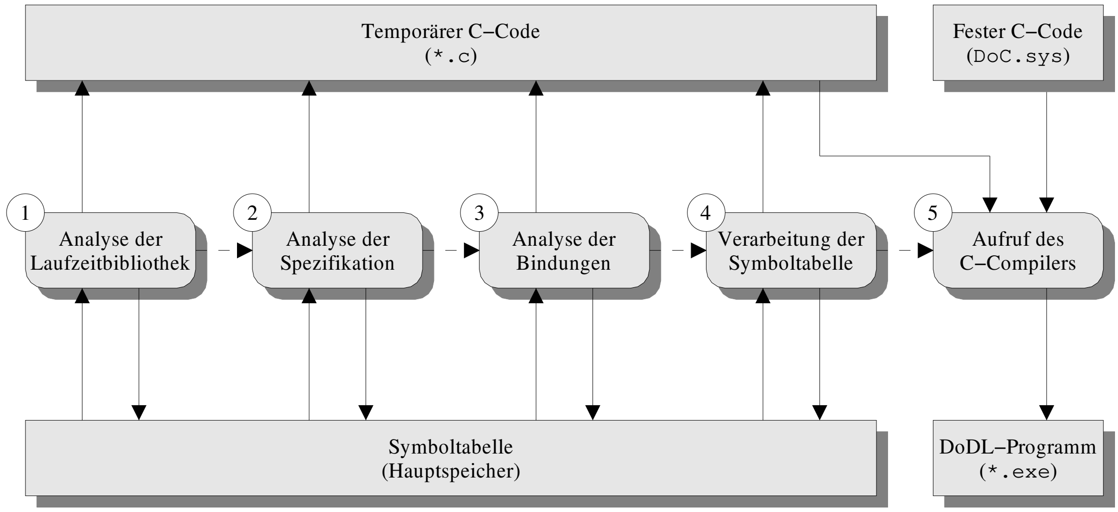

Abbildung 3.3.: Vorgänge im *DoDL*-Compiler

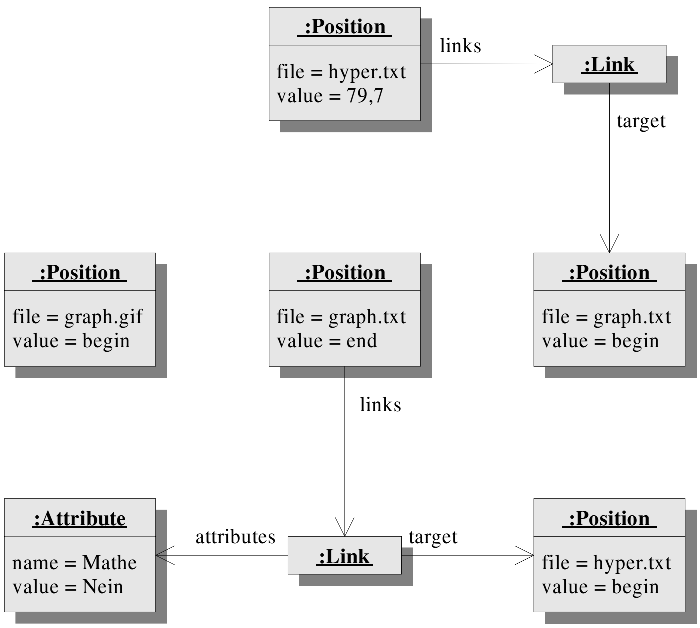

Abbildung 3.4.: Struktur des Beispiel-Hyperdokumentes als Graph

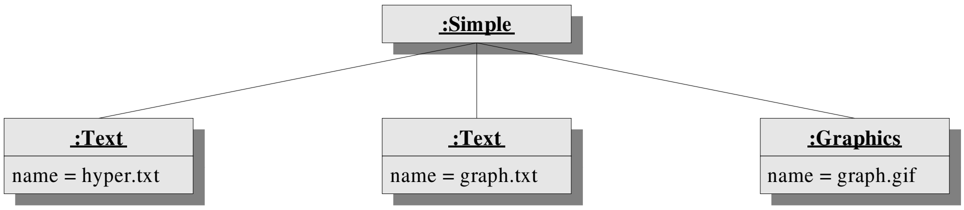

Abbildung 3.5.: Inhalt des Beispiel-Hyperdokumentes als Baum

besteht eine Assoziation zwischen jeder Instanz von `Link` und der Instanz von `Position`, die Ziel des Verweises ist. Abbildung 3.4 zeigt ein Objektdiagramm, das den Graphen aus Abbildung 3.2 durch Instanzen dieser drei Klassen beschreibt.

Zur Repräsentation der Medienobjekte bieten sich ebenfalls Klassen an. Die Information über den konkreten Typ eines Medienobjektes muß dann nicht mehr in einem Attribut von `DBUnit` codiert werden, wie dies bei der imperativen Lösung der Fall war. Stattdessen wird `DBUnit` zu einem abstrakten Vorfahren der spezialisierten Klassen `Text` und `Graphics`. Der Medientyp `Listing` kann als Nachkomme von `Text` realisiert werden. Es ist leicht einzusehen, daß dem System auf diese Weise sehr leicht neue Medientypen hinzugefügt werden können, die – wie im Fall von `Listing` – sogar einen Großteil der Funktionalität ihres Vorfahren erben.

Die Funktionen `begin()`, `end()` und `occ()` werden zu Methoden der vorgestellten Klassen. Dadurch und durch den Mechanismus der Methodenzuteilung zur Laufzeit fällt ihre Implementierung nicht nur einfacher aus als bei der imperativen Lösung, sondern auch wesentlich eleganter: Jede Klasse, die zum Beispiel `occ()` benötigt, implementiert die Methode auf ihre spezifische Weise. Es ist insbesondere kein zentrales `switch`-Konstrukt mehr notwendig, das in Abhängigkeit vom Medientyp eine Behandlung vornimmt. Der *DoDL*-Compiler bzw. das Laufzeitsystem sorgen dafür, daß stets die richtige Implementierung aufgerufen wird.

Die Medienobjekte stehen natürlich nicht für sich. Sie müssen, da sie in den `documents`-Sektionen einer Spezifikation verwendet werden, in die Klassen eingebettet werden, die der Benutzer entwirft. Hier scheint sich also eine Schnittstelle zwischen den Bibliotheksklassen und den Benutzerklassen zu befinden. Diese Schnittstelle entpuppt sich bei genauerer Betrachtung als *1:n*-Aggregation, denn jede Benutzerklasse kann beliebig viele Medienobjekte verwenden, die ihr direkt untergeordnet sind. Zwischen den Benutzerklassen selbst besteht genau die gleiche Beziehung, denn jede Benutzerklasse kann ebenfalls beliebig viele andere Benutzerklassen aggregieren.

Aus diesen Überlegungen ergibt sich für die Inhaltsrepräsentation des bekannten Beispiels eine Struktur, wie sie in Abbildung 3.5 dargestellt ist. Offensichtlich bilden die Objekte, aus denen sich der Inhalt zusammensetzt, eine Baumstruktur. Da bei dem Baum die Beziehung zwischen über- und untergeordneten Knoten aus der Anordnung ersichtlich ist, wurde der Einfachheit halber auf Pfeilspitzen und Beschriftungen an den Verbindungsli-

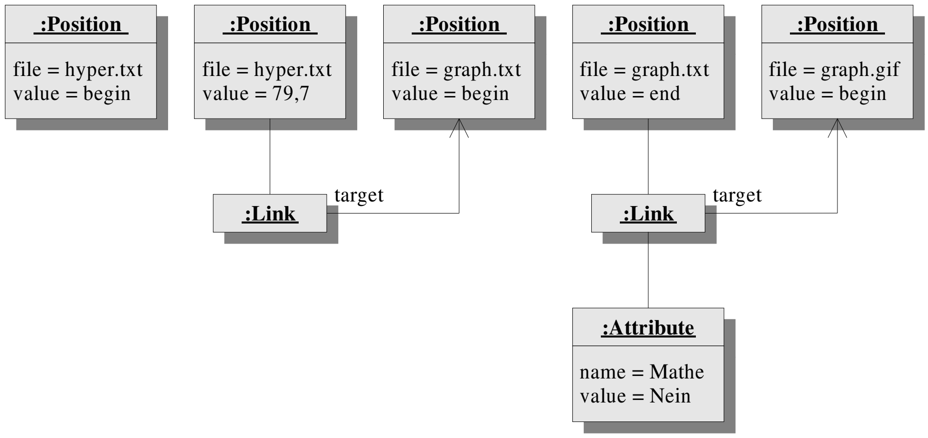

Abbildung 3.6.: Struktur des Beispiel-Hyperdokumentes als Baum

nien verzichtet, was im folgenden beibehalten werden soll.

Weniger offensichtlich ist, daß auch die Objekte, aus denen sich der Strukturgraph zusammensetzt, eine Baumstruktur bilden bzw. also solche aufgefaßt werden können. Läßt man die Assoziation `target` zwischen einem Verweis und seinem Ziel außer acht, dann bleiben mehrere zyklenfreie *1:n*-Assoziationen, die auch als Aggregationen betrachtet werden können, durch die Positionen, Verweise und Attribute geordnet werden. Diese Struktur, die in Abbildung 3.6 dargestellt ist, ist offensichtlich ebenfalls hierarchisch. Sie hat nur den Nachteil, daß sie nicht einen einzigen Baum bildet, sondern einen ganzen Wald. Dieser Nachteil läßt sich aber sehr leicht aus dem Weg räumen, wenn man versucht, die beiden Strukturen, die sich aus Inhalt und Struktur ergeben, zu vereinen.

Diese Vereinigung geschieht auf der Ebene von Medienobjekten und Positionen. Statt die Informationen über Medienobjekte in den Graphen zu zwängen, wie es bei der imperativen Lösung der Fall war, wird nun der Graph über eine weitere *1:n*-Aggregation an die Medienobjekte angehängt. Jedem Medienobjekt werden die Positionen untergeordnet, die sich in ihm befinden. Damit fügt sich jeder Wald des Strukturbaumes nahtlos in den Inhaltsbaum ein. Wie Abbildung 3.7 zeigt, ergibt sich der gewünschte einheitliche Baum von Objekten, der das gesamte Hyperdokument konsistent beschreibt. Der attributierte Graph, der die Verknüpfungsstruktur repräsentiert, rückt dabei etwas in den Hintergrund. Er ist in diesem Modell nicht mehr zentrale Datenstruktur, sondern nur noch Teil einer größeren.

Die Vorteil der einheitlichen Datenstruktur ist offensichtlich: Sie ist durch einen Tiefendurchlauf leicht zu traversieren. Im Gegensatz zur imperativen Lösung, deren Graph eventuell nicht zusammenhängend ist, werden hier in jedem Fall alle Medienobjekte beim Traversieren berücksichtigt, da der Baum immer zusammenhängend ist.

Analog zum vorangehenden Abschnitt sollen auch für die objektorientierte Lösung die beiden zentralen Ideen kurz herausgestellt werden:

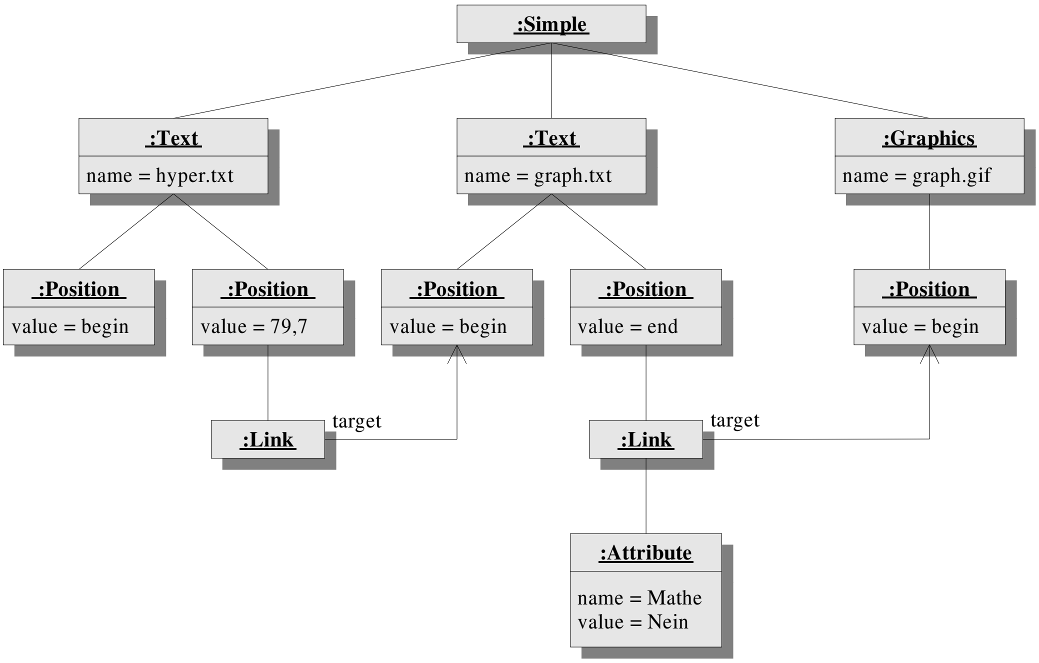

Abbildung 3.7.: Gesamtes Beispiel-Hyperdokument als Baum

- Sämtliche Klassen geben die Begriffe und Zusammenhänge von *DoDL* intuitiv wieder. Zwischen allen Elementen besteht nicht nur eine Vererbungsbeziehung, die Wiederverwertung von Code ermöglicht, sondern auch eine *1:n*-Aggregation, über die alle Objekte in eine Baumstruktur eingeordnet werden. Durch Traversieren dieser Baumstruktur von der Wurzel aus kann die externe Zwischendarstellung erzeugt werden.

- Zur Implementierung der Laufzeitbibliothek werden eine oder mehrere Spezifikationen verwendet, die automatisch vor denen des Benutzers übersetzt werden. Diese Spezifikationen enthalten alle Klassen, die zum Erzeugen von Hyperdokumenten benötigt werden.

Es sei noch angemerkt, daß die auf den ersten Blick etwas unorthodoxe Lösung, die für die Implementierung der Laufzeitbibliothek gewählt wurde, aus der Sicht des Entwicklers eines Compilers gar nicht so ungewöhnlich ist. Es ist eine übliche Praxis, nur den Teil der Laufzeitbibliothek in der Zielsprache des Compilers (meist Assembler, in diesem Fall C) zu realisieren, für den dies unumgänglich ist. Der weitaus größte Teil der Bibliothek wird nach der persönlichen Erfahrung des Autors mit Sprachen wie Pascal, Java und C in der Quellsprache des Compilers implementiert. Dies erleichtert nicht zuletzt eine Portierung des Compilers auf andere Zielplattformen. Auch im Fall von *DoDL* hat sich diese Vorgehensweise bereits ausgezahlt, denn im Rahmen einer parallel stattfindenden

Diplomarbeit [Haas99] konnte die Bibliothek mit sehr geringen Änderungen für einen weiteren *DoDL*-Compiler verwendet werden, der Java als Zielsprache benutzt.

3.4. Änderungen am Compiler

Es ist leicht einzusehen, daß einige Änderungen am *DoDL*-Compiler notwendig sind, um die Laufzeitbibliothek auf die vorgestellte objektorientierte Weise realisieren zu können. Diese betreffen im wesentlichen die Übersetzung von *DoDL*-Klassen in C-Code, bewegen sich also auf der Ebene der Codegenerierung.

Das Objektmodell des Compilers aus [FrPl99], also die Art und Weise, wie Objekte dort implementiert sind, läßt sich am besten durch die beiden folgenden Begriffe charakterisieren:

- Es ist *statisch*. Anzahl und Typ der im Programm verwendeten Objekte stehen spätestens nach der syntaktischen Analyse der Bindungen, in jedem Fall aber noch zur Zeit der Übersetzung fest.
- Es ist *strukturbasiert*. Eine *DoDL*-Klasse wird also direkt in eine C-Struktur übersetzt. Das hat konkrete Auswirkungen auf den Umgang mit Objekten. Wird einem Objekt `First` zum Beispiel ein weiteres Objekt `Second` zugewiesen (Typgleichheit vorausgesetzt), dann ist `First` anschließend eine identische Kopie von `Second`. Weitere Änderungen an einem der Objekte wirken sich auf das jeweils andere nicht aus.

Es ist unmittelbar klar, daß sich die Laufzeitbibliothek mit diesem Objektmodell nicht oder nur schwer realisieren läßt. Die erste Eigenschaft steht der Notwendigkeit im Weg, zur Laufzeit eine beliebige Anzahl von Positionen, Verweisen und Attributen zu erzeugen. Die zweite Eigenschaft verhindert die Verwendung von Aggregation und Assoziation auf die beschriebene Weise oder erschwert sich doch zumindestens. Grund dafür ist die Tatsache, daß in der `documents`-Sektion neben dem primitiven Typ `string` nur Instanzen von Klassen verwendet werden können, nicht aber Zeiger auf solche Instanzen. Um diese Probleme zu beseitigen, muß das Objektmodell so geändert werden, daß die beiden oben genannten Eigenschaften zugunsten anderer Charakteristika beseitigt werden:

- Das Objektmodell muß *dynamisch* werden. Es muß also möglich sein, zur Laufzeit neue Objekte zu erzeugen. Das erfordert insbesondere die Existenz mindestens einer Konstruktormethode für jede Klasse. Damit die Objekte die beschriebene Baumstruktur bilden können, sollte dem Konstruktur zudem ein übergeordnetes Objekt übergeben werden können, so daß die Wartung der Baumstruktur im Konstruktor automatisiert wird.

- Das Objektmodell muß *referenzbasiert* werden. Eine *DoDL*-Klasse wird also nicht mehr direkt in eine C-Struktur, sondern in einen Zeiger auf eine solche Struktur übersetzt. Wird einem Objekt `First` dann ein weiteres Objekt `Second` zugewiesen (wieder Typgleichheit vorausgesetzt), dann ist `First` anschließend eine weitere Referenz auf das Objekt, auf das auch die Variable `Second` verweist. Weitere Änderungen an einem der Objekte wirken sich folglich auf das jeweils andere aus.

Es sei angemerkt, daß das gewünschte Objektmodell zum Beispiel dem von Java [ArGo96, GoJo97] oder Delphi [Borl99] entspricht. Auch dort wird ausschließlich mit Referenzen auf Objekte gearbeitet. Die Arbeiten am Compiler, die zur Änderung des Objektmodells nötig sind, sollen hier nicht im Detail erläutert werden. Sie wurden bereits in der Frühphase der Diplomarbeit durchgeführt, um festzustellen, ob sich das Konzept der objektorientierten Laufzeitbibliothek überhaupt als tragfähig erweist.

3.5. Implementierung

Die Implementierung der Laufzeitbibliothek setzt genau die Ideen um, die bei der Vorstellung der objektorientierten Lösung angeführt wurden. Jede der benötigten Klassen wird in einer einzelnen Spezifikationsdatei realisiert. Die Dateien werden automatisch verarbeitet, bevor eine Spezifikation des Benutzers übersetzt wird. Um die Laufzeitbibliothek erweiterbar zu halten, wurde eine neue Datei `DoC.rtl` eingeführt, die quasi das objektorientierte Gegenstück zu `DoC.sys` ist. Jede Zeile dieser Datei enthält den Namen einer automatisch zu verarbeitenden Spezifikationsdatei. Neue Klassen, die Teil der Laufzeitbibliothek werden sollen, müssen nur in diese Datei eingetragen werden.

Abbildung 3.8 zeigt ein einfaches Klassendiagramm der Laufzeitbibliothek. Die folgenden Abschnitte beschreiben die einzelnen Klassen im Detail.

3.5.1. Klasse `Node`

Die Klasse `Node` ist die abstrakte Basis, von der alle anderen Klassen abgeleitet sind. Das gilt insbesondere für Klassen, die der Benutzer in einer Spezifikation deklariert. `Node` kapselt einen grundlegenden Knoten im Objektbaum, der eine beliebige Anzahl von Kindern verwalten kann. Die Klasse stellt zudem die Methode `save()` bereit, die vor dem Beenden des Programms aufgerufen wird, um die externe Zwischendarstellung zu erzeugen. Kapitel 4 beschäftigt sich sowohl mit dem Aufbau dieser Zwischendarstellung als auch der Implementierung von `save()`.

3.5.2. Klasse `DBUnit`

Die Klasse `DBUnit` ist die Basisklasse für Medienobjekte. Sie besitzt ein Attribut `file` sowie Methoden zum Zugriff auf dieses Attribut. Außerdem stellt die Klasse die beiden

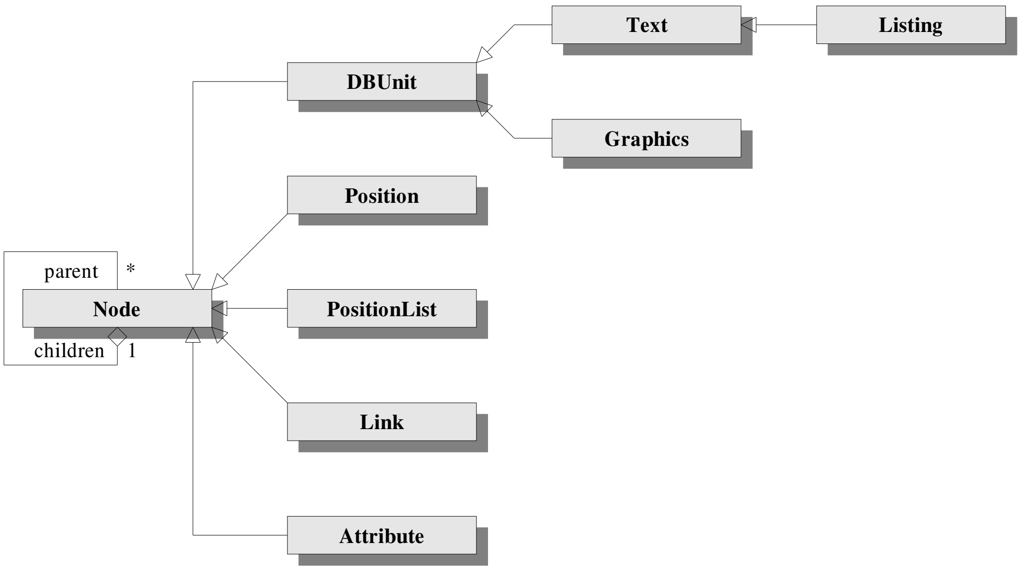

Abbildung 3.8.: Klassenstruktur der Laufzeitbibliothek

Methoden `getBegin()` und `getEnd()` bereit, welche die in Abschnitt 3.1 aufgeführten Funktionen `begin()` und `end()` implementieren. Es ist sichergestellt, daß jede der beiden Funktionen auch bei wiederholtem Aufruf immer das gleiche Positionsobjekt zurückgibt.

3.5.3. Klasse `Text`

Die Klasse `Text` ist ein spezialisierter Nachkomme von `DBUnit` zur Repräsentation von einfachem (Fließ-) Text. Die Klasse besitzt eine Methode `getOcc()`, die sämtliche Vorkommen einer gegebenen Zeichenkette in dem entsprechenden Medienobjekt ermittelt. `getOcc()` implementiert also die in Abschnitt 3.1 aufgeführte Funktion `occ()`.

3.5.4. Klasse `Listing`

Die Klasse `Listing` ist ein Nachkomme von `Text`, der für die Repräsentation von bereits formatiertem Text (speziell Quellcode) verwendet wird. `Listing` besitzt keine speziellen neuen Attribute oder Methoden. Die wesentliche Information verbirgt sich in diesem Fall im Typ selbst, also darin, daß zum Beispiel der HTML-Generator zwischen `Text` und `Listing` unterscheiden kann und für beide Typen eine unterschiedliche Ausgabe erzeugt.

3.5.5. Klasse `Graphics`

Die Klasse `Graphics` ist ein weiterer spezialisierter Nachkomme von `DBUnit`, der ein graphisches Medienobjekt repräsentiert. Auch hier steckt die wesentliche Information bereits im Typ des Medienobjektes.

3.5.6. Klasse `Position`

Die Klasse `Position` kapselt eine Position. Sie besitzt eine Methode `setLink()`, die einen Verweis zu einer anderen Position erzeugt. Außerdem besitzt `Position` ein Attribut `value` zur Aufnahme des Wertes der Position sowie Methoden zum Zugriff auf dieses Attribut.

`value` gibt den Punkt oder Ausschnitt des zur Position gehörenden Medienobjektes an, an dem die Position „liegt". Die genaue Interpretation dieses Attributes ist abhängig vom Typ des Medienobjektes. Für Medienobjekte des Typs `Text` ist dies zum Beispiel ein Wertepaar, das Anfang und Länge des Textausschnittes in Zeichen ausgehend vom Beginn der Datei definiert. Für Grafiken sind es vier Werte, die einen rechteckigen Ausschnitt festlegen. Zusätzlich werden für alle Medienobjekte die folgenden beiden ausgezeichneten Werte unterstützt:

- `begin` – legt fest, daß die Position den Anfang des Medienobjektes repräsentiert (also durch `getBegin()` entstanden ist).
- `end` – legt fest, daß die Position das Ende des Medienobjektes repräsentiert (also durch `getEnd()` entstanden ist).

Man könnte argumentieren, daß diese beiden Werte redundant sind, da sie sich – je nach Medienobjekt – auch anders formulieren lassen. Für Texte ließe sich der Anfang zum Beispiel als eine Position der Länge null ab dem nullten Zeichen beschreiben. Jedoch setzt dies in jedem Fall die genaue Kenntnis und Unterstützung des jeweiligen Medientyps im *DoDL*-Compiler wie auch im HTML-Generator voraus. Dies ist allein angesichts der Vielzahl von existierenden Grafikformaten eine Aufgabe, die zumindest den Rahmen dieser Arbeit bei weitem übersteigt. Bei komplexeren Medientypen – etwa Videos – existiert derzeit noch gar keine formale Definition des Begriffs *Ausschnitt* im Rahmen des *DoDL*-Projektes.

Um trotzdem eine grundsätzliche Unterstützung möglichst vieler Medientypen ohne übermäßigen Aufwand zu ermöglichen, werden Anfang und Ende eines Medienobjektes durch die Werte `begin` und `end` ausgedrückt. Damit ist es dem HTML-Generator möglich, auch ohne Kenntnis der internen Details eines Medientyps einen Verweis auf Anfang oder Ende eines „unbekannten" Medienobjektes zu erzeugen, indem er einen Anker unmittelbar vor oder hinter dem Medienobjekt in die Zieldatei einfügt.

3.5.7. Klasse `PositionList`

Die Klasse `PositionList` kapselt eine Liste von Positionen. Diese Klasse wurde als Hilfsklasse eingeführt, da `getOcc()` eine solche Liste von Positionen zurückgibt. `PositionList` besitzt keine besonderen Eigenschaften, stellt jedoch – wie `Position` – eine Methode `setLink()` bereit. Die Methode fügt jedem Element der Positionsliste den angegebenen Verweis hinzu, so daß dies letztlich die Implementierung der in Abschnitt 3.1 erwähnten Funktion `bindAll()` ist.

3.5.8. Klasse `Link`

Die Klasse `Link` kapselt einen Verweis. Ausgangspunkt des Verweises ist die Position, welcher der Verweis untergeordnet ist. Das Ziel kann über Methoden ermittelt und verändert werden. Zudem stellt die Klasse eine Methode `setAttribute()` bereit, die dem Verweis ein Attribut hinzufügt.

3.5.9. Klasse `Attribute`

Die Klasse `Attribute` kapselt ein (Feature-) Attribut. Sie besitzt zwei Attribute `name` und `value`, die Namen und Wert des Attributes aufnehmen, sowie Methoden zum Zugriff auf diese Attribute.

3.6. Beispiel

Abbildung 3.9 zeigt, wie die aus Kapitel 2 bekannte Beispiel-Spezifikation aussieht, wenn ihre `construct`-Sektion mit der neuen Laufzeitbibliothek implementiert wird. Der Objektbaum, der sich im ausführbaren Programm aus dieser Spezifikation ergibt, soll hier nicht komplett dargestellt werden, da er den Rahmen einer Seite sprengen würde und zur Hälfte bereits aus Abbildung 3.7 bekannt ist.

3.7. Alternativen

Es ist klar, daß die beiden vorgestellten Lösungen nicht die einzigen Möglichkeiten darstellen, eine Laufzeitbibliothek für das *DoDL*-System zu implementieren. Vielmehr sind es Extreme, die zur Veranschaulichung der unterschiedlichen grundsätzlichen Konzepte dienen. Zwischen den beiden Extremen sind zahlreiche Mischformen denkbar, die das eigentliche Ziel, die Erzeugung der Verknüpfungsstruktur eines Hyperdokumentes, bestimmt ebenso effektiv erreicht hätten. Allerdings besäße keine dieser Lösungen die Konsequenz und Flexibilität des durchgängig objektorientierten Ansatzes, weswegen diesem letztlich der Vorzug gegeben wurde.

Auch im Detail sind Alternativen möglich. Eine dieser Alternativen, die auch in der Anfangsphase der Diplomarbeit diskutiert wurde, ist die Verwendung einer bestehenden, externen Graphbibliothek zur Repräsentation der Verknüpfungsstruktur. Zu den Bibliotheken, die in diesem Zusammenhang in Betracht gezogen wurden, gehören die folgenden:

```
class Simple is Document with
  documents
    introTxt :  Text;
    figurePic : Graphics;
    samplePrg : Listing;
    hyperTxt :  Text;
    graphTxt :  Text;
    graphPic :  Graphics;

  construct
    void main(void)
    {
      hyperTxt.getOcc("Graphen").setLink(graphTxt.getBegin());
      introTxt.getOcc("Hyper").setLink(hyperTxt.getBegin());
      introTxt.getOcc("Graphen").setLink(graphTxt.getBegin());

      introTxt.getEnd().setLink(figurePic.getBegin());
      figurePic.getEnd().setLink(samplePrg.getBegin());
      graphTxt.getEnd().setLink(graphPic.getBegin()).
        setAttribute("Mathe").setValue("Nein");
    }

end Simple;
```

Abbildung 3.9.: Spezifikation des Beispiels

- Die *Stanford GraphBase* von Knuth [Knut93].
- Die *Library of Efficient Datatypes and Algorithms (LEDA)* [MeNa99].
- Die *Graph Template Library (GTL)* der Universität Passau [Pass99].
- Die Bibliothek *PLGraph*, die ebenfalls am Fachbereich Informatik der Universität Dortmund entwickelt wurde [FBI99b].

Jede dieser Bibliotheken bietet nicht nur eine ausgetestete und robuste Implementierung des eigentlichen Graphen, sondern auch zahlreiche Algorithmen, die auf diesem Graphen arbeiten können. Gerade letzteres sprach natürlich für die Verwendung einer fertigen Bibliothek, schien es doch so, daß dadurch der eigene Implementierungsaufwand in Grenzen gehalten werden könnte. Wie allerdings dieses Kapitel gezeigt haben sollte, werden für den Graphen, den ein *DoDL*-Programm konstruiert, innerhalb des *DoDL*-Programms keine besonderen Graphalgorithmen benötigt. Es müssen weder Spannbäume noch Zusammenhangskomponenten oder kürzeste Wege bestimmt werden. Der Graph muß, nachdem seine Konstruktion abgeschlossen ist, nur in die externe Zwischendarstellung überführt werden, womit sich Kapitel 4 beschäftigen wird.

Die vier genannten Beispiele haben zudem den großen Nachteil, daß sie die Installation von sehr umfangreichen C-Bibliotheken erfordern. Da die Compilierung des den Graphen erzeugenden Programms aber auf dem Rechner eines Benutzers von *DoDL* (also des Autors eines Hypertextes) geschieht, würde diesem Benutzer ebenfalls die Installation und Verwendung der Bibliotheken aufgezwungen, was nicht akzeptabel scheint. Speziell GTL ist zudem nur auf bestimmten Compilern und für wenige Plattformen übersetzbar, was den Einsatz von *DoDL* unnötig einschränken würde.

Die Verwendung einer Graphbibliothek hätte ohnehin nur bei der imperativen Lösung Sinn gemacht. Bei der objektorientierten Lösung wäre eine Kapselung der C-Strukturen der Graphbibliothek in *DoDL*-Klassen nötig geworden, um den konsequenten Charakter der Lösung nicht zu untergraben. Angesichts der tatsächlich benötigten Funktionalität wäre diese Kapselung wahrscheinlich ähnlich aufwendig ausgefallen wie die Implementierung der Klassen für den Graphen, so daß der Einsatz der Graphbibliotheken letztlich verworfen wurde.

3.8. Erweiterungen

Auf der Liste der möglichen Erweiterungen der Laufzeitbibliothek steht natürlich die Integration der Selektionsregeln auf einem der vorderen Ränge, ist es doch insbesondere die Berücksichtigung des Laufzeitverhaltens, die *DoDL* von anderen Ansätzen zur Konstruktion von Hyperdokumenten abhebt. Allerdings ist dies eine Erweiterung, die außer der Bibliothek auch alle anderen Komponenten des Systems betrifft und somit eine sehr grundsätzliche und entsprechend aufwendige Änderung darstellt.

Weniger weitreichende Auswirkungen haben solche Erweiterungen, die sich auf die Medienobjekte beziehen. Zum Beispiel können der Laufzeitbibliothek leicht neue Medientypen hinzugefügt werden, wie der Typ `Listing` gezeigt hat. Dies könnte ausgenutzt werden, um *DoDL* „multimedialer“ werden zu lassen, indem etwa Video- oder Tondokumente unterstützt werden. Allerdings sind alle zusätzlichen Medientypen, die nur `getBegin()` und `getEnd()` unterstützen, trivial, weil sie praktisch identisch zu `DBUnit` sind. Alternativ könnten jedoch die bestehenden Medientypen um neue Methoden ergänzt werden. Die derzeitige Implementierung von `getOcc()` ist sicherlich nicht die einzige Möglichkeit, Text in einer Datei zu suchen. Denkbar sind Varianten, die wahlweise mit oder ohne Berücksichtigung von Groß- und Kleinschreibung suchen oder sogar reguläre Ausdrücke unterstützen.

Schön wäre auch eine Erweiterung der Bibliothek um Methoden, die eine Manipulation der Verknüpfungsstruktur über das reine Erzeugen des attributierten Graphen hinaus ermöglichen. Damit könnten andere Arten von Spezifikationen erstellt werden, nämlich solche, die zunächst die Grundstruktur des Hyperdokumentes erzeugen und anschließend Änderungen dieser Struktur anhand bestimmter Kriterien vornehmen. Zum Beispiel könnte im Graphen nach allen Knoten (oder Medienobjekten) gesucht werden, die keinen Vorgänger besitzen, in dem Sinne, daß kein Verweis auf diesen Knoten existiert. Diese Knoten könnten automatisch in eine Art Inhaltsverzeichnis eingetragen werden, so ihre Erreichbarkeit gesichert ist, auch wenn der gesamte Strukturgraph nicht zusammenhängend ist.

4. Eine Zwischendarstellung für Hyperdokumente

Die zweite Teilaufgabe der Diplomarbeit behandelt den Entwurf der Zwischendarstellung, die das Hyperdokument abstrakt repräsentiert und die als Schnittstelle zwischen dem ausführbaren Programm und dem Ausgabewerkzeug für HTML dient. Dies umfaßt auch die Realisierung der Methode `save()`, die in Kapitel 3 zwar als Teil der Basisklasse `Node` vorgestellt, aber nicht implementiert wurde, um Vorgriffe auf das aktuelle Kapitel zu vermeiden.

4.1. Anforderungen

Wie bereits festgestellt, ist eine sorgfältige Spezifikation und Dokumentation der Zwischendarstellung von hoher Bedeutung, insbesondere wenn gewünscht ist, daß zu einem späteren Zeitpunkt auch andere Werkzeuge auf ihr aufbauen können. Die Zwischendarstellung soll im einzelnen den folgenden Anforderungen genügen:

- Sie soll für ein Programm sowohl leicht zu erzeugen als auch leicht zu lesen sein.
- Die Zwischendarstellung soll nicht in einer Binärdatei, sondern in einer Textdatei resultieren. Das erleichtert die Fehlersuche während der Testphase, da der Dateiinhalt auch von Menschen gelesen und verstanden werden kann. Außerdem können die Dateien leicht zwischen verschiedenen Plattformen ausgetauscht werden, was der Plattformunabhängigkeit von *DoDL* entgegenkommt.
- Sie soll eine möglichst strukturierte Beschreibung des Objektbaums ermöglichen, den die Laufzeitbibliothek während der Laufzeit des *DoDL*-Programms erzeugt. Diese Beschreibung sollte auch auf noch zu implementierende Eigenschaften von *DoDL*, etwa die Feature-Terme, vorbereitet sein bzw. leicht um diese ergänzt werden können.
- Die Zwischendarstellung soll – speziell im Hinblick auf die Erweiterbarkeit – so aufgebaut sein, daß es keinerlei Kompatibilitätsprobleme zwischen den verschiedenen Versionen des *DoDL*-Compilers und des HTML-Generators geben kann.

  Das schließt insbesondere die beiden folgenden Punkte mit ein:

  - Neue Elemente, die der *DoDL*-Compiler bzw. das ausführbare Programm erzeugt, sollen beim Einlesen der Zwischendarstellung vom Generator ignoriert werden können, wenn dieser die Elemente noch nicht kennt. Sie würden dann einfach „überlesen“.
  - Das Hinzufügen von neuen Elementen soll möglich sein, ohne daß dadurch die Wohlgeformtheit oder Korrektheit von bestehenden Dateien, die Zwischendarstellungen enthalten, negativ beeinträchtigt wird.

  Dadurch wäre es später zum Beispiel leicht möglich, zunächst den Compiler um die Feature-Terme zu erweitern und trotzdem für eine Übergangszeit den bestehenden HTML-Generator unverändert weiterzuverwenden.

- Es sollte sich – wenn möglich – um ein bestehendes Dateiformat handeln, oder die Lösung sollte sich an ein solches Format anlehnen. Dies ermöglicht eventuell die Verwendung von existierendem Quellcode innerhalb der Diplomarbeit und spart so Implementierungsaufwand.

Wie an der Liste der Anforderungen leicht zu erkennen ist, verlangt die Zwischendarstellung für Hyperdokumente nicht nach einem simplen Dateiformat. Insbesondere die Anforderungen nach Strukturiertheit und Erweiterbarkeit lassen vermuten, daß eher eine spezielle *Beschreibungssprache* gesucht wird, was auch der Fall ist.

4.2. Mögliche Kandidaten

Eine Sprache bzw. Sprachfamilie, die die genannen Anforderungen erfüllt, ist die *Standard Generalized Markup Language* (SGML) [Szil94]. Es wäre also möglich, die Zwischendarstellung als SGML-Applikation zu entwerfen, womit man sich gleichzeitig auf einen anerkannten Standard zurückziehen würde. Leider ist die Verwendung von SGML nicht ganz unkompliziert, unter anderem deshalb, weil Sprachdefinitionen in SGML für den Leser oft nur schwer zu durchschauen sind.

Es existiert jedoch eine einfachere Sprachfamilie, deren Ausdrucksmöglichkeit zwar nicht die von SGML erreicht, sich jedoch bereits in vielen Fällen als ausreichend erwiesen hat. Es handelt sich dabei um die Familie der Sprachen, die auf der Basis der *Extensible Markup Language* (XML) [W3C98a, BeMi98] entworfen wurden. XML ist eine Untermenge von SGML und wird seit 1996 vom *World Wide Web Consortium* (W3C) als einfache, erweiterbare Auszeichnungssprache für Internet-Anwendungen entwickelt.

Anfang 1998 wurde die Sprache in der Version 1.0 vorgestellt. XML hat seitdem auch in anderen Bereichen eine hohe Verbreitung gefunden. Als ein Beispiel von vielen sei hier die Verwendung als Kommunikationssprache zwischen den Datenbank-Komponenten des COMRIS-Projektes [FBI99a] genannt, an dem der Lehrstuhl für Künstliche Intelligenz der Universität Dortmund beteiligt ist. Inzwischen existiert auch ein Entwurf für eine

```
1  <!-- Anfang der Datei-->
2
3  <?xml version="1.0"?>
4
5  <grusskarte adresse="joerg.pleumann@trantor.de">
6    <bild file="feuerwerk.gif"/>
7    <text>
8      Ein fr&ouml;hliches Jahr 2000!
9    </text>
10 </grusskarte>
11
12 <!-- Ende der Datei-->
```

Abbildung 4.1.: Einfache XML-Datei

Neufassung von HTML auf der Basis von XML, die sogenannte *Extensible HyperText Markup Language* (XHTML) [W3C99b], die nach den Plänen des W3C in absehbarer Zeit das klassische HTML als Beschreibungssprache für das WWW ablösen soll.

Der einfache Aufbau von XML macht insbesondere das Schreiben eines Parsers zum Einlesen einer XML-Datei zu einer trivialen Aufgabe. Es existieren auch bereits zahlreiche Bibliotheken, die XML-Parser für die verschiedensten Programmiersprachen bereitstellen. Nicht zuletzt dies gab den Ausschlag für die Entscheidung, XML als Basis der Zwischendarstellung für Hyperdokumente innerhalb des *DoDL*-Systems zu verwenden.

4.3. Eine kurze Einführung in XML

Abbildung 4.1 zeigt eine XML-Datei, die einen kleinen elektronischen Neujahrsgruß realisiert. Die Datei ist sehr einfach gehalten, aber es lassen sich an ihr alle wesentlichen Merkmale von XML ablesen:

- Zunächst einmal enthält die XML-Datei *Zeichendaten*, die den eigentlichen textuellen Inhalt der Datei festlegen. Für solche Zeichen, die nicht dem ASCII-Zeichensatz entstammen (wie etwa die deutschen Umlaute), oder solche, die in XML eine spezielle Bedeutung haben (wie die spitzen Klammern), existieren Umschreibungen. Diese werden durch ein `&` eingeleitet. Zeile 6 des Beispiels enthält sowohl Zeichendaten („`Ein fr`“ und „`hliches Jahr 2000!`“) als auch eine Umschreibung für das kleine „ö“ („`ö`“)[1].
- Auffälligstes Merkmal von XML sind die *Elemente*, die der XML-Datei Struktur geben. Zur Markierung der Elemente werden in XML (wie in HTML) spitze Klammern verwendet, die den Namen des Elementes umschließen. Ein Element setzt

[1] Das Beispiel ist etwas ungünstig gewählt, weil viele Plattformen heute mit der Zeichencodierung *ISO-Latin-1* arbeiten. Dort können die Umlaute auch direkt eingegeben werden. Auf dem OS/2-System des Autors ist das jedoch nicht der Fall.

```
<!-- Anfang der Datei-->

<?xml version="1.0"?>

<!ELEMENT grusskarte (text|bild)*>
<!ATTLIST grusskarte adresse CDATA #REQUIRED>

<!ELEMENT text (#PCDATA)>

<!ELEMENT bild EMPTY>
<!ATTLIST bild file CDATA #REQUIRED>

<!-- Ende der Datei-->
```

Abbildung 4.2.: Einfache XML-DTD

sich entweder aus einem Paar von Start- und Endmarkierungen zusammen (zum Beispiel <text> und </text> in den Zeilen 7 und 9) oder es besteht nur aus einer sogenannten *leeren* Markierung (wie etwa <bild/> in Zeile 6). In einer Markierung können zudem Attribute für das Element festgelegt werden, wovon die Zeilen 5 und 6 Gebrauch machen.

- Es ist möglich, eine XML-Datei mit *Kommentaren* zu versehen, die naheliegenderweise nicht zum Inhalt der Datei zählen und dementsprechend auch in einer Anwendung nicht angezeigt werden. Die Zeilen 1 und 12 enthalten jeweils einen Kommentar.

- Außerdem können in einer XML-Datei *Instruktionen* angegeben werden, die sich an das Programm richten, das die Datei verarbeitet. Im allgemeinen geben diese Instruktionen Hinweise zum Umgang mit der Datei. Sie können, müssen aber nicht beachtet werden. Üblicherweise beginnt jede XML-Datei mit einer Instruktion, die Auskunft über die XML-Version gibt, auf welcher die Datei basiert. Im Beispiel enthält Zeile 3 diese Information.

Von diesen vier Merkmalen sind es natürlich die Elemente, die das Salz in der Suppe von XML ausmachen, denn sie erlauben eine Strukturierung der Datei nach nahezu beliebigen Gesichtspunkten. Zwischen einem Paar aus Start- und Endmarkierung kann sich nicht nur reiner Text befinden, sondern – wie das Beispiel zeigt – auch andere Elemente. Es ist also möglich, Elemente zu schachteln, wodurch die XML-Datei eine baumartige Struktur erhält. Dabei ist selbstverständlich zu beachten, daß die Schachtelung korrekt durchgeführt wird, daß also zusammengehörige Start- und Endmarkierungen auch dem gleichen Element untergeordnet sind. Der XML-Standard fordert zudem, daß jedes XML-Dokument ein einzelnes äußeres Element besitzt, das als Wurzel des Dokumentbaums dient. Im Beispiel ist dies das Element, das sich aus dem Markierungen <grusskarte> und </grusskarte> zusammensetzt.

Die Schachtelung der XML-Elemente kann im Prinzip völlig frei durchgeführt werden. Es ist jedoch auch möglich, Regeln für den Aufbau einer konkreten XML-Datei mit anzugeben, die als *Document Type Description (DTD)* bezeichnet werden. Eine DTD ist im Prinzip eine in spezieller Notation abgefaßte kontextfreie Grammatik, die festlegt, welches Element an welcher Stelle einer XML-Datei erlaubt ist und welche Attribute es besitzen darf. Die DTD gibt also erlaubte Produktionen vor. Abbildung 4.2 zeigt eine solche DTD für elektronische Grußkarten, allerdings sollen die Details an dieser Stelle keine Berücksichtigung finden. Wie leicht zu erkennen ist, handelt es sich bei einer DTD um ein nicht sonderlich ästhetisches Konstrukt.

Zwei Begriffe sind im Zusammenhang mit XML-Dateien und DTDs wichtig:

- Eine Datei, die die allgemeinen Syntaxregeln von XML erfüllt, wird als *wohlgeformt* bezeichnet.
- Eine Datei, die zudem die Produktionsregeln einer vorgegebenen DTD erfüllt, heißt *gültig* bezüglich dieser DTD.

Die Beispieldatei aus Abbildung 4.1 ist sowohl wohlgeformt als auch gültig bezüglich der DTD aus Abbildung 4.2. Der Beweis für letzteres soll zwar an dieser Stelle nicht erbracht werden, aber die Möglichkeit, diesen Beweis zu erbringen, ist durchaus von Bedeutung: Es ist eine übliche Vorgehensweise, für einen speziellen Anwendungsfall von XML auch eine angepaßte DTD zu entwerfen und so die Menge der möglichen Dokumente auf die für den Anwendungsfall sinnvollen einzuschränken. Zum Beispiel geschieht die bereits erwähnte Neufassung von HTML in XML mit Hilfe einer solchen DTD. Wird, um ein weiteres Beispiel zu nennen, XML als Dateiformat für eine Datenbank-Anwendung benutzt, dann drückt die DTD die Struktur dieser Datenbank aus. Der Vorteil dieser Vorgehensweise liegt darin, daß ein XML-Parser neben der Wohlgeformtheit auch die Gültigkeit einer Datei automatisch überprüfen kann, wenn ihm die DTD bekannt ist. Der XML-Parser erbringt also automatisch den oben angesprochenen Beweis. Die eigentliche Anwendung bzw. deren Entwickler kann davon ausgehen, daß eine XML-Datei die gewünschte Struktur besitzt, wenn sie die syntaktische Analyse erfolgreich besteht.

4.4. Entwurf einer eigenen Sprache

Mit den einführenden Informationen aus dem letzten Abschnitt ist das nötige Rüstzeug vorhanden, um eine eigene XML-basierte Sprache zu entwerfen, die die Anforderungen aus Abschnitt 4.1 erfüllt. Die Frage ist nun, welche XML-Elemente benötigt werden, wie sie zueinander in Beziehung stehen und welche Attribute sie besitzen sollen.

Die benötigten Elemente ergeben sich aus den Klassen der Laufzeitbibliothek, die zum Erzeugen des Hyperdokumentes wichtige Informationen beinhalten. Das sind alle Klassen, die Medienobjekte repräsentieren, sowie die Klassen `Position`, `Link` und `Attribute`, aus denen sich der attributierte Graph der Verknüpfungsstruktur zusammensetzt. Die Klassen des Benutzers werden zum Erzeugen des Hyperdokumentes zu diesem Zeitpunkt nicht mehr benötigt. Sie waren nur zur Bildung und Auswertung der Spezifikation von Bedeutung und tauchen somit in der Zwischendarstellung nicht mehr auf.

Benötigt werden also im einzelnen die folgenden Elemente:

- Ein Element `<dbunit>` zur Repräsentation der Medienobjekte.
- Ein Element `<position>` zur Repräsentation der Positionen.
- Ein Element `<link>` zur Repräsentation der Verweise.
- Ein Element `<attribute>` zur Repräsentation der (Feature-) Attribute.

Wie aus Abschnitt 4.3 hervorgeht, wird außerdem ein eindeutiges Element benötigt, das die Wurzel der Dateistruktur bildet. Dieses erhält naheliegenderweise den Namen `<document>`. Dem Element könnten die vier Elemente `<dbunit>`, `<position>`, `<link>` und `<attribute>` theoretisch als direkte Unterelemente zugeordnet werden. Um aber die von *DoDL* propagierte Trennung von Inhalt, Struktur und Verhalten auch in der XML-Datei aufrechtzuerhalten, werden zunächst noch die weiteren Elemente `<content>`, `<structure>` und `<browsing>` als Unterelemente von `<document>` definiert. Diese spannen drei feste Hauptäste im Elementbaum der XML-Datei auf, in welche die eigentlich informationstragenden Elemente `<dbunit>`, `<position>`, `<link>` und `<attribute>` entsprechend einsortiert werden: `<dbunit>` wird Unterelement von `<content>`, `<position>` wird Unterlement von `<structure>`, und `<attribute>` wird zu einem Unterelement von `<browsing>`. Wegen der engen Beziehung zwischen Positionen und Verweisen wird `<link>` Unterelement von `<position>` – jeder Verweis wird der Position untergeordnet, an der er beginnt.

4.5. Implementierung der Sprache

Die folgenden Abschnitte geben die „Implementierung“ der entworfenen Beschreibungssprache im Detail wieder. Sie sind gleichzeitig als Referenz aller Elemente und Attribute der Sprache zu betrachten, wobei ihre Reihenfolge einem *Top-Down*-Durchlauf der Hierarchie entspricht, die zwischen den Elementen besteht. Eine komplette DTD der Sprache befindet sich in Anhang A.

4.5.1. Element `<document>`

Das Element `<document>` ist das äußerste Element einer jeden Zwischendarstellung. Es leitet die Datei ein und schließt sie ab. Stellt man sich die verschachtelte Struktur der einzelnen Elemente als Baum vor, dann ist `<document>` die Wurzel dieses Baums. Das gilt auch für den Syntaxbaum, der aus der syntaktischen Analyse einer Zwischendarstellung hervorgeht. Das folgende Beispiel zeigt die Verwendung des Elementes:

```
<document spec="simple.spec" bind="simple.bind"
          user="Pleumann"    date="09-Jan-00">
  <content>
    ...
  </content>
  <structure>
    ...
  </structure>
  <browsing>
    ...
  </browsing>
</document>
```

Wie aus dem Beispiel hervorgeht, besitzt das Element vier Attribute:

- `spec` – gibt den Namen der Spezifikationsdatei an, aus welcher die Zwischendarstellung hervorgegangen ist.
- `bind` – gibt den Namen der Bindungsdatei an, über welche die freien Variablen der Spezifikation an Medienobjekte gebunden wurden.
- `user` – gibt den Namen des Benutzers an, der die ausführbare Spezifikation aufgerufen und so die Zwischendarstellung erzeugt hat.
- `date` – gibt den Zeitpunkt an, zu dem die Zwischendarstellung erzeugt wurde.

Die Zwischendarstellung aus dem Beispiel wurde also durch ein *DoDL*-Programm erzeugt, das am 9. Januar 2000 von einem Benutzer namens „Pleumann“ ausgeführt wurde. Das Programm ging aus der Spezifikation `simple.spec` und den Bindungen `simple.bind` hervor. Zugegebenermaßen ist keines der vier Attribute für den Generator oder für die Erzeugung von HTML-Code unbedingt erforderlich. Im praktischen Einsatz kann es aber durchaus interessant oder hilfreich sein, solche Informationen zu verwalten und mit in das endgültige HTML-Dokument aufzunehmen. Unter anderem ermöglicht es eine sehr einfache – weil manuelle – Form der Versionskontrolle für Hyperdokumente, die mit dem *DoDL*-System erzeugt werden.

Innerhalb von `<document>` werden die drei Unterelemente `<content>`, `<structure>` und `<browsing>` erwartet. Jedes der Elemente muß genau einmal vorhanden sein, und auch die Reihenfolge ist vorgegeben. Daraus folgt, daß die in den Elementen enthaltenen Informationen sich nach der syntaktischen Analyse auf drei feste Hauptäste des Syntaxbaumes aufteilen und nicht über diesen verteilt sind. Das erleichtert die Implementierung sowohl des Parsers für die Beschreibungssprache als auch den HTML-Generators.

4.5.2. Element <content>

Das Element <content> umschließt den Teil einer Zwischendarstellung, in welchem der Inhalt des Hyperdokumentes festgelegt wird. Es besitzt keine Attribute. Das Element enthält die Information aller documents-Sektionen einer Spezifikation und der zugehörigen Bindungen. Innerhalb von <content> darf das Element <dbunit> beliebig oft vorkommen. Andere Elemente sind nicht erlaubt.

4.5.3. Element <structure>

Das Element <structure> umschließt den Teil einer Zwischendarstellung, in welchem die Verknüpfungsstruktur des Hyperdokumentes festgelegt wird. Es besitzt keine Attribute. Sein Inhalt ergibt sich aus der Auswertung der construct-Sektionen einer Spezifikation. Innerhalb von <structure> darf das Element <position> beliebig oft vorkommen. Andere Elemente sind nicht erlaubt.

4.5.4. Element <browsing>

Das Element <browsing> umschließt den Teil einer Zwischendarstellung, in welchem das Laufzeitverhalten des Hyperdokumentes festgelegt wird. Es besitzt keine Attribute. Innerhalb von <browsing> darf das Element <attribute> beliebig oft vorkommen. Andere Elemente sind nicht erlaubt. Damit ergibt sich die Information, die in <browsing> enthalten ist, ebenfalls aus der Auswertung der constructs-Sektionen einer Spezifikation, weil dort die Verweise attributiert werden.

Vermutlich wird es zu einem späteren Zeitpunkt ein weiteres untergeordnetes Element <rule> oder <selection> geben, mit dem eine Selektionsregel für Verweise notiert wird. Die Implementierung der Feature-Terme und der Selektionsregeln ist jedoch nicht Bestandteil dieser Arbeit.

4.5.5. Element <dbunit>

Das Element <dbunit> nimmt ein Medienobjekt des Hyperdokumentes auf, wobei zusätzlich Dateiname und Typ des Medienobjektes festgelegt werden müssen. Ein kleines Beispiel soll die Verwendung von <dbunit> für die drei Medienobjekte zeigen, die in der rechten Hälfte von Abbildung 2.1 enthalten sind:

```
<dbunit id="doc1" file="hyper.txt" type="Text">
</dbunit>
<dbunit id="doc2" file="graph.txt" type="Text">
</dbunit>
<dbunit id="doc3" file="graph.gif" type="Graphics">
</dbunit>
```

Wie aus dem Beispiel hervorgeht, besitzt das Element die folgenden Attribute, die alle erforderlich sind:

- `id` – gibt die innerhalb der Datei eindeutige Kennung des Medienobjektes an. Die Elemente, die zur Bildung der Struktur verwendet werden, verweisen auf Medienobjekte anhand dieser Kennung.
- `file` – gibt den Dateinamen des Medienobjektes an.
- `type` – gibt den Typ des Medienobjektes an. Unterstützt werden derzeit die drei Typen `Text`, `Graphics` und `Listing`.

Innerhalb von `<dbunit>` sind keine anderen Elemente erlaubt.

4.5.6. Element `<position>`

Das Element `<position>` dient zur Speicherung einer Position des Hyperdokumentes. Positionen werden als Anfangs- und Endpunkte für Verweise benötigt. Auch bei diesem Element soll ein kleines Beispiel die Verwendung demonstrieren. Es ergibt sich ebenfalls aus der rechten Hälfte von Abbildung 2.1:

```
<position id="pos1" ref="doc1" value="79,7">
  <link>
     ...
  </link>
</position>

<position id="pos2" ref="doc2" value="begin">
</position>

<position id="pos3" ref="doc2" value="end">
  <link>
     ...
  </link>
</position>

<position id="pos4" ref="doc3" value="begin">
</position>
```

Wie aus dem Beispiel hervorgeht, besitzt das Element die folgenden Attribute:

- `id` – gibt die innerhalb der Datei eindeutige Kennung der Position an. Verweise bauen auf Positionen auf und referenzieren diese anhand der Kennung.
- `ref` – gibt die Kennung des Medienobjektes an, auf das sich die Position bezieht. Dabei handelt es sich um das Medienobjekt, in dem die Position beheimatet ist.

- `value` – gibt den Punkt oder Ausschnitt des durch `ref` spezifizierten Medienobjektes an, an dem die Position liegt. Die genaue Interpretation dieses Attributes ist abhängig vom Typ des Medienobjektes (vergleiche dazu Abschnitt 3.5.6).

Innerhalb von `<position>` sind beliebig viele Vorkommen des Elements `<link>` erlaubt. Es sei an dieser Stelle erwähnt, daß die Beschreibungssprache damit in der Lage ist, unbegrenzt viele Verweise zu verwalten, die von einer Position ausgehen. Auch die Erweiterung des Systems um bidirektionale Verweise ist sehr einfach möglich, wenn z.B. ein neues Attribut `direction` mit den Werten `in`, `out` und `inout` eingeführt wird.

4.5.7. Element `<link>`

Das Element `<link>` fügt der Zwischendarstellung des Hyperdokumentes einen Verweis hinzu. Anfangspunkt des Verweises ist die Position, innerhalb derer das Element aufgeführt ist. Das folgende Beispiel, das wieder auf den Inhalt von Abbildung 2.1 zurückgreift, verdeutlicht die Verwendung von `<link>`:

```
<position id="pos1" ref="doc1" value="79,7">
  <link id="lnk1" target="pos2">
    ...
  </link>
</position>

<position id="pos2" ref="doc2" value="begin">
</position>

<position id="pos3" ref="doc2" value="end">
  <link id="lnk2" target="pos4">
    ...
  </link>
</position>

<position id="pos4" ref="doc3" value="begin">
</position>
```

Wie aus dem Beispiel hervorgeht, besitzt das Element die folgenden Attribute:

- `id` – gibt die innerhalb der Datei eindeutige Kennung des Verweises an. Attribute referenzieren Verweise anhand dieser Kennung.

- `target` – gibt die Kennung der Position an, die Ziel des Verweises ist.

Innerhalb vom `<link>` sind keine anderen Elemente erlaubt.

4.5.8. Element `<attribute>`

Das Element `<attribute>` fügt einem Verweis des Hyperdokumentes ein aus einem Namen und einem Wert bestehendes Attribut hinzu. Name und Wert sollten den üblichen Regeln für Bezeichner in Programmiersprachen folgen, sich also aus Buchstaben und Ziffern zusammensetzen und mit einem Buchstaben beginnen. Attribute können von einem geeigneten Laufzeitsystem genutzt werden, um zu ermitteln, welche Verweise für einen Benutzer sichtbar sind und welche nicht. Momentan wird die Information einfach an den HTML-Generator weitergereicht.

Auch an dieser Stelle soll ein Beispiel, das sich auf Abbildung 2.1 bezieht, die – zugegebenermaßen triviale – Verwendung des Elementes demonstrieren:

```
<attribute ref="lnk1" name="Mathe" value="Nein">
</attribute>
```

Der Vollständigkeit halber seien auch hier die drei (XML-) Attribute des Elementes `<attribute>` noch einmal genannt:

- `ref` – gibt die eindeutige Kennung des Verweises an, den das Attribut dekoriert.
- `name` – gibt den Namen des Attributs an.
- `value` – gibt den Wert des Attributs an.

Innerhalb von `<attribute>` sind keine anderen Elemente erlaubt.

4.6. Implementierung von `save()`

Die Implementierung der Methode `save()` der Klasse `Node` schreibt zunächst den allgemeinen Rahmen der XML-Datei, also das Wurzelelement `<document>` inklusive seiner drei Unterelemente `<content>`, `<structure>` und `<browsing>`. Innerhalb dieser drei Elemente ruft `save()` jeweils eine weitere Methode auf (`writeContent()`, `writeStructure()` oder `writeBrowsing()`). Jede dieser Methoden führt einen kompletten Tiefendurchlauf des Objektbaums durch, indem sie sich selbst rekursiv für alle Kinder eines Objektes aufruft. Dadurch wird jedes Objekt der gesamten Datenstruktur insgesamt dreimal besucht.

Eine für die Erzeugung der XML-Datei benötigte Klasse überschreibt nun eine dieser drei Methoden, um ihre Informationen in einen der drei Hauptäste der XML-Datei „einzubringen". Die Klasse `DBUnit` überschreibt `writeContent()`, und ihre spezialisierten Nachkommen `Text`, `Graphics` und `Listing` erben diese Implementierung. Die Klassen `Position` und `Link` überschreiben `writeStructure()`, und für `Attribute` wird `writeBrowsing()` überschrieben.

Der Vorteil dieser Vorgehensweise ist offensichtlich: Wenn dem System neue Klassen hinzugefügt werden, zum Beispiel die Selektionsregeln für das Laufzeitverhalten, dann ist deren Implementierung komplett innerhalb der neuen Klassen möglich. Es ist insbesondere nicht notwendig, die Methode `save()` anzupassen, da diese nur für den Tiefendurchlauf und den festen Rahmen, nicht aber für das Erzeugen der XML-Elemente einzelner Objekte zuständig ist. Das Gesamtsystem ist also auch an dieser Stelle leicht erweiterbar.

4.7. Beispiel

Wie sieht die komplette XML-Datei aus, die sich aus Abbildung 2.1 ergibt? Um dies herauszufinden, müssen einfach die einzelnen Beispiele der letzten Abschnitte zu einer Datei zusammengesetzt werden, wie es in Abbildung 4.3 geschehen ist. Auch diese Abbildung beschränkt sich wieder auf die rechte Hälfte des bekannten Hyperdokumentes, um den Rahmen einer Seite nicht zu sprengen.

4.8. Alternativen

Die entworfene XML-basierte Sprache ist nicht die einzige Möglichkeit, die Zwischendarstellung des Hyperdokumentes zu realisieren. Alternativen liegen jedoch eher im Detail, bewegen sich also in gewisser Weise „innerhalb“ dieser Lösung, denn komplett andere Ansätze sind ja schon im Vorfeld als entweder unzureichend (einfache Dateiformate) oder zu kompliziert (SGML) ausgeschieden.

Auch macht es keinen Sinn, an dieser Stelle auf die Beschreibungssprachen anderer Hypermedia-Systeme, etwa *HyTime* oder *HyperWave*, zurückzugreifen, da die Zwischendarstellung dann offenbar nicht mehr abstrakt wäre, sondern bereits ein bestimmtes Zielformat bevorzugen würde. Sinnvoller wäre es, diese Systeme neben HTML als Zielformate des *Generators* zu erlauben.

Die Zwischendarstellung besitzt allerdings eine gewisse Verwandtschaft zu dem, was in [HaSc94] als einfaches Austauschformat für Hyperdokumente vorgeschlagen wird, das sich aus dem Dexter-Modell ergibt. Jedoch ist dieses Austauschformat weder komplett ausgearbeitet worden, noch ist die Trennung zwischen Struktur und Inhalt dort ähnlich konsequent wie in der hier entworfenen Zwischendarstellung.

Auch wenn man sich auf eine Implementierung der Zwischendarstellung als XML-Applikation festlegt, ist die vorgestellte Lösung nicht die einzig mögliche. Die Sprache enthält zum Beispiel nur Elemente, die sich aus einer Start- und einer Endmarkierung zusammensetzen (etwa `<dbunit>` und `</dbunit>`), obwohl in vielen Fällen auch leere XML-Elemente möglich wären (also etwa `<dbunit/>`). Dies würde sowohl eine Vereinfachung der Sprachstruktur als auch eine Verkleinerung der Dateien bedeuten, die sich der Sprache bedienen.

```
<?xml version="1.0"?>
<document spec="simple.spec" bind="simple.bind"
          name="Pleumann"    date="09-Jan-00">
  <content>
    <dbunit id="doc1" file="hyper.txt" type="Text">
    <dbunit>
    <dbunit id="doc2" file="graph.txt" type="Text">
    <dbunit>
    <dbunit id="doc3" file="graph.gif" type="Graphics">
    <dbunit>
  </content>

  <structure>
    <position id="pos1" ref="doc1" value="79,7">
      <link id="lnk1" target="pos2">
      </link>
    </position>

    <position id="pos2" ref="doc2" value="begin">
    </position>

    <position id="pos3" ref="doc2" value="end">
      <link id="lnk2" target="pos4">
      </link>
    </position>

    <position id="pos4" ref="doc3" value="begin">
    </position>
  </structure>

  <browsing>
    <attribute ref="lnk1" name="Mathe" value="Nein">
    </attribute>
  </browsing>

</document>
```

Abbildung 4.3.: Komplette XML-Datei für das Beispiel-Dokument

Die ausschließliche Verwendung der Start- und Endmarkierungen wurde jedoch bewußt gewählt, weil sie einen entscheidenden Vorteil hat: Sie gewährleistet eine Erweiterbarkeit der Sprache, ohne daß Kompatibilitätsprobleme entstehen können. Leere XML-Elemente können keine untergeordneten Elemente besitzen. Sollte also z.B. ein leeres Element `<dbunit/>` aus irgendeinem Grund um untergeordnete Elemente erweitert werden, dann müßte es zunächst in ein aus Start- und Endmarkierung bestehendes Element umgewandelt werden. Dies wäre allerdings eine Änderung der Sprache, die zuvor gültige Zwischendarstellungen ungültig werden läßt, was nicht akzeptabel scheint.

Auch könnte man argumentieren, daß die Struktur einer Zwischendarstellung unnötig „aufgebläht“ erscheint. Eine alternative Implementierung könnte `<position>` als Unterelement von `<dbunit>` und `<attribute>` als Unterelement von `<link>` verwenden. Die Notwendigkeit für die `ref`-Attribute würde damit entfallen, weil unmittelbar klar wäre, auf welches Medienobjekt bzw. auf welchen Verweis sich Position und Attribut beziehen. Die vorgestellte Implementierung hat jedoch den Vorteil, daß Inhalt, Struktur und Verhalten des beschriebenen Hyperdokumentes nach wie vor voneinander getrennt sind, was der ursprünglichen Philosophie von *DoDL* entspricht.

4.9. Erweiterungen

Die Sprache wurde mit dem Ziel entworfen, erweiterbar zu sein, und bei der Vorstellung der einzelnen Elemente kam auch bereits die eine oder andere mögliche Erweiterung zur Sprache – etwa die bidirektionalen Verweise. Welche anderen Erweiterungen sind denkbar?

Theoretisch wäre es zum Beispiel möglich, den kompletten Inhalt eines Medienobjektes zwischen den zugehörigen Markierungen `<dbunit>` und `</dbunit>` in den inhaltlichen Teil der Zwischendarstellung aufzunehmen. Das hätte den Vorteil, daß die Zwischendarstellung alle zum Erzeugen des Hyperdokumentes benötigten Daten enthielte. Der HTML-Generator müßte nicht mehr auf die Datenbank mit den Medienobjekten zugreifen, und die Zwischendarstellung könnte in einer einzigen Datei weitergegeben werden. Da Inhalt, Struktur und Verhalten sich dann zwar in der gleichen Datei befinden, aber dort in verschiedenen Bereichen untergebracht sind, ist der Grundsatz des *separation of concerns* weiterhin gesichert.

Textuelle Medienobjekte könnten in diesem Schritt in eine Standard-Codierung überführt werden, zum Beispiel in eine ISO-Zeichentabelle oder Unicode, wodurch die Implementierung des HTML-Generators erheblich vereinfacht würde. Es wäre allerdings zu überlegen, ob eine textuelle Codierung anderer Medienobjekte, etwa der Grafiken, effizient möglich und sinnvoll ist.

Eine andere interessante Erweiterung wäre die, ein Hyperdokument inkrementell zu erzeugen. Man stelle sich dazu ein komplexes Hyperdokument vor, das sich aus vielen hundert Medienobjekten zusammensetzt, die zudem *sehr groß* sind. Es sei weiterhin angenommen, daß Verweise vorwiegend über das Suchen von Zeichenketten in textuellen

Medienobjekten (also über die Funktion `getOcc()`) gebildet werden. Das Erzeugen der Verknüpfungsstruktur eines solchen Hyperdokumentes nimmt möglicherweise so viel Zeit in Anspruch, daß es wünschenswert wäre, nicht immer den kompletten Graphen erzeugen zu müssen, also die gesamte Spezifikation auszuwerten, wenn sich eines der Medienobjekte ändert.

Die Idee ähnelt der des inkrementellen Compilierens. Sie ist realisierbar, setzt jedoch voraus, daß ein *DoDL*-Programm in der Lage ist, seine eigene XML-strukturierte Ausgabe zu lesen und Änderungen daran vorzunehmen. Ist dieser Schritt geschafft, dann ist es ein leichtes herauszufinden, welche Medienobjekte sich seit der letzten Ausführung der Spezifikation geändert haben und welche Positionen nun nicht mehr gültig sind, also neu berechnet werden müssen. Der Rest könnte unverändert übernommen werden, was in dem geschilderten Szenario zu erheblichen Laufzeitersparnissen führen dürfte.

5. Ein Ausgabewerkzeug für HTML

Die dritte Teilaufgabe der Diplomarbeit umfaßt den Entwurf und die Implementierung des Ausgabewerkzeugs für HTML, des *Generators*. Dieser Generator ist der eigentliche Kernpunkt der Arbeit und gleichzeitig auch der letzte Schritt, den ein Hyperdokument auf seinem Weg durch das *DoDL*-System zu gehen hat.

Vergleicht man das gesamte *DoDL*-System mit einem Compiler, dann ist der Generator das *back-end* des Compilers und somit verantwortlich für das Erzeugen des maschinenspezifischen Zielcodes. Der Zielcode ist in diesem Fall HTML, die Zielmaschine entsprechend ein beliebiger HTML-Browser. Ein Schwerpunkt des Entwurfes lag von Anfang an darauf, dieses *back-end* möglichst austauschbar zu gestalten, so daß mit minimalem Aufwand Zielcode für andere Hypermedia-Systeme erzeugt werden kann.

5.1. Anforderungen

Die grundlegenden Anforderungen an den Generator wurden bereits in der Systemübersicht in Kapitel 2.6 vorgestellt. Sie lassen sich in zwei Sätzen zusammenfassen:

- Der Generator liest die Zwischendarstellung, die von der ausführbaren Spezifikation erzeugt wurde.
- Er traversiert diese Datenstruktur und erzeugt unter Zuhilfenahme der beteiligten Medienobjekte die Ausgabe in Form einer oder mehrerer HTML-Seiten.

Diese sicherlich nicht sehr detaillierte Beschreibung soll zunächst etwas verfeinert werden, bevor in den folgenden Abschnitten die Kernpunkte der Implementierung beleuchtet werden. Tatsächlich führt der Generator vier wesentliche Schritte durch, bis das Endergebnis in Form von HTML-Seiten auf der Festplatte vorliegt:

1. Zunächst wird der Inhalt der Zwischendarstellung gelesen. Hierzu ist offensichtlich ein *Parser* notwendig, der die XML-basierte Sprache aus Kapitel 4 „versteht“. Während der syntaktischen Analyse wird eine *Repräsentation* des Strukturgraphen bzw. des gesamten Hyperdokumentes im Speicher aufgebaut. Abschnitt 5.3 stellt die Klassen vor, aus denen sich diese Repräsentation zusammensetzt. Abschnitt 5.4 beschreibt den Lesevorgang im Detail.

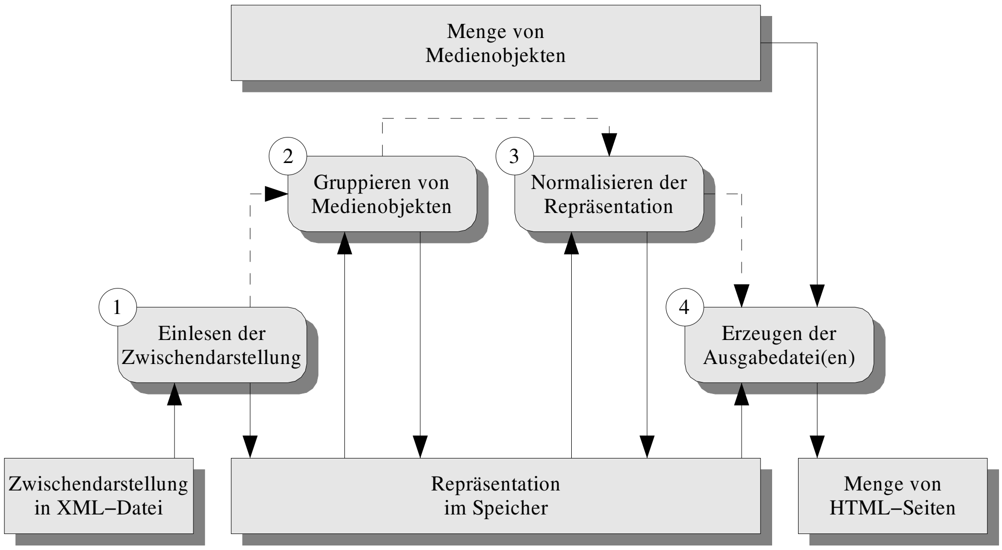

Abbildung 5.1.: Die vier Arbeitsschritte des Generators

2. Um die einzelnen Medienobjekte nicht in einer vollkommen willkürlichen Reihenfolge auf die Ausgabedatei(en) zu verteilen, wird im folgenden Schritt eine *Gruppierung* der Medienobjekte vorgenommen. Außerdem werden Verweise, die nach der Gruppierung überflüssig sind, entfernt. Dieser Schritt, der sich vielleicht ein wenig mit der Arbeit des Layouters einer Tageszeitung vergleichen läßt, stellt eine rudimentäre Ordnung des aus vielen Einzelstücken („Einzelartikeln“) bestehenden Endresultates her. Abschnitt 5.5 geht genauer auf den Vorgang der Gruppierung ein und beschreibt, anhand welcher Kriterien dieser durchgeführt wird.

3. Im nächsten Schritt wird die Repräsentation des Hyperdokumentes in eine *Normalform* überführt. Diese Normalform dient im wesentlichen dazu, den abschließenden Ausgabeschritt zu vereinfachen, da dann während des Traversierens weniger Sonderfälle zu berücksichtigen sind als bei einer nicht normalisierten Darstellung. Abschnitt 5.6 beschreibt, welche Eigenschaften die Normalform des Hyperdokumentes besitzt und wie diese herbeigeführt wird.

4. Abgeschlossen wird die Arbeit des Generators im letzten Schritt durch das Erzeugen einer oder mehrerer Ausgabedateien, die mit HTML-Code gefüllt sind. Nach den umfangreichen Vorbereitungsschritten fällt die eigentliche Ausgabe relativ unkompliziert aus. Wie Abschnitt 5.7 zeigen wird, kommt sie mit einem einzigen Durchlauf der Struktur aus, die das Hyperdokument repräsentiert.

Abbildung 5.1 stellt die Abfolge der vier Arbeitsschritte dar. Durchgezogene Pfeile symbolisieren den Datenfluß, gestrichelte Pfeile stehen für den Kontrollfluß innerhalb des Generators.

5.2. Implementierung

Der Generator ist komplett in Java implementiert und sollte – da keine speziellen Merkmale von Java 2 verwendet wurden – in beliebigen Java-Laufzeitumgebungen ausgeführt werden können. Tatsächlich wurde die Implementierung unter dem *Java Developer's Kit 1.1.8* für OS/2 Warp durchgeführt, das von IBM stammt. Jede Klasse befindet sich in einer eigenen Quelldatei, wie es in Java üblich ist. Die Implementierung teilt sich auf in ein Paket `dodl.core`, welches die Basisklassen zur Realisierung der Datenstruktur enthält, sowie ein Paket `dodl.util`, in welchem sich die eher algorithmischen Anteile wiederfinden. Hinzu kommt ein Paket `dodl`, das die startbare Klasse, gewissermaßen das Hauptprogramm des Generators, enthält.

Die Schritte ② und ④, die direkt vom Zielformat bzw. der gewünschten Gruppierung der Medienobjekte abhängen, sind jeweils in einer eigenen Klasse gekapselt. Durch einfaches Ersetzen einer dieser Klassen kann also ein völlig anderes Zielformat bzw. eine andere Gruppierung von Medienobjekten erreicht werden.

Schritt ① ist ebenfalls in einer eigenen Klasse gekapselt, so daß es theoretisch möglich ist, auch diesen durch Ersetzen der Klasse anders zu realisieren. Das scheint auf den ersten Blick wenig Sinn zu machen, da man sich auf die XML-basierte Zwischendarstellung festgelegt hat. Es wäre aber denkbar, Compiler, ausführbare Spezifikation und Generator irgendwann als ein einziges, in sich geschlossenes Java-Programm zu realisieren. In diesem Fall könnte Schritt ① so abgeändert werden, daß die Repräsentation des Hyperdokumentes im Generator direkt aus der Repräsentation im Compiler oder in der übersetzten Spezifikation übernommen wird. Es würde dann eine direkte Strukturtransformation stattfinden, an der keine externe Datei mehr beteiligt sein muß.

5.3. Speicherrepräsentation des Hyperdokumentes

Um das Verständnis der einzelnen Arbeitsschritte des HTML-Generators zu erleichtern, soll zunächst die Datenstruktur vorgestellt werden, auf der diese Schritte ablaufen, also die Repräsentation des Hyperdokumentes im Speicher.

Die Repräsentation greift einige Ideen auf, die sich bereits bei der Implementierung der Laufzeitbibliothek des *DoDL*-Compilers in Kapitel 3 bewährt haben. Außerdem ist selbstverständlich eine gewisse Verwandtschaft zu den Elementen der in Kapitel 4 vorgestellten Sprache vorhanden.

- Die Klassen orientieren sich unmittelbar an den grundlegenden Begriffen und Beziehungen, mit denen wir im Kontext von *DoDL* zu tun haben. Es existieren also wieder Klassen zur Modellierung von Positionen, Verweisen und Attributen. Auch die einzelnen Medientypen werden durch Klassen repräsentiert, so daß dem Gesamtsystem leicht neue Medientypen hinzugefügt werden können.

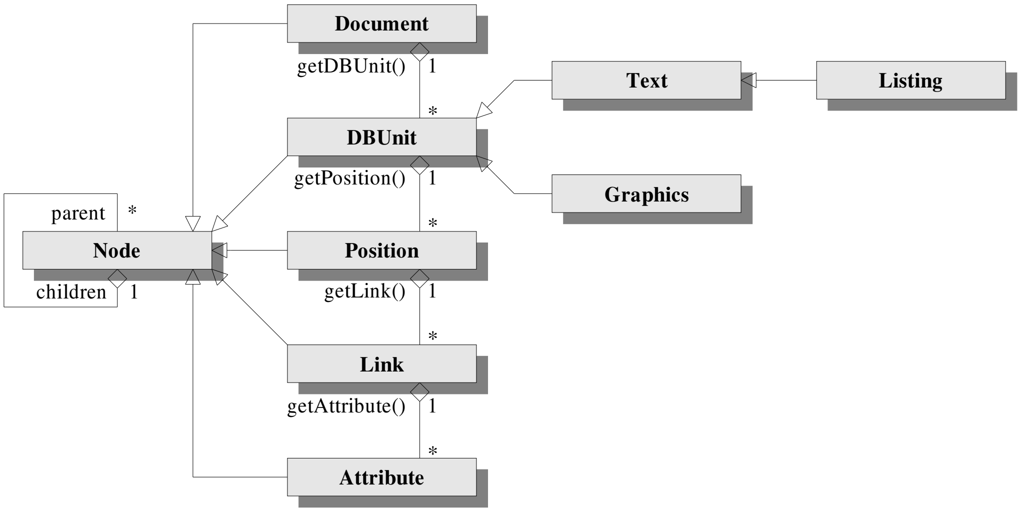

Abbildung 5.2.: Die Basisklassen des Generators

- Zwischen allen Objekten besteht zur Laufzeit eine zyklenfreie *1:n*-Aggregation (jedes Objekt besitzt eine beliebige Anzahl untergeordneter Objekte), wodurch die gesamte entstehende Datenstruktur erneut ein Baum von Objekten ist, der den eigentlichen Strukturgraphen implizit enthält. Jedoch fällt der Objektbaum im Generator etwas einfacher aus als in der Laufzeitbibliothek, da keine benutzerdefinierten Klassen mit beliebiger „Schachtelungstiefe" mehr benötigt werden. Der Baum hat also eine feste maximale Tiefe.

- Die einzelnen Klassen sollen wieder die Vorteile der Vererbung nutzen. Unter anderem sollen alle Klassen letztlich Nachkommen einer ultimativen abstrakten Basisklasse `Node` sein. Diese besitzt bereits die Funktionalität zum Verwalten untergeordneter Objekte, ihrer Kindknoten, und zum Traversieren der daraus entstehenden Struktur.

- Die Klassen stellen keine öffentlich zugänglichen Konstruktoren bereit. Statt dessen kommt das Entwurfsmuster *Fabrik* [Gamm95] zum Einsatz: Jede Klasse besitzt eine Methode zum Erzeugen von untergeordneten Objekten einer bestimmten Klasse. Auf diese Weise wird verhindert, daß unsinnige Objektbäume entstehen, etwa solche, bei denen einem Attribut ein Verweis untergeordnet ist und nicht umgekehrt. Die algorithmischen Teile des Generators profitieren davon, indem sie den Objektbaum nicht auf Korrektheit prüfen oder abwegige Sonderfälle berücksichtigen müssen, die eigentlich nicht auftreten sollen oder dürfen.

5.3.1. Implementierung

Aus diesen Grundideen ergeben sich nun einige wenige Basisklassen, deren Aufbau und grundsätzliche Beziehungen zueinander in Abbildung 5.19 dargestellt sind. Es fällt auf, daß an einigen Stellen der Abbildung Aggregationen mit dem Namen einer Methode markiert sind. Grund dafür ist die Tatsache, daß die Verwaltung untergeordneter Objekte zwar bereits im Attribut `children` der Klasse `Node` implementiert ist, aber der Zugriff auf diese Objekte erst in den Nachkommen und dafür mit dem korrekten Typ der Kinder ermöglicht wird. Konkret bedeutet dies folgendes:

- Die Basisklasse `Node` besitzt das private Attribut `children` zur Verwaltung aller Kindknoten, auf das über eine geschützte Methode `getChild()` zugegriffen wird. Der Ergebnistyp von `getChild()` ist `Node`.
- Die Klasse `Document` – zum Beispiel – bedient sich dieser Methode, um eine eigene öffentliche Methode `getDBUnit()` bereitzustellen, welche einen Kindknoten ermittelt und mit dem korrekten Typ zurückliefert. Im Fall von `getDBUnit()` ist der Ergebnistyp naheliegenderweise `DBUnit`.

Durch diese auf den ersten Blick etwas umständlich anmutende Vorgehensweise entfällt im restlichen Code die Notwendigkeit von Typwandlungen, womit eine potentielle Quelle von Laufzeitproblemen eliminiert wird. Die folgenden Abschnitte beschreiben die einzelnen Klassen im Detail.

5.3.2. Klasse `Node`

Die Klasse `Node` ist die abstrakte Basis, von der alle anderen Klassen abgeleitet sind. Sie kapselt einen grundlegenden Knoten im Objektbaum, der eine beliebige Anzahl von Kindern verwalten kann. Es existieren außerdem Möglichkeiten, den unmittelbar übergeordneten Knoten sowie den Wurzelknoten des Baums zu ermitteln. Schließlich wird noch eine Methode bereitgestellt, die für jeden Knoten eine programmweit eindeutige Kennung auf der Basis eines Instanzzählers zurückliefert. Diese kann zum Erzeugen von Dateinamen oder HTML-Ankern verwendet werden.

5.3.3. Klasse `Document`

Die Klasse `Document` kapselt ein Dokument, ist also die Wurzel des Objektbaums. Sie besitzt private Attribute zur Aufnahme der XML-Attribute `spec`, `bind`, `user` und `date` (siehe Abschnitt 4.5.1) sowie öffentliche Methoden zum Zugriff auf diese Attribute. Zudem stellt die Klasse eine Fabrikmethode für untergeordnete Knoten bereit. Im Unterschied zu allen anderen Klassen, die Kindknoten erzeugen können, erwartet die Fabrikmethode `createDBUnit()` als Argument die *Klasse* eines Medienobjektes (also `DBUnit` oder einen

Nachfahren davon). Sie erzeugt dann eine Instanz dieser Klasse und fügt sie in den Baum ein. Durch die parametrisierte Fabrikmethode wirken sich neue Medientypen nicht auf die Implementierung der Klasse `Document` aus, was die Erweiterung des Systems erleichtert.

5.3.4. Klasse `DBUnit`

Die Klasse `DBUnit` ist die Basisklasse für Medienobjekte. Sie besitzt ein privates Attribut zur Aufnahme des XML-Attribute `file` (siehe Abschnitt 4.5.5) sowie öffentliche Methoden zum Zugriff auf dieses Attribut. Zudem stellt die Klasse eine Fabrikmethode für untergeordnete Knoten der Klasse `Position` und Methoden zum Erfragen dieser Knoten bereit.

5.3.5. Klasse `Text`

Die Klasse `Text` ist ein spezialisierter Nachkomme von `DBUnit` zur Repräsentation von einfachem (Fließ-) Text. Die Klasse besitzt eine Methode, die den gesamten Inhalt der zugehörigen Textdatei ermittelt. Außerdem existieren verschiedene Hilfsmethoden, welche für die in Abschnitt 3.5.6 angesprochene, vom Medientyp abhängige Interpretation des Wertes einer Position zuständig sind.

5.3.6. Klasse `Listing`

Die Klasse `Listing` ist ein Nachkomme von `Text`, der für die Repräsentation von bereits formatiertem Text (speziell Quellcode) verwendet wird. `Listing` besitzt keine speziellen neuen Attribute oder Methoden. Die wesentliche Information verbirgt sich in diesem Fall im Typ selbst, also darin, daß ein bestimmtes Medienobjekt Instanz von `Listing` ist und entsprechend behandelt werden kann.

5.3.7. Klasse `Graphics`

Die Klasse `Graphics` ist ein weiterer spezialisierter Nachkomme von `DBUnit`, der ein graphisches Medienobjekt repräsentiert. Auch hier steckt die wesentliche Information bereits im Typ des Medienobjektes.

5.3.8. Klasse `Position`

Die Klasse `Position` kapselt eine Position. Sie besitzt ein privates Attribut zur Aufnahme des XML-Attributes `value` (siehe Abschnitt 4.5.6) sowie ein weiteres Attribut `type`, das den Typ der Position speichert. Der Typ ist `BEGIN`, `END` oder `OCC`, je nachdem, durch welche Funktion die Position entstanden ist. Außerdem existieren öffentliche Methoden zum Zugriff auf die beiden Attribute. Schließlich stellt die Klasse noch eine Fabrikmethode für untergeordnete Knoten der Klasse `Link` und Methoden zum Erfragen dieser Knoten bereit.

5.3.9. Klasse `Link`

Die Klasse `Link` kapselt einen Verweis. Ausgangspunkt des Verweises ist die Position, welcher der Verweis untergeordnet ist. Das Ziel kann über Methoden ermittelt und verändert werden. Zudem stellt die Klasse eine Fabrikmethode für untergeordnete Knoten der Klasse `Attribute` und Methoden zum Erfragen dieser Knoten bereit.

5.3.10. Klasse `Attribute`

Die Klasse `Attribute` kapselt ein (Feature-) Attribut. Sie besitzt private (Java-) Attribute zur Aufnahme der (XML-) Attribute `name` und `value` sowie öffentliche Methoden zum Zugriff auf diese Attribute[1]. Die Klasse besitzt keine Fabrikmethode zum Erzeugen untergeordneter Objekte. Instanzen von `Attribute` sind damit immer Blätter und niemals innere Knoten des Objektbaumes.

Nachdem die grundsätzlichen Klassen zum Aufbau der Datenstruktur bekannt sind, können wir uns nun den einzelnen Arbeitsschritten des Generators zuwenden.

5.4. Einlesen der Zwischendarstellung

Der erste Arbeitsschritt besteht darin, die von der ausführbaren Spezifikation erzeugte Zwischendarstellung aus der XML-Datei zu lesen und in eine Datenstruktur zu überführen, die sich aus den zuvor beschriebenen Klassen zusammensetzt.

5.4.1. Verwendete XML-Bibliothek

Für die syntaktische Analyse der Datei wird ein Parser benötigt, der sowohl die allgemeine Syntax von XML als auch die spezielle Syntax der Zwischendarstellung versteht. Da die Existenz frei verfügbarer Bibliotheken als eines der Argumente sowohl für die Verwendung von XML als Basis der Zwischendarstellung als auch für Java als Implementierungssprache angeführt wurde, bedient sich dieser Schritt wesentlich eines freien XML-Paketes für Java [Sun99], das von Sun Microsystems angeboten wird. Dieses Paket stellt einen Parser für XML bereit, der unter anderem die folgenden beiden Eigenschaften besitzt:

[1] An dieser Stelle ist die Konfusion natürlich groß, aber das kann passieren, wenn sich alle Gebiete des gleichen Vokabulars bedienen.

- Er arbeitet *ereignisorientiert*. Es wird also nicht automatisch ein Syntaxbaum aufgebaut, sondern es können stattdessen spezielle Ereignisse, die während der Syntaxanalyse auftreten, über Methodenaufrufe an ein anderes Objekt weitergeleitet werden. Zu den Ereignissen, die generiert werden, gehören zum Beispiel folgende:
  - Zwei Ereignisse `startDocument()` und `endDocument()` die auftreten, wenn die Syntaxanalyse einer XML-Datei beginnt oder endet.
  - Zwei Ereignisse `startElement()` und `endElement()`, die auftreten, wenn ein XML-Element eingeleitet oder abgeschlossen wird.
  - Das Ereignis `characters()`, das auftritt, wenn textueller Inhalt gelesen wird.
- Der Parser arbeitet auf Wunsch *validierend*, das heißt, er prüft neben Wohlgeformtheit eines XML-Dokumentes auch dessen Gültigkeit, also die Konformität zu einer vorgegebenen DTD (siehe auch Abschnitt 4.3).

Ein kleines Beispiel soll die Arbeitsweise dieses ereignisorientierten Parsers verdeutlichen. Es sei die folgende sehr einfache XML-Datei gegeben:

```
<test name="value">
 Hallo, Welt!
 </test>
 </fehler>
```

Die syntaktische Analyse dieser Datei mit dem vorgestellten Parser würde das Auftreten der folgenden vier Ereignisse in Form von Methodenaufrufen mit den entsprechenden Argumenten zur Folge haben:

```
startDocument();
startElement("test", attributes);
characters("Hallo, Welt!");
endElement("test");
```

Dabei ist `attributes` ein spezielles Objekt, das alle Attribute eines XML-Elementes enthält und sehr bequem den Zugriff auf den Wert jedes Attributes über dessen Namen erlaubt. Nach dem Ereignis `endElement()` würde die Syntaxanalyse mit einer Ausnahmebedingung abbrechen – das Auftreten des schließenden Elementes `</fehler>` ohne vorheriges Auftreten des zugehörigen öffnenden Elementes ist eine Verletzung der allgemeinen Syntaxregeln von XML. Das Beispiel zeigt also ein nicht wohlgeformtes (und damit auch nicht gültiges) XML-Dokument.

5.4.2. Anbindung an eigenen Code

Das verwendete XML-Paket bietet verschiedene Varianten an, wie die vorgestellten Ereignisse vom Parser an ein Objekt des Benutzers weitergegeben werden können. Die einfachste Variante besteht darin, eine eigene Klasse von einer bereitgestellten Klasse `HandlerBase` abzuleiten, in dieser die benötigten Methoden zur Ereignisbehandlung zu überschreiben und dem Parser mitzuteilen, welche Instanz der eigenen Klasse zur Ereignisbehandlung verwendet werden soll.

Für den speziellen und im Sinne von XML sehr einfachen Anwendungsfall dieser Diplomarbeit reicht es vollkommen aus, die Methode `startElement()` zu überschreiben, also auf den Beginn eines neuen XML-Elementes zu reagieren. Das liegt daran, daß sich sämtliche Information der Zwischendarstellung innerhalb der Elemente befindet. Textueller Inhalt (das „Hallo, Welt!“ aus dem obigen Beispiel), ist derzeit nicht vorgesehen und wird folglich ignoriert, weshalb insbesondere die Methode `characters()` nicht benötigt wird.

Die Implementierung von `startElement()` erhält als Argumente den Namen des aufgetretenen XML-Elementes und dessen Attribute. Ist ein Attribut `id` vorhanden, so wird es in eine Hash-Tabelle eingetragen, die später ein schnelles Auffinden von XML-Elementen bzw. assoziierten Objekten ermöglicht. Anschließend führt die Methode abhängig vom Typ des aufgetretenen Elementes eine der folgenden Aktionen durch:

- `<document>` – Für dieses Element, das nur einmal auftreten kann, wird eine neue Instanz von `Document` erzeugt. Die XML-Attribute `spec`, `bind`, `user` und `date` werden unmittelbar in das neue Objekt übernommen, das die Wurzel des Objektbaums bildet.

- `<dbunit>` – Für dieses Element wird eine neue Instanz der Klasse erzeugt, die durch das XML-Attribut `type` vorgegeben ist. Derzeit kann dies entweder `Text`, `Graphics` oder `Listing` sein. Anderenfalls bricht der Generator mit einem Fehler ab. Das XML-Attribut `file` wird unmittelbar in das Objekt übernommen.

- `<position>` – Für dieses Element wird eine neue Instanz von `Position` erzeugt und der Instanz von `DBUnit` untergeordnet, die durch das XML-Attribut `ref` vorgegeben ist. Das mit `ref` assoziierte Objekt wird aus der Hash-Tabelle ermittelt. Dies ist möglich, da aufgrund der Struktur der Zwischendarstellung sämtliche Elemente `<dbunit>` den Elementen `position` vorangehen (vergleiche Kapitel 4). Das Attribut `value` wird direkt in das neue Objekt übernommen, der Wert von `type` wird daraus abgeleitet.

- `<link>` – Für dieses Element wird eine neue Instanz von `Link` erzeugt und der zuletzt erzeugten Instanz von `Position` untergeordnet. Das Ziel des Verweises wird aus dem XML-Attribut `target` übernommen, kann jedoch nicht unmittelbar über die Hash-Tabelle aufgelöst werden, da es möglicherweise auf eine Position verweist, die noch nicht gelesen wurde. Aus diesem Grund werden die Ziele aller Verweise erst im Anschluß an die Syntaxanalyse aufgelöst.

- `<attribute>` – Für dieses Element wird eine neue Instanz von `Attribute` erzeugt und der Instanz von `Link` untergeordnet, die durch das XML-Attribut `ref` vorgegeben ist. Auch hier kann die Auflösung des Attributes `ref` über die Hash-Tabelle be-

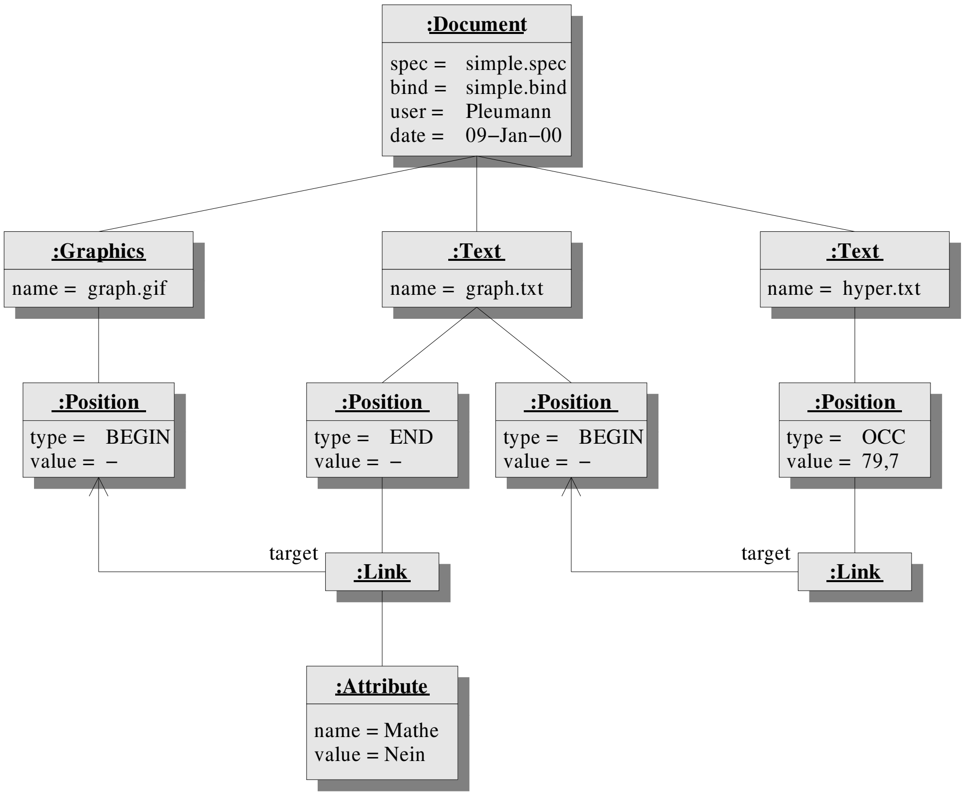

Abbildung 5.3.: Beispieldokument nach dem Lesen der XML-Datei

reits während syntaktischen Analyse geschehen, weil sämtliche Verweise in der Zwischendarstellung den Attributen vorangehen. Die XML-Attribute `name` und `value` werden direkt in dieses Objekt übernommen.

5.4.3. Implementierung

Die Implementierung des gesamten Lesevorgangs ist in der Klasse `Loader` gekapselt, die in der Datei `Loader.java` implementiert ist. Das Laden wird durch einen Aufruf der Methode `execute()` angestoßen, die als Argument den Namen der XML-Datei erwartet und als Ergebnis – eine erfolgreiche syntaktische Analyse vorausgesetzt – die Wurzel des erzeugten Objektbaums zurückgibt.

5.4.4. Beispiel

Abbildung 5.3 zeigt als Beispiel eine Untermenge der Datenstruktur, die sich aus der Syntaxanalyse der Zwischendarstellung aus Abschnitt 4.3 ergibt, welche ihrerseits auf

Abbildung 2.1 beruht. Sie enthält aus Platzgründen wieder nur die Objekte, die sich in der rechten Hälfte des ursprünglichen Beispiels wiederfinden. Jeder Knoten ist mit dem Namen seiner Klasse sowie seinen wichtigsten Attributen markiert, wobei ein Strich auf ein undefiniertes Attribut hindeutet. Die Beziehung zwischen über- und untergeordneten Knoten im Baum ergibt sich wieder durch die Anordnung, so daß auch hier bei der entsprechenden Relation auf Pfeilspitzen verzichtet wurde. Die mit `target` markierten Relationen geben das Ziel eines Verweises an. Der Weg dieses Beispiel-Baumes durch den Generator soll in den folgenden Abschnitten weiterverfolgt werden.

5.5. Gruppieren von Medienobjekten

Im Zusammenhang mit dem HTML-Generator stellt sich erstmalig die Frage nach dem Layout des entstehenden Hyperdokumentes. Der Ansatz von *DoDL* legt Schwerpunkte auf Verknüpfungsstruktur und Laufzeitverhalten eines Hyperdokumentes und läßt die Frage nach dem Layout vollkommen offen. Dennoch sollte das Layout nicht vollkommen ignoriert werden, wenn das Werkzeug, das Ergebnis dieser Arbeit ist, tatsächlich benutzt werden soll. Tatsächlich tauchen im Kontext des Layouts zwei verschiedene Fragen auf, die eng miteinander verknüpft sind.

5.5.1. Aufteilung der Medienobjekte

Die erste Frage ist die nach der Aufteilung des Hyperdokumentes bzw. der enthaltenen Medienobjekte: Welcher Zusammenhang besteht zwischen den Medienobjekten der *DoDL*-Spezifikation und den Seiten des entstehenden HTML-Dokumentes? Auf diese Frage gibt es verschiedene mögliche Antworten:

1. Jedes Medienobjekt wird in einer eigenen HTML-Seite plaziert. Verweise, die von einem Teil des Medienobjektes (etwa dem Vorkommen einer Zeichenkette in einem Text) ausgehen, werden direkt zu HTML-Ankern. Für Verweise, die vom Anfang oder vom Ende eines Medienobjektes ausgehen, werden – im einfachsten Fall textuelle – Verweise „Weiter“ und „Zurück“ hinzugefügt, mit denen der Leser zum vorangehenden beziehungsweise nächsten Medienobjekt springen kann. Offensichtlich befindet sich das Ziel der Verweise bei diesem Lösungsansatz stets in einer anderen Datei. Das Ergebnis ist ein sehr stark fragmentiertes Hyperdokument, das zwar die gewünschte Struktur besitzt, aber insbesondere bei vielen kleinen Medienobjekten nur schwer zu lesen ist, da es nicht möglich ist, zusammengehörige Information auch im Zusammenhang zu präsentieren.

2. Es existiert nur eine einzige HTML-Seite, die alle Medienobjekte beinhaltet. Die Behandlung von Verweisen entspricht der in Lösung 1, jedoch sind alle Verweise zwangsläufig sogenannte *inline links*, da sie auf eine andere Stelle der gleichen Datei verweisen. Bei dieser Lösung wird ein Effekt erzielt, der dem von Lösung 1 genau

entgegengesetzt ist: Es ist zwar möglich, zusammengehörige Information auch im Zusammenhang zu präsentieren, aber da zwangsläufig das gesamte Hyperdokument sichtbar ist, dürfte der Leser ebenfalls eher verwirrt sein.

3. Da die beiden ersten Lösungen nicht wirklich befriedigend sind, wird eine Möglichkeit geschaffen, Einfluß auf die Aufteilung der Seiten zu nehmen. Dies geschieht innerhalb der Spezifikation mit Hilfe eines Attributes `group`, das jedem Medienobjekt zugeordnet werden kann. Genau die Medienobjekte, deren Attribute `group` den gleichen Wert besitzen, bilden einen Sinnzusammenhang, eine *Gruppe* von Medienobjekten, die vom Generator entsprechend behandelt werden soll: Der HTML-Generator plaziert jede Gruppe naheliegenderweise in einer einzelnen HTML-Seite.

5.5.2. Reihenfolge der Medienobjekte

Ein weiteres Problem, das zwar nicht bei Lösung 1, jedoch bei den Lösungen 2 und 3 auftritt, ist das der Reihenfolge der Medienobjekte in den erzeugten HTML-Seiten. Dieses Problem mag für einen Leser, der mit *DoDL* nicht vertraut ist, überraschend sein, da dieser vielleicht annimmt, die Reihenfolge ergäbe sich aus der Reihenfolge der zugehörigen Dokument-Deklarationen in der Spezifikation. Dies ist jedoch nicht der Fall: Die Struktur ergibt sich allein aus den Verweisen, die in der `construct`-Sektion festgelegt werden. Es stellt sich also die Frage, in welcher Reihenfolge jene Medienobjekte, die Teil einer HTML-Seite sind, in dieser plaziert werden? Auch hier existieren wieder mehrere mögliche Lösungen:

1. Das Problem der Reihenfolge wird gänzlich ignoriert. Die Medienobjekte werden so in die HTML-Seite geschrieben, wie es für den Generator angenehm ist, also so, wie sie beim Traversieren des Graphen vorgefunden werden. Da zwischen verknüpften Medienobjekten Verweise existieren, ist es stets möglich, dem vorgesehenen Textfluß zu folgen. Es gibt jedoch einige Fälle, in denen diese Lösung etwas „entartet“. Man stelle sich als Beispiel eine Folge von mehreren Medienobjekten vor (etwa Text$\rightarrow$ Grafik$\rightarrow$ Text), die zu einer linearen Liste verknüpft sind, also auch als zusammenhängende Gruppe in entsprechender Reihenfolge zu lesen sind.

   - Es kann passieren, daß diese Medienobjekte in vollkommen verschiedenen Teilen einer HTML-Seite plaziert werden, daß sich also zwischen ihnen Medienobjekte befinden, die nicht Teil der Gruppe sind. Dabei wäre es offensichtlich wünschenswert, wenn die „intuitive“ Reihenfolge der Medienobjekte auch bei der Präsentation eingehalten würde.
   - Angenommen, die Reihenfolge der drei Medienobjekte wird eingehalten oder ist zufällig erfüllt. Dann ist es trotzdem so, daß zwischen den Medienobjekten jeweils ein Verweis „Weiter“ existiert, der zum nächsten Medienobjekt führt, obwohl sich dieses unmittelbar unter dem vorangehenden befindet. Dieser Verweis ist offensichtlich überflüssig.

2. Es ist möglich, analog zu `group` ein Attribut `order` zu verwenden, mit dem die Reihenfolge der Medienobjekte in einer Gruppe kontrolliert werden kann. Der Generator kann auch dieses Attribut bei der Ausgabe berücksichtigen, die überflüssigen Verweise verschwinden jedoch dadurch nicht.

5.5.3. Automatisches Bilden von Gruppen

Die beiden vorgestellten Lösungen für die Probleme von Aufteilung und Reihenfolge funktionieren zwar, bedeuten jedoch für den Autor eines Hyperdokumentes zusätzlichen Aufwand bei der Spezifikation. Diesen zusätzlichen Aufwand würde man sicherlich gerne minimieren oder möglichst ganz vermeiden. Es stellt sich also die Frage, ob eine alternative Lösung möglich ist, bei der Aufteilung und Reihenfolge automatisch bestimmt werden. Eine solche Lösung ist in der Tat möglich. Sie basiert auf der Beobachtung, daß nicht alle Verknüpfungen innerhalb des Strukturgraphen Hyperlinks im eigentlichen Sinne sind oder in jedem Fall als solche behandelt werden müssen. Tatsächlich scheint es zwei Typen von Verküpfungen zu geben:

1. Verknüpfungen, die vom Ende eines Medienobjektes zum Beginn eines anderen Medienobjektes führen. Es macht eigentlich keinen Sinn, für diese Verknüpfungen Hyperlinks zu generieren, denn es existiert kein Text oder Grafik-Ausschnitt, um den ein Anker gelegt werden könnte. Ohne diese Hervorhebung hat der Leser aber keine Möglichkeit, dem Verweis zu folgen – er kann ihn überhaupt nicht wahrnehmen. Aus diesem Grund wurden bei den obigen Lösungen die Verweise „Weiter“ und „Zurück“ automatisch hinzugefügt.
2. Alle anderen Verknüpfungen, insbesondere solche, die von einem Bereich innerhalb eines Medienobjektes ausgehen, also auf dem Vorkommen eines Suchparameters basieren und mit Hilfe der Funktion `getOcc()` entstanden sind. Diese Verknüpfungen sind tatsächlich nutzbar, um Hyperlinks zu erzeugen, denn hier läßt sich ein Anker um den entsprechenden Ausschnitt legen, den der Leser dann sowohl sehen als auch anwählen kann.

Der Grundgedanke der Gruppierungsautomatik ist nun, aus Verweisen des Typs 1 im Hyperdokument die Aufteilung und Reihenfolge der Medienobjekte, also die Werte der Attribute `group` und `order` abzuleiten. Dies ist tatsächlich möglich, solange nicht von einer Position aus mehrere Verweise ausgehen, die in dieser Beziehung „mehrdeutig“ sind. Bevor jedoch ein Algorithmus vorgestellt wird, der diese Automatik realisiert, soll zunächst der Begriff der *Gruppe* etwas exakter definiert werden, als dies durch die umgangssprachliche Charakterisierung eines *Sinnzusammenhangs* möglich ist.

5.5.4. Definition von Gruppen

Gegeben sei die Menge M aller Medienobjekte eines Hyperdokumentes und die Menge P aller Positionen dieses Hyperdokumentes. Gegeben seien ferner die folgenden Funktionen und Relationen, die sich direkt aus den in [Dobe95, Dobe96a] verwendeten Prolog-Prädikaten ableiten:

- Eine Funktion $begin : M \to P$ zur Abbildung eines Medienobjektes auf die feste, ausgezeichnete Position, die seinen Beginn repräsentiert.
- Eine Funktion $end : M \to P$ zur Abbildung eines Medienobjektes auf die feste, ausgezeichnete Position, die sein Ende repräsentiert.
- Eine Relation $link \subseteq P \times P$ mit der Eigenschaft, daß $(p, q) \in link$ genau dann gilt, wenn im Hyperdokument ein Verweis von p nach q existiert.

Aufbauend auf diesen Gegebenheiten ist der Begriff der *Gruppe* wie folgt definiert:

Jedes 1-Tupel $g = (m)$ mit $m \in M$ heißt Gruppe der Länge 1.

Ein n-Tupel $g = (m_1, \ldots, m_n)$ mit $m_1, \ldots, m_n \in M$ und $n > 1$ heißt Gruppe der Länge n, wenn die folgenden drei Bedingungen erfüllt sind:

1. Die n Elemente des Tupels sind *von links nach rechts* über $n-1$ Verweise des Typs 1 verbunden.

$$\forall i \in \{1, \ldots, n-1\} : (end(m_i), begin(m_{i+1})) \in link$$

2. Die letzten $n-1$ Elemente des Tupels dürfen Ziel nur eines Verweises vom Typ 1 sein, müssen also innerhalb des gesamten Hyperdokumentes *eindeutige Vorgänger* besitzen.

$$\forall i \in \{2, \ldots, n\} : \neg\exists m \in M, m \neq m_{i-1} : (end(m), begin(m_i)) \in link$$

3. Die ersten $n-1$ Elemente des Tupels dürfen Ausgangspunkt nur eines Verweises vom Typ 1 sein, müssen also innerhalb des gesamten Hyperdokumentes *eindeutige Nachfolger* besitzen.

$$\forall i \in \{1, \ldots, n-1\} : \neg\exists m \in M, m \neq m_{i+1} : (end(m_i), begin(m)) \in link$$

Abbildung 5.4 zeigt eine legale Gruppierung eines einfachen, auf Typ-1-Verweise reduzierten Hyperdokumentes. Medienobjekte und Verweise sind durch Rechtecke und Pfeile dargestellt. Die Gruppen sind schraffiert hinterlegt. Daß die Gruppierung legal ist, läßt sich leicht anhand der Definition nachprüfen. Die in Abbildung 5.5 dargestellte Gruppierung ist hingegen illegal. Medienobjekt 2 besitzt keinen eindeutigen Nachfolger, die Gruppe dürfte also hinter diesem Medienobjekt nicht fortgesetzt werden. Außerdem besitzt Medienobjekt 7 keinen eindeutigen Vorgänger, weshalb es der Anfang einer Gruppe sein müßte.

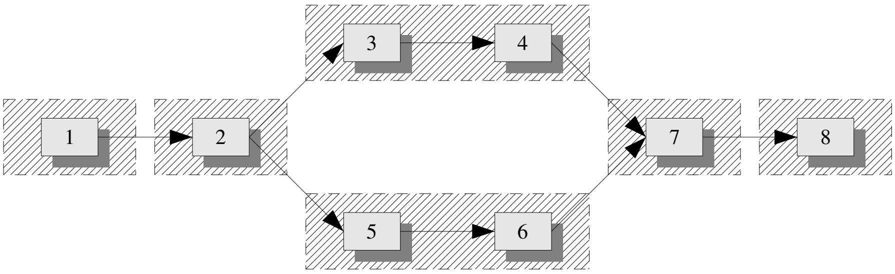

Abbildung 5.4.: Legale Gruppierung

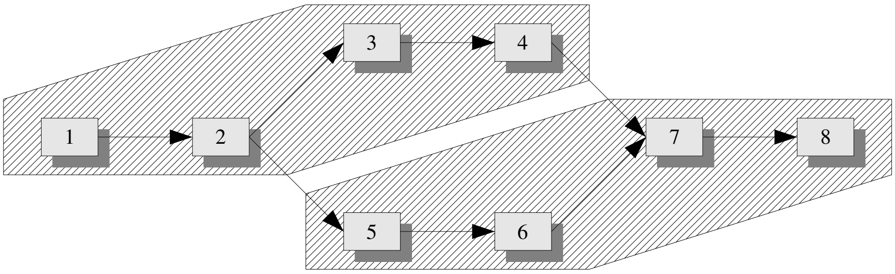

Abbildung 5.5.: Illegale Gruppierung

5.5.5. Eigenschaften von Gruppen

Angenehm an der vorgestellten Definition von Gruppen ist die Tatsache, daß die Reihenfolge der Elemente des Tupels der gewünschten Reihenfolge der Medienobjekte entspricht. Hat man also einmal ein solches Tupel gefunden, dann folgt daraus sowohl eine Belegung für das Attribut `group` der Medienobjekte (das für alle Elemente des Tupels identisch ist) als auch eine für das Attribut `order` (das dem Index des jeweiligen Elementes entspricht).

Etwas weniger angenehm scheint auf den ersten Blick, daß aus der Definition allein keine eindeutige Aufteilung eines Hyperdokumentes auf Gruppen folgt. Es gibt meist mehrere mögliche Aufteilungen, wozu auch die weiter oben erwähnte *triviale* Aufteilung gehört, bei der jedes Medienobjekt seine eigene Gruppe bildet. Es gibt jedoch für jedes Hyperdokument nur genau eine Aufteilung, bei der alle Gruppen maximale Länge besitzen. Dies folgt aus der Forderung nach sowohl eindeutigen Vorgängern als auch eindeutigen Nachfolgern. Diese Aufteilung ist zudem noch effizient berechenbar, wie der im folgenden kurz umrissene Algorithmus zeigt. Er macht sich im wesentlichen die Tatsache zunutze, daß zwei bestehende Gruppen zu einer neuen verschmolzen werden können, wenn anschließend immer noch die Gruppeneigenschaften erfüllt sind. Wendet man diese Verschmelzung auf die Gruppierung aus Abbildung 5.4 an, ergibt sich die in Abbildung 5.6 dargestellte optimale Gruppierung.

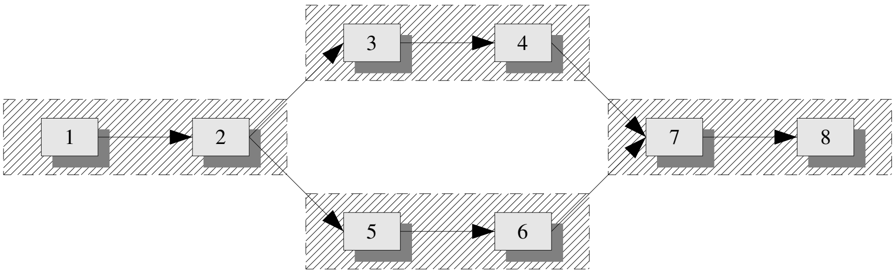

Abbildung 5.6.: Optimale Gruppierung

5.5.6. Algorithmus zur Bestimmung von Gruppen maximaler Länge

Der Algorithmus bedient sich einer Hilfsfunktion *getNextElement*(), deren Argument ein Medienobjekt m ist. Geht von dem Medienobjekt m genau ein Verweis des Typs 1 aus, hat das Medienobjekt also einen eindeutigen Nachfolger n, und ist m der eindeutige Vorgänger von n, dann liefert die Funktion das Ergebnis n. In allen anderen Fällen ist das Ergebnis ein ausgezeichneter Wert *null*, der keinem Medienobjekt entspricht. Abbildung 5.7 zeigt die Arbeitsweise dieser Funktion in Pseudo-Code.

Ferner geht der Algorithmus davon aus, daß die Werte der Attribute `group` und `order` aller Medienobjekte zu Beginn undefiniert sind. Aufbauend auf diesen Voraussetzungen durchläuft er alle Medienobjekte des Hyperdokumentes einmal. Findet er ein Medienobjekt, das noch keiner Gruppe zugeordnet ist, so versucht er, von diesem Medienobjekt ausgehend eine Gruppe zu bilden. Dazu ruft er solange die Hilfsfunktion *getNextElement*(), bis diese *null* zurückliefert. Abbildung 5.8 zeigt die Arbeitsweise des Algorithmus in Pseudo-Code.

Eine kurze Laufzeitbetrachtung zeigt, daß der Algorithmus in polynomieller Zeit arbeitet: Die Hilfsfunktion *getNextElement*() untersucht alle ausgehenden Verweise eines Medienobjektes und alle eingehenden Verweise von dessen potentiellem Nachfolger genau einmal. Die Laufzeit dieser Hilfsfunktion läßt sich also durch $O(v)$ abschätzen, wobei v die Anzahl der Verweise des Hyperdokumentes ist.

Die äußere Schleife des eigentlichen Algorithmus durchläuft alle Medienobjekte genau einmal und versucht dabei Gruppen zu bilden, die vom aktuell betrachteten Medienobjekt m ausgehen. Die innere Schleife hängt einzelne Medienobjekte n an das Ende dieser Gruppe an, solange die Gruppeneigenschaften erfüllt bleiben. Dabei werden im Einzelfall „ältere“ Gruppen an die neue Gruppe angehängt, indem die Gruppennummern der Medienobjekte nacheinander überschrieben werden.

Der *worst case* tritt auf, wenn das Hyperdokument eine Gruppe erlaubt, die alle Medienobjekte umfaßt, und der Algorithmus die Medienobjekte genau in der der Gruppe entgegengesetzten Reihenfolge im Hyperdokument vorfindet. Abbildung 5.9 zeigt diesen Fall, bei dem nach jedem Schritt der äußeren Schleife die innere Schleife alle bisher betrachteten Medienobjekte erneut durchlaufen muß. Die Laufzeit des äußeren Teils des

```
// Eingabe:    Medienobjekt m
// Ausgabe:    Nächstes Medienobjekt der gleichen Gruppe oder null
// Variablen: Medienobjekt n, Verweis v

n := null

for all Verweise v, die von m ausgehen do
  if Typ von v ist 1 then
    if n ungleich null then
      return null // Es existiert kein eindeutiger Nachfolger von m
    else
      n := Medienobjekt, auf das v verweist
    end if
  end if
end for

for all Verweise v, die bei n ankommen do
  if Typ von v ist 1 then
    if n ungleich m then
      return null // m ist nicht eindeutiger Vorgänger von n
    end if
  end if
end for

return n // Nächstes Element der Gruppe wurde gefunden
```

Abbildung 5.7.: Algorithmus zum Finden des nächsten Elementes einer Gruppe

```
// Eingabe:    Alle Medienobjekte und Verweise eines Hyperdokumentes
// Ausgabe:    Gruppierung der Medienobjekte in den Arrays group[] und order[]
// Variablen: Medienobjekte m und n, Integer g und i

g := 0 // Zähler für Gruppen

for all Medienobjekte m des Hyperdokumentes do
  if m besitzt noch keine Gruppe then
    g := g + 1 // Nummer der neuen Gruppe
    i := 1     // Erster Index in der neuen Gruppe
    n := m     // Erstes Medienobjekt der neuen Gruppe
    while n <> null do
      group[n] := g
      order[n] := i
      i := i + 1
      n := getNextElement(n)
    end while
  end if
end for
```

Abbildung 5.8.: Algorithmus zum Finden von Gruppen maximaler Länge

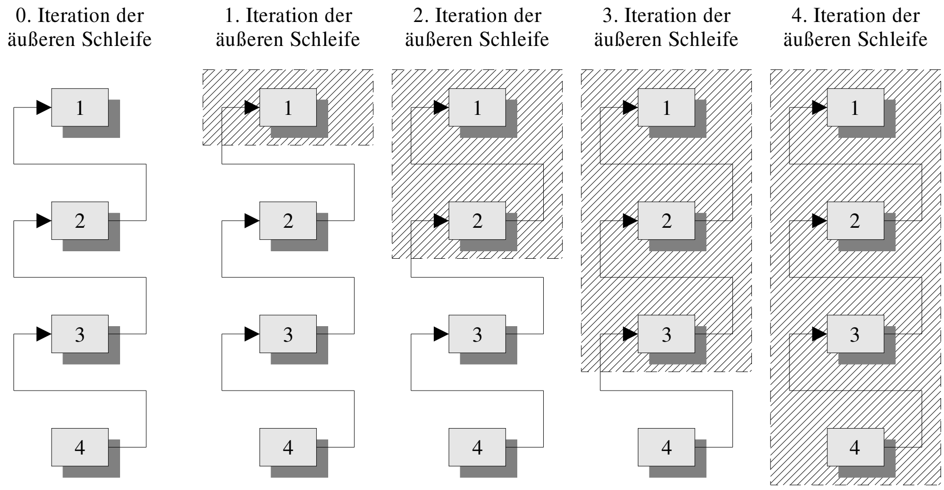

Abbildung 5.9.: Worst-Case-Betrachtung der Laufzeit

Algorithmus ist also $O(m^2)$, wobei m die Anzahl der Medienobjekte des Hyperdokumentes ist. Da jede Iteration der inneren Schleife einmal *getNextElement*() aufruft, ergibt sich eine Gesamtlaufzeit von $O(vm^2)$.

5.5.7. Implementierung

Der Algorithmus zum automatischen Bestimmen maximaler Gruppen ist in der Klasse `Grouper` gekapselt, die in der Datei `Grouper.java` implementiert ist. Die Implementierung entspricht exakt der oben angegebenen Beschreibung. Um die Gruppierung anzustoßen, wird die Methode `execute()` mit einem einzigen Parameter aufgerufen – der Instanz von `Document`, für welche die Gruppierung vorgenommen werden soll.

5.5.8. Beispiel

Abbildung 5.10 zeigt das bekannte Beispieldokument nach der automatischen Bestimmung von Gruppen. Die schraffierten Rechtecke zeigen die Medienobjekte, die sich nun innerhalb in einer Gruppe befinden. Der schraffierte Kreis umschließt einen Verweis, der als überflüssig erkannt entsprechend markiert wurde. Die Ausgabeklasse braucht (und sollte) für diesen Verweis keinen HTML-Code zu erzeugen.

5.6. Normalisieren der Repräsentation

Nach dem Gruppierungsschritt scheint die Repräsentation des Hyperdokumentes eine Gestalt zu besitzen, die es erlaubt, unmittelbar den abschließenden Ausgabeschritt fol-

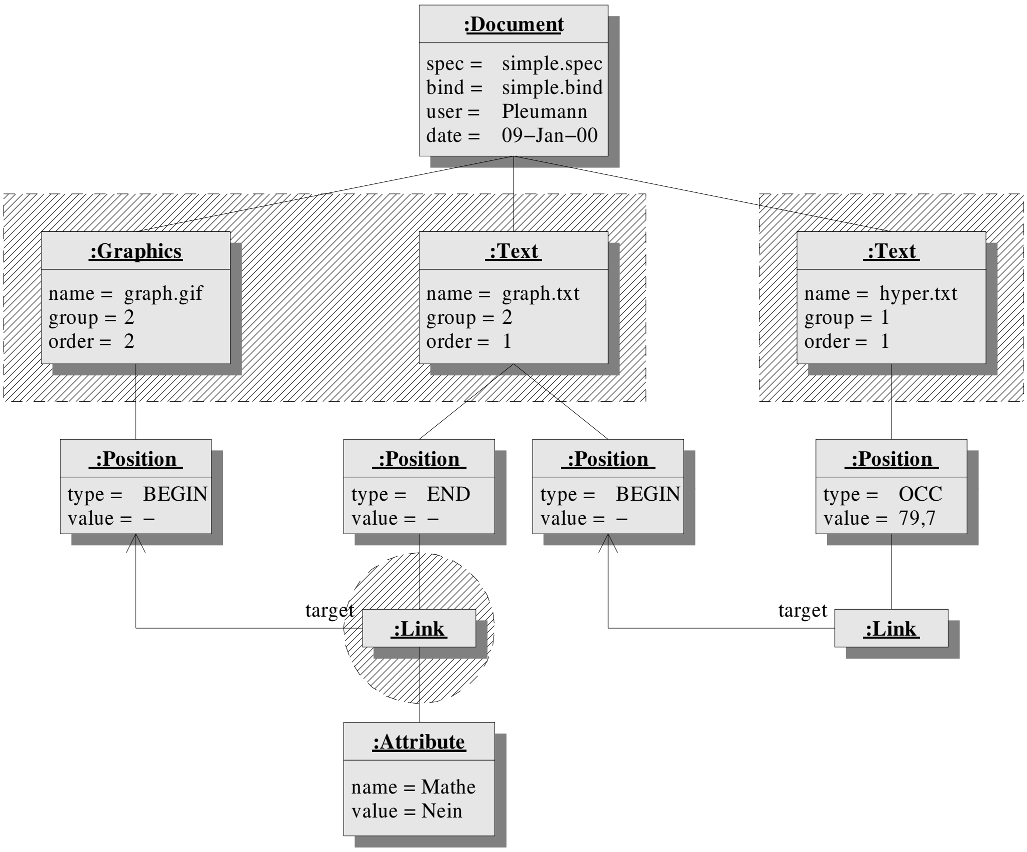

Abbildung 5.10.: Beispieldokument nach der Gruppierung

gen zu lassen: Die gesamte Strukturinformation des Hyperdokumentes ist bereits seit dem ersten Arbeitsschritt des Generators im Speicher vorhanden, durch die Gruppierung existiert ein einfaches Layout, und der eigentliche Inhalt der Medienobjekte muß ohnehin während der Ausgabe aus den einzelnen Dateien gelesen werden. Schaut man sich jedoch die Repräsentation etwas genauer an, dann stellt man fest, daß Teile der Baumstruktur noch ein wenig „ungeordnet“ sind und somit der für die Ausgabe zuständigen Klasse die Arbeit unnötig erschweren:

- Für die Ausgabeklasse wäre es sicherlich hilfreich, die Medienobjekte in genau der Reihenfolge im Baum vorzufinden, in der sie auch in die Ausgabedatei(en) geschrieben werden. Die oberste Stufe des Ausgabealgorithmus würde sich dann auf einen einfachen Durchlauf aller Medienobjekte, also aller direkten Kinder des Wurzelknotens reduzieren. Momentan sind die Medienobjekte zwar durch die Attribute `group` und `order` in geordnete Gruppen aufgeteilt, aber ihre Reihenfolge im Baum entspricht nicht notwendigerweise diesen geordneten Gruppen, wie Abbildung 5.10 zeigt.
- Man kann sich leicht vorstellen, daß bei der Ausgabe eines einzelnen Medienobjektes abwechselnd Positionen und reine Textstücke zwischen diesen Positionen geschrieben werden, bis das Ende des Medienobjektes erreicht ist. Auch dies ließe sich sehr einfach über einen Durchlauf aller Positionen des Medienobjektes realisieren, jedoch sind diese ebenfalls nicht entsprechend geordnet.

Um diese Mängel zu beheben, wird die Repräsentation vor dem abschließenden Ausgabeschritt in eine Art *Normalform* überführt, die folgende Eigenschaften besitzt:

- Sämtliche Medienobjekte des Hyperdokumentes sind aufsteigend nach dem Wert ihres Attributes `group` sortiert. Ist der Wert bei zwei Medienobjekten identisch, sind diese also in der gleichen Gruppe, dann leitet sich ihre Reihenfolge aus dem Attribut `order` ab.
- Für die Positionen eines Medienobjektes gilt:
  - Die erste Position ist stets die, die den Beginn des Medienobjektes repräsentiert. Existiert keine solche Position, wird sie erzeugt.
  - Die letzte Position ist stets die, die das Ende des Medienobjektes repräsentiert. Existiert keine solche Position, wird sie erzeugt.
  - Alle anderen Positionen werden aufsteigend nach ihrem Startindex innerhalb des Medienobjektes sortiert.

Der Vorteil der immer vorhandenen Positionsobjekte für den Beginn und das Ende eines Medienobjektes ist implementierungstechnischer Natur: Es hat sich gezeigt, daß eine ganze Reihe von Fallunterscheidungen während des Traversierens entfallen, wenn sichergestellt ist, daß diese beiden Positionen immer vorhanden sind (und alle Positionen zwischen ihnen folglich solche sein müssen, die durch `getOcc()` entstanden sind).

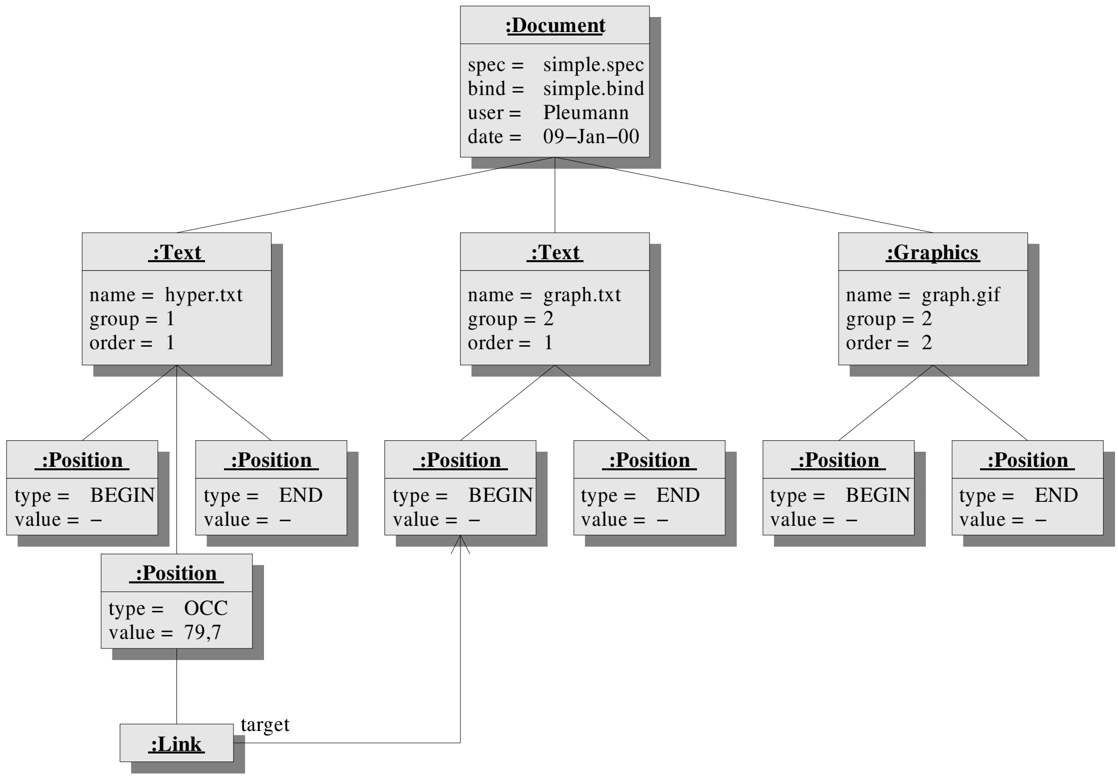

Abbildung 5.11.: Beispieldokument nach der Normalisierung

5.6.1. Implementierung

Zum Überführen der Repräsentation in die beschriebene Normalform ist bereits in der Basisklasse `Node` eine virtuelle Methode `normalize()` implementiert. Diese Methode ruft zunächst wiederum `normalize()` auf untergeordneten Knoten auf, führt also einen rekursiven Abstieg auf dem Objektbaum durch. Anschließend sortiert sie die Kindknoten mit Hilfe eines Sortieralgorithmus, der sich einer geschützten Methode `compare()` zum Schlüsselvergleich bedient. Diese Methode gibt einen Wert kleiner als, gleich oder größer als Null zurück, je nachdem in welcher Relation die beiden zu vergleichenden Knoten stehen. Die Klassen `Document` und `DBUnit` überschrieben diese Methode so, daß die oben aufgeführten Bedingungen erfüllt sind. Für alle anderen Klassen ist die Reihenfolge der Kindknoten nicht von Bedeutung, so daß sie die Standardimplementierung von `compare()` nutzen können, die einfach die gegebene Reihenfolge beibehält.

Die Normalisierung der Repräsentation wird angestoßen, indem `normalize()` für die Wurzel des Objektbaums, also das Hyperdokument selbst aufgerufen wird. Abbildung 5.11 zeigt das bereits bekannte Beispieldokument nach der Überführung in Normalform.

5.7. Erzeugen der Ausgabedateien

Abgeschlossen wird die Arbeit des Generators im vierten und letzten Schritt durch das Erzeugen der HTML-Seite(n). Nach den umfangreichen Vorbereitungsschritten fällt die eigentliche Ausgabe relativ unkompliziert aus. Tatsächlich beschränkt sie sich im wesentlichen auf einen kompletten Tiefendurchlauf des Objektbaumes und das gleichzeitige Schreiben von entsprechendem HTML-Code für die einzelnen Knoten. Trotzdem soll der Ausgabeteil des Generators natürlich nicht *ad hoc* implementiert werden.

5.7.1. Allgemeine und spezielle Teile der Ausgabe

Bereits in den einleitenden Kapiteln der Arbeit wurde betont, daß die leichte Erweiterbarkeit des Systems um neue Zielformate ein wesentlicher Aspekt ist, der beim Design Berücksichtigung finden soll. Dem muß die Architektur Rechnung tragen, indem sie versucht, die allgemeinen Teile der Ausgabe, die für alle Zielformate identisch sind, möglichst klar von denen zu trennen, die spezifisch für ein Zielformat – in diesem Fall HTML – sind. Naheliegenderweise soll der allgemeine Teil in einer Basisklasse realisiert werden und sich auf leere oder abstrakte Methoden abstützen, die von einer spezialisierten Nachkommenklasse HTML-spezifisch implementiert werden. Es stellt sich nun die Frage nach der Aufteilung der Ausgabelogik in allgemeine und HTML-spezifische Teile.

Zum allgemeinen Teil gehört sicherlich das grundsätzliche Traversieren der Baumstruktur in einer geeigneten Reihenfolge. Bei dieser Reihenfolge handelt es sich um einen Tiefendurchlauf, wobei sich nicht klar zwischen einem Präfix-, Infix- und Postfix-Durchlauf unterscheiden läßt. Dies ist stark von der Klasse des aktuell besuchten Knotens abhängig. Insbesondere auf der Ebene der Medienobjekte kann das Traversieren eine recht komplexe Angelegenheit werden, wie man sich an Abbildung 5.12 klarmachen kann: Bei einem textuellen Medienobjekt etwa müssen abwechselnd Positionen und Textstücke zwischen diesen Positionen in die Ausgabe geschrieben werden. Die Textstücke müssen zuvor anhand des „Wertes“ der sie umschließenden Positionen aus dem gesamten Text des Medienobjektes bestimmt werden.

Das konkrete Schreiben von HTML-Code für Positionen (im HTML-Kontext *Anker*), Verweise oder etwa Grafiken hingegen ist sicherlich eine Aufgabe, die in die HTML-spezifische Klasse gehört. Auch die Behandlung von einfachem Text gehört an diese Stelle, weil die Codierung zum Beispiel der Umlaute oder bestimmter anderer Sonderzeichen vom Zielformat abhängig ist – die spezialisierte Nachkommenklasse muß also eine Möglichkeit anbieten, den reinen ASCII-Text eines Medienobjekts korrekt in das Zielformat zu übersetzen.

Es gibt jedoch auch Teile, bei denen die Zuordnung zu einer der beiden Klassen nicht so einfach ist oder doch zumindest etwas besser überlegt sein will: Dazu zählt zum Beispiel die Verwaltung der Ausgabedateien, also das Öffnen und Schließen dieser Dateien. Man könnte versucht sein, diese Operationen mit dem Anfang und dem Ende einer

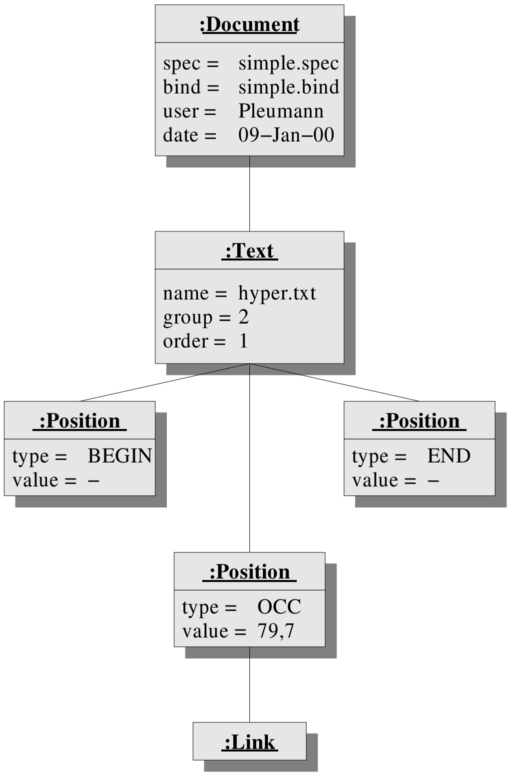

Abbildung 5.12.: Linker Hauptast des normalisierten Beispieldokumentes

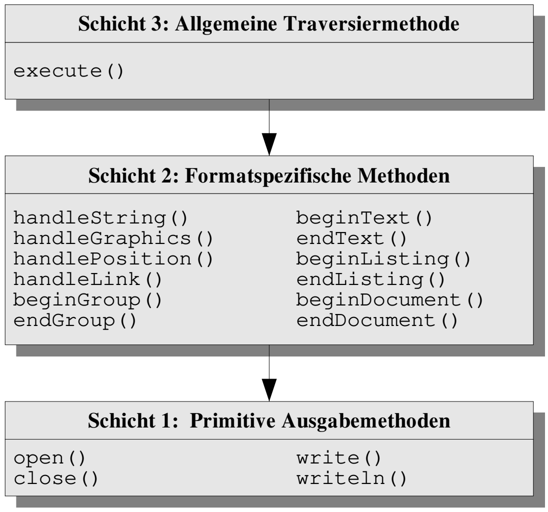

Abbildung 5.13.: Aufbau der Ausgabeklasse

Gruppe zu identifizieren und fest in der Basisklasse zu implementieren. Dies wäre allerdings eine etwas kurzsichtige Entscheidung, denn der direkte Zusammenhang zwischen Gruppen und Ausgabedateien ist eine spezifische Eigenschaft des HTML-Generators. Ein LaTeX-Generator würde stattdessen nur eine einzige große Datei erzeugen, dafür aber pro Gruppe zum Beispiel ein Kapitel anlegen. Eine Variante des HTML-Generators könnte ähnlich verfahren, die Ausgabe in einer Datei sammeln, aber einzelne Gruppen durch Überschriften oder horizontale Trennlinien voneinander absetzen. Es zeigt sich also, daß sich der Begriff der Gruppe, der ja nicht ohne Grund so abstrakt gewählt wurde, nicht gleichbedeutend mit dem einer Seite ist. Folglich ist es nicht sinnvoll, das Öffnen und Schließen von Ausgabedateien bereits zum Teil des fixen Verhaltens der Basisklasse zu machen.

5.7.2. Eine allgemeine Ausgabeklasse

Aus diesen Überlegungen ergibt sich für die allgemeine Ausgabeklasse `Writer` eine dreischichtige Architektur, wie sie in Abbildung 5.13 dargestellt ist. Jede der Schichten stützt sich ausschließlich auf Funktionalität ab, die ihr von der unmittelbar darunterliegenden Schicht bereitgestellt wird:

- Die untere Schicht enthält primitive Methoden zum Öffnen und Schließen der aktuellen Ausgabedatei sowie zum Schreiben einer Zeichenkette in diese Ausgabedatei. Das eigentliche Dateiobjekt, das diesen Operationen zugrunde liegt, wird durch die Methoden gekapselt. Die Schicht wird naheliegenderweise bereits in der Basisklasse implementiert. Die Methoden sind aber alle geschützt, da sie nur für die Entwicklung von spezialisierten Nachkommen von Belang sind.

- Die mittlere Schicht enthält die Methoden, deren Implementierung vom Zielformat abhängig ist. Hier finden sich also die erwähnten Operationen zur Behandlung von Positionen, Verweisen oder Grafiken wieder, aber auch die zur Konvertierung von ASCII-Text in HTML-codierten Text benötigte Methode. Es existieren auch zwei Methoden, die aufgerufen werden, wenn eine Gruppe beginnt oder endet. Wie darauf reagiert wird, liegt im Verantwortungsbereich der spezialisierten Nachfahrenklasse. Andere Methoden leiten ein Medienobjekt ein oder schließen es ab. Auch die Methoden der mittleren Schicht sind alle geschützt, und genau diese Methoden sind es, die für jedes neue Zielformat überschrieben werden müssen.

- Die obere Schicht enthält nur die Methode `execute()`, die als Argument eine Instanz von `Document` erwartet, also den Wurzelknoten des Objektbaums. Sie führt die angedeuteten Schritte zum Traveriseren des Baums durch und ruft an den entsprechenden Stellen Methoden aus der mittleren Schicht auf, um tatsächlich Zielcode zu generieren. Die Methode `execute()` ist als einzige öffentlich. Sie wird aufgerufen, um das Erzeugen der Ausgabe aus dem Hauptteil des Generators anzustoßen.

5.7.3. Beispiel

Abbildung 5.14 zeigt die Arbeitsweise der allgemeinen Ausgabeklasse, indem jedem Knoten eines Dokumentbaumes die Methodenaufrufe der mittleren Schicht zugeordnet sind, die sich aus ihm ergeben, sowie die Reihenfolge, in der diese Aufrufe stattfinden. Das bereits bekannte Beispiel wurde dazu erneut eingeschränkt, in diesem Fall auf den linken Hauptast der normalisierten Darstellung aus Abbildung 5.11, damit die Baumstruktur mit den zusätzlichen Informationen Platz auf einer Seite findet.

Die Abbildung ist beginnend von der Wurzel entgegen dem Uhrzeigersinn zu lesen: In Schritt 1 wird zunächst für den Wurzelknoten die Methode `beginDocument()` aufgerufen. Dies ist stets der erste Aufruf, der während des Traversierens erfolgt. Anschließend wird das erste Medienobjekt besucht, das zwangsläufig den Beginn einer neuen Gruppe und somit einen Aufruf von `beginGroup()` zur Folge hat, wie Schritt 2 zeigt.

Da es sich um ein textuelles Medienobjekt handelt, wird es in Schritt 3 zudem durch einen Aufruf von `beginText()` eingeleitet. Anschließend werden in den Schritten 4 bis 9 alle Positionen dieses Medienobjektes, deren Reihenfolge ja durch die Normalisierung bereits korrekt ist, durchlaufen, und für jede Position wird die Methode `handlePosition()` aufgerufen. Zwischen je zwei dieser Positionen wird mit `handleString()` der entsprechende Text ausgegeben, in Schritt 7 wird außerdem die Methode `handleLink()` aufgerufen, um einen Verweis in die Ausgabedatei zu schreiben.

Die abschließenden Schritte 10 bis 12 rufen die Gegenstücke der Methoden auf, die in den Schritten 1 bis 3 aufgerufen wurden. Sie schließen also das Medienobjekt, die Gruppe und das gesamte Hyperdokument ab.

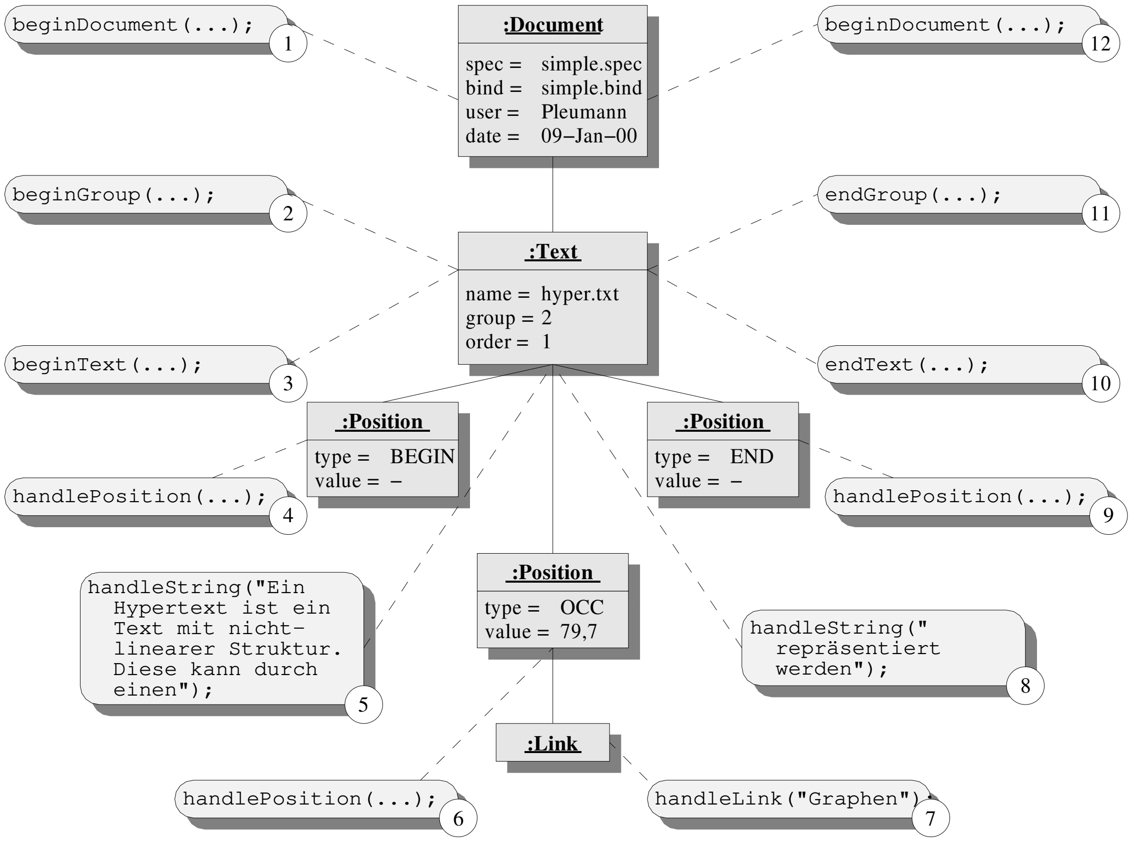

Abbildung 5.14.: Aufgerufene Methoden für das Beispieldokument

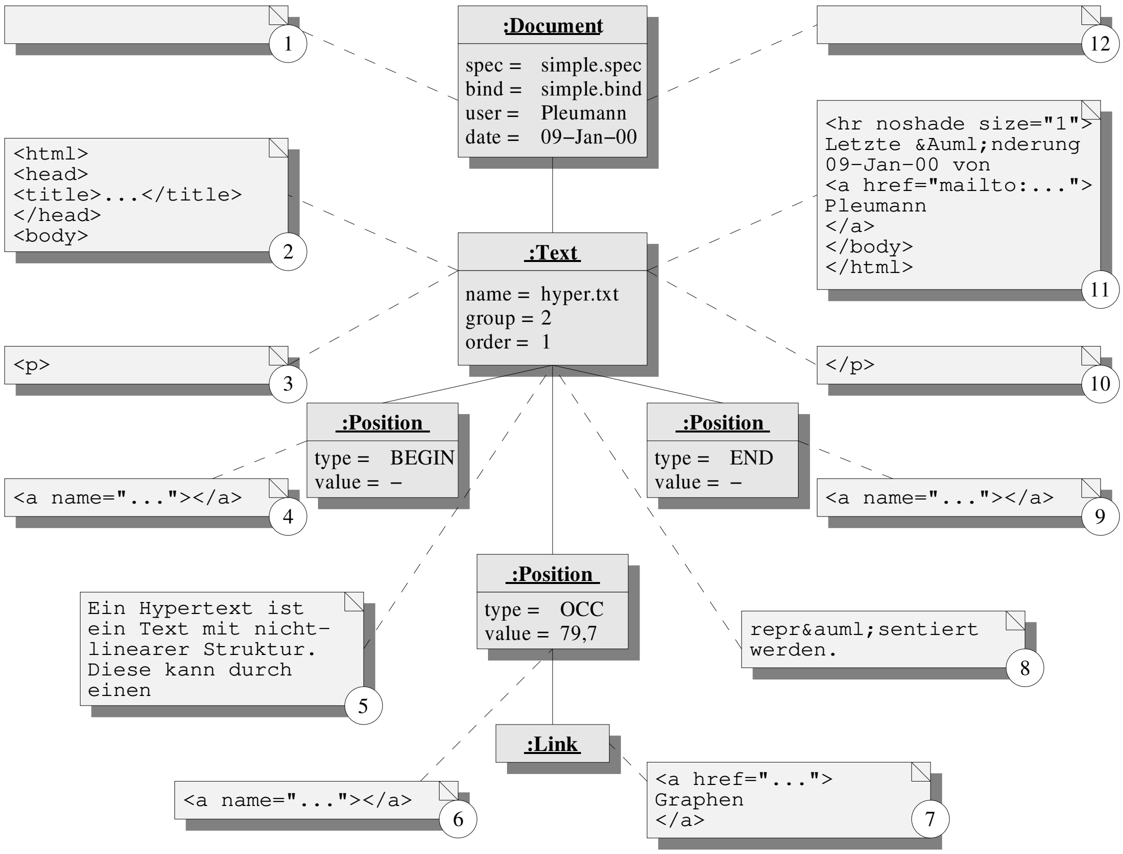

Abbildung 5.15.: Erzeugter HTML-Code für das Beispieldokument

Dies sind zunächst nur die reinen Methodenaufrufe der mittleren Schicht, sozusagen die *Infrastruktur* der Ausgabeklasse. Welcher Zielcode daraus folgt, ob eine Methode überhaupt Zielcode erzeugt und welche anderen Seiteneffekte sie besitzt, ist abhängig von dem spezialisierten Nachkommen, der für die Ausgabe von HTML-Code zuständig ist.

5.7.4. Eine spezialisierte Ausgabeklasse für HTML

Die Klasse `HTMLWriter` ist Nachkomme von `Writer`. Sie überschreibt die zuvor leeren Methoden der zweiten Schicht so, daß HTML-Code erzeugt wird. Abbildung 5.15 zeigt, welche Ausgabe aus den einzelnen Aufrufen resultiert. Diese Abbildung ist auf die gleiche Weise zu lesen wie Abbildung 5.14.

Es fällt auf, daß in den Schritten 1 und 12 überhaupt kein HTML-Code erzeugt wird. Grund dafür ist die Tatsache, daß der HTML-Generator eine Datei pro Gruppe erzeugen soll. Der übliche Kopf einer HTML-Datei wird also erst beim Beginn der Gruppe in

Schritt 2 erzeugt, wo auch die neue Ausgabedatei geöffnet wird. Das Schließen der Datei fällt entsprechend mit dem Ende der Gruppe in Schritt 11 zusammen, wo zusätzlich die Attribute `user` und `date` des Dokumentes verwendet werden, um eine Fußzeile für die Seite zu generieren.

In diesen vier Schritten zeigt sich sehr deutlich, daß die Basisklasse `Writer` mehr Infrastruktur (in Form von Methoden) bereitstellt, als tatsächlich im Fall von HTML benötigt wird. Jedoch sollte man nicht den Fehler machen, die Methoden `beginDocument()` und `endDocument()` deswegen für überflüssig zu erklären, denn für ein Zielformat wie zum Beispiel LaTeX sind sie sehr wohl notwendig: Hier würden genau diese Methoden zum Öffnen und Schließen der Ausgabedatei dienen, `beginGroup()` würde ein neues Kapitel erzeugen, und `endGroup()` würde unbenutzt bleiben.

Der HTML-Code, der von den restlichen Methodenaufrufen in den Schritten 4 bis 9 erzeugt wird, ergibt sich mehr oder minder intuitiv aus dem, was die einzelnen Knoten des Baums repräsentieren. So umschließen zum Beispiel die Methoden `beginText()` und `endText()` ein Medienobjekt des Typs `Text` mit den HTML-Elementen `<p>` und `</p>`, damit sich daraus ein deutlich sichtbarer Absatz ergibt. Der eigentliche Inhalt gelangt durch Aufrufe der Methode `handleString()` in die Ausgabedatei, wobei diese Methode sich hauptsächlich um die Konvertierung der nicht darstellbaren oder nicht erlaubten Zeichen kümmert. Schritt 8 verdeutlicht dies. Unterbrochen wird die Ausgabe des Textes von Aufrufen der Methode `handlePosition()`, die für jede Position ein HTML-Element `<a name="..."></a>` erzeugt. Die Methode `handleLink()` erzeugt zudem bei jedem Verweis direkt hinter dessen Ausgangsposition die HTML-Elemente `<a href="...">` und `</a>`, die den anklickbaren Text des Verweises umschließen.

Für Medienobjekte des Typs `Listing`, die im Beispiel nicht enthalten sind, gilt im wesentlichen das gleiche wie für `Text`, jedoch werden sie durch die HTML-Elemente `<pre>` und `</pre>` umschlossen, um die Darstellung in einer Schriftart mit fester Breite und den Erhalt der Zeilenumbrüche zu erreichen. Medienobjekte des Typs `Graphics` werden über das HTML-Element `<img src="...">` in die Ausgabedatei eingebettet.

5.7.5. Beispiel

Abbildung 5.16 zeigt abschließend, wie die generierten HTML-Seiten in einem entsprechenden Browser angezeigt werden. Die Pfeile sollen verdeutlichen, welcher der insgesamt drei verbliebenen Verweise zu welcher Seite führt. Die Verweise, die sich im Fuß jeder Seite befinden, rufen unmittelbar ein Fenster auf, über welches dem Autor des Dokumentes eine Nachricht geschickt werden kann.

5.7.6. Eine spezialisierte Ausgabeklasse für LaTeX

Die Klasse `LaTeXWriter` ist weiterer Nachkomme von `Writer`, der eine Variante des Hyperdokumentes erzeugt, die zum Ausdrucken geeignet ist. In `LaTeXWriter` werden die

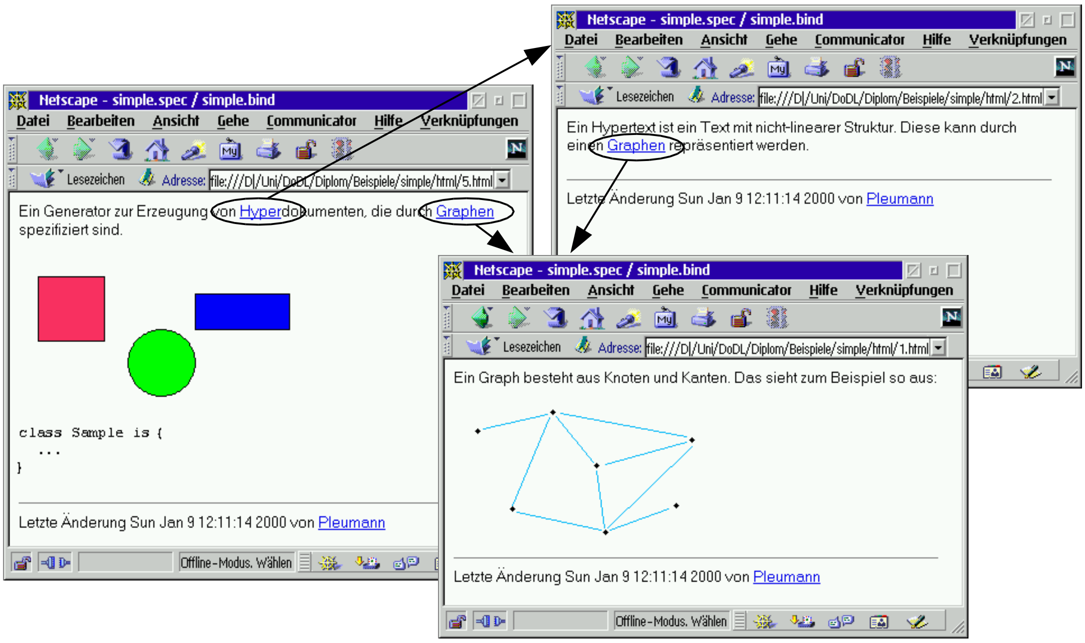

Abbildung 5.16.: Das fertige Beispieldokument für HTML

Methoden der zweiten Schicht so überschrieben, daß Code erzeugt wird, der mit LATEX und `dvips` in eine Postscript-Datei übersetzt werden kann. Abbildung 5.17 zeigt anhand des bekannten Beispiels, welche Ausgabe aus den einzelnen Aufrufen resultiert. Diese Abbildung ebenfalls zu lesen wie Abbildung 5.14.

Wie bereits mehrfach angedeutet, hebt sich der Generator für LATEX in der Behandlung von Gruppen und Dateien von dem Generator für HTML ab. Da für LATEX nur eine große Datei benötigt wird, erzeugen die Methoden `beginDocument()` und `endDocument()` deren Anfang und Ende, während `beginGroup()` nur einen neuen Abschnitt (eine `\section{}`) in dieser Datei einleitet und `endGroup()` komplett leer bleibt.

Die deutlich sichtbare Abgrenzung jedes Medienobjektes des Typs `Text` wird von den Methoden `beginText()` und `endText()` erzeugt, indem vor und nach dem Medienobjekt eine Leerzeile eingefügt wird, was LATEX als neuen Absatz interpretiert. Die Methode `handleString()` führt eine LATEX-spezifische Zeichenkonvertierung durch, tut aber im Prinzip das gleiche wie im Fall von HTML. Die Methoden `handlePosition()` und `handleLink()` erzeugen `\label{...}` und `\ref{...}`, wobei der Verweis als textueller Querverweis der Art „`(siehe auch ...)`“ realisiert wird.

Für den im Beispiel nicht enthaltenen Medientyp `Listing` gilt wieder ähnliches wie im Fall der Klasse `HTMLWriter`: Er wird in eine `verbatim`-Umgebung eingebettet, so daß er wie gewünscht dargestellt wird. Eine wirkliche Besonderheit verbirgt sich nur in der Implementierung der Methode `handleGraphics()`. Da LATEX bzw. `dvips` nicht in der Lage ist, beliebige Grafikformate zu importieren, ruft diese Methode zunächst ein Konvertie-

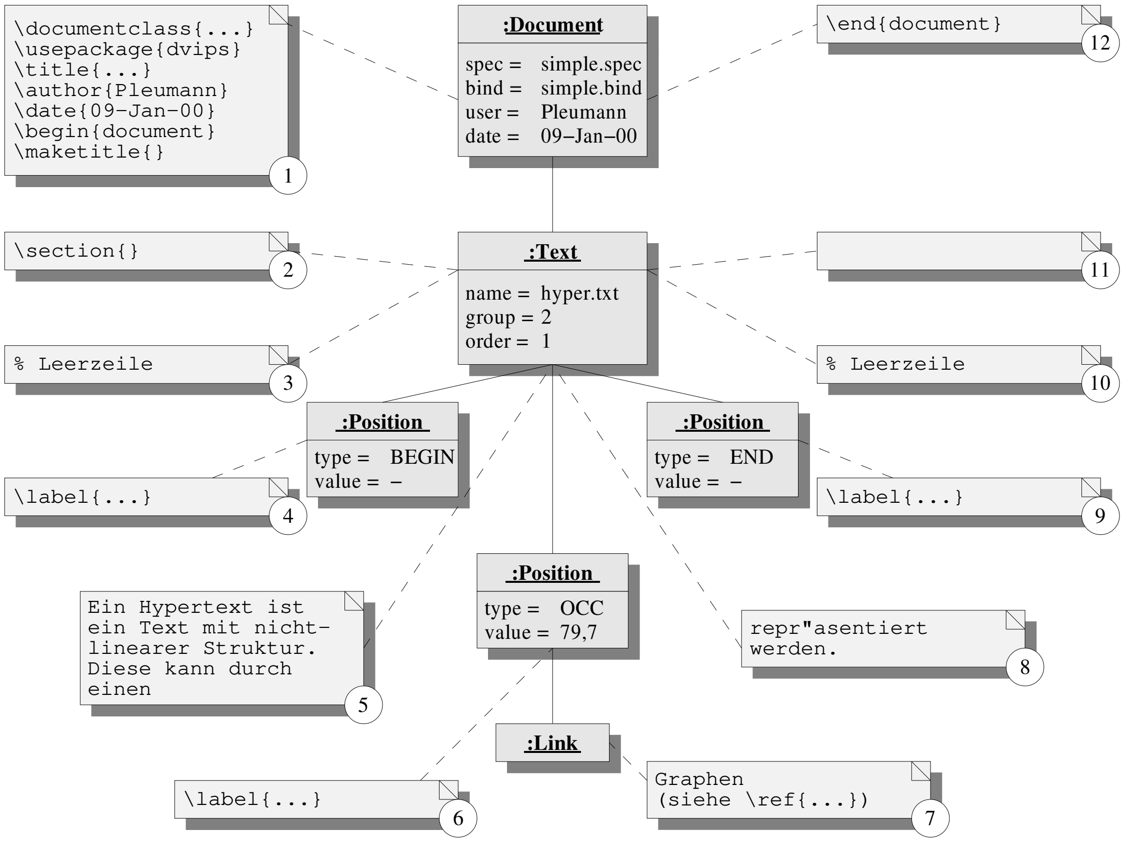

Abbildung 5.17.: Erzeugter LaTeX-Code für das Beispieldokument

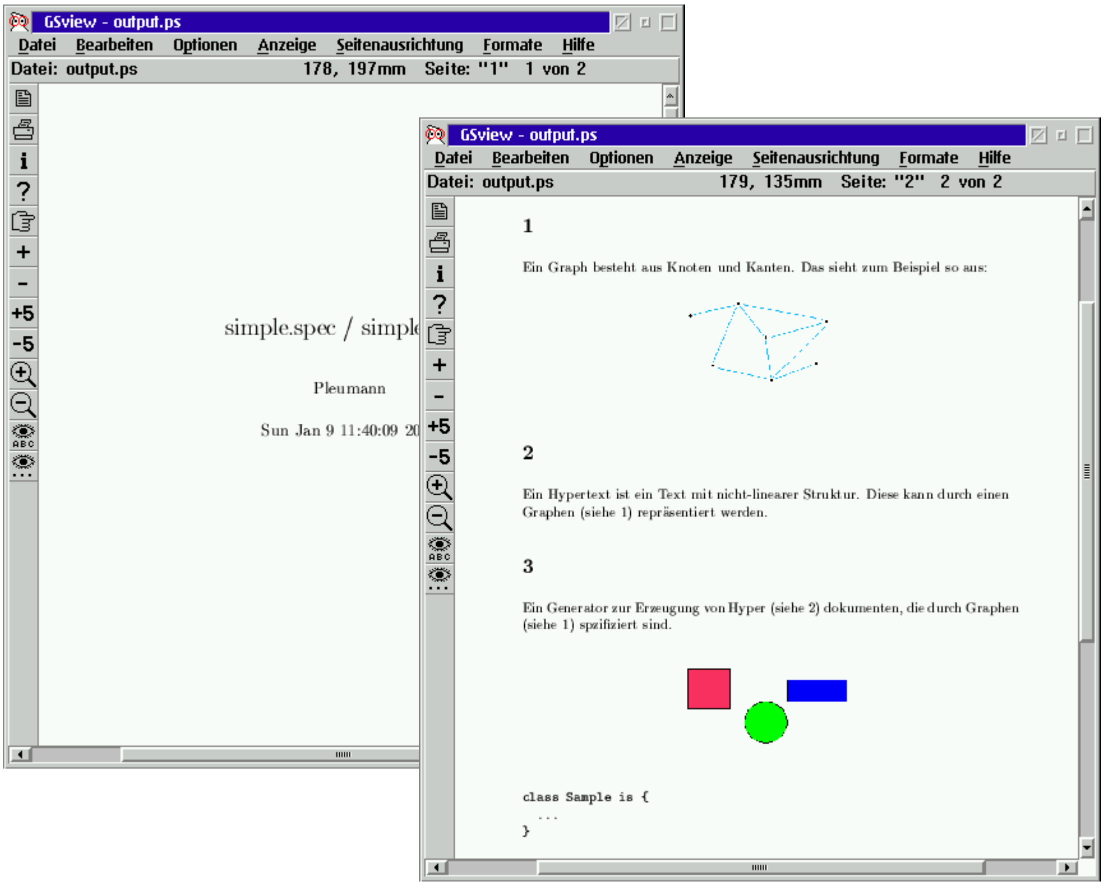

Abbildung 5.18.: Das fertige Beispieldokument für LaTeX

rungswerkzeug (z.B. `gif2eps`) auf, um die Grafik in eine Postscript-Datei umzuwandeln, die dann problemlos verwendet werden kann.

5.7.7. Beispiel

Abbildung 5.18 zeigt, daß auch der LaTeX-Generator funktioniert. Es wurde ein Dokument erzeugt, das aus einer Titelseite und einer weiteren Seite besteht. Die Titelseite enthält die bekannten Informationen über Autor, Datum und Ursprungsdateien. Die andere Seite enthält die erwarteten drei Abschnitte, in denen sich die einzelnen Gruppen wiederfinden. Die Verweise fordern den Leser auf, an der gewünschten Stelle weiterzulesen, und auch das automatische Konvertieren der Grafik hat funktioniert.

5.8. Die Summe der Teile

Die Abbildungen 5.19 und 5.19 zeigen einen kompletten Entwurf des Generators. Sämtliche Klassen, die in diesem Kapitel vorgestellt wurden, sind darin enthalten, ebenso wie einige wichtige Klassen aus der XML-Bibliothek und der Java-Laufzeitbibliothek. Alle Attribute und Methoden sind mit ihrer Sichtbarkeit aufgeführt, jedoch wurde aus Platzgründen auf die Angabe von Typen und Parametern verzichtet.

Die Abbildungen enthalten eine bislang nicht erwähnte Klasse `Error`, auf die alle zur Laufzeit auftretenden Ausnahmen abgebildet werden. Da `Error` ein Nachkomme der Java-Klasse `RuntimeException` ist, braucht diese Klasse nicht in der Liste der möglichen Ausnahmen einer Methode aufgeführt zu werden, was die Fehlerbehandlung erheblich vereinfacht (obwohl es zugegebenermaßen nicht ganz der Philosophie von Java entspricht). Unterschlagen wurde in der Abbildung eine lokale Hilfsklasse `UnresolvedLink`, die von der Klasse `Loader` zur Auflösung der Verweise während des Lesens der XML-Datei benutzt wird. Außerdem fällt auf, daß an einigen Stellen von Abbildung 5.19 Aggregationen mit dem Namen einer Methode markiert sind. Diese Methoden dienen zum Zugriff auf die von `Node` geerbte Aggregation `children` mit dem korrekten Typ, wie es in Abschnitt 5.3.1 beschrieben wurde.

Die gestrichelten Rechtecke verdeutlichen, in welchen Java-Paketen die einzelnen Klassen beheimatet sind. Das Paket `dodl.core` mit der einzelnen Klasse `Document` im unteren Bereich von Abbildung 5.20 steht stellvertretend für die komplette Abbildung 5.19. Die beiden Klassen `dodl2html` und `dodl2latex` im Paket `dodl` stellen jeweils das vom Benutzer aufzurufende „Hauptprogramm“ dar.

5.9. Alternativen

Mit Blick auf die gewünschten Eigenschaften des Generators, speziell die leichte Erweiterbarkeit, ist die Zahl der Alternativen, die sich substantiell von der vorgestellten Lösung unterscheiden, statt nur Variationen des gleichen Themas zu sein, begrenzt. Die intuitive Modellierung der *DoDL*-Begriffe durch Java-Klassen bietet sich ebenso an wie die Kapselung der Arbeitsschritte in einzelnen Klassen. Die vorgestellte Lösung scheint auch hinreichend effizient zu sein: Es werden drei Durchläufe durch die Datenstruktur benötigt, ein „echter“ Durchlauf des Graphen zum Bilden der Gruppen und zwei Tiefendurchläufe des gesamten Baums für Normalisierung und Ausgabe. Hinzu kommt die Sortierung der Knoten während der Normalisierung.

Eine echte Alternative in der Implementierung bietet sich eigentlich nur bei der Wahl der verwendeten XML-Bibliothek und dem enthaltenen Parser. Zwischen den beiden Bibliotheken, die zur Auswahl standen, waren vom Leistungsumfang keine großen Unterschiede sichtbar, so daß letztlich die Bibliothek von Sun den Vorzug gegenüber der von IBM [IBM99] erhielt, weil sie gut dokumentiert ist und eine unkomplizierte Lizenzvereinbarung beinhaltet. Diese Bibliothek hätte zwar auch die Möglichkeit geboten, einen

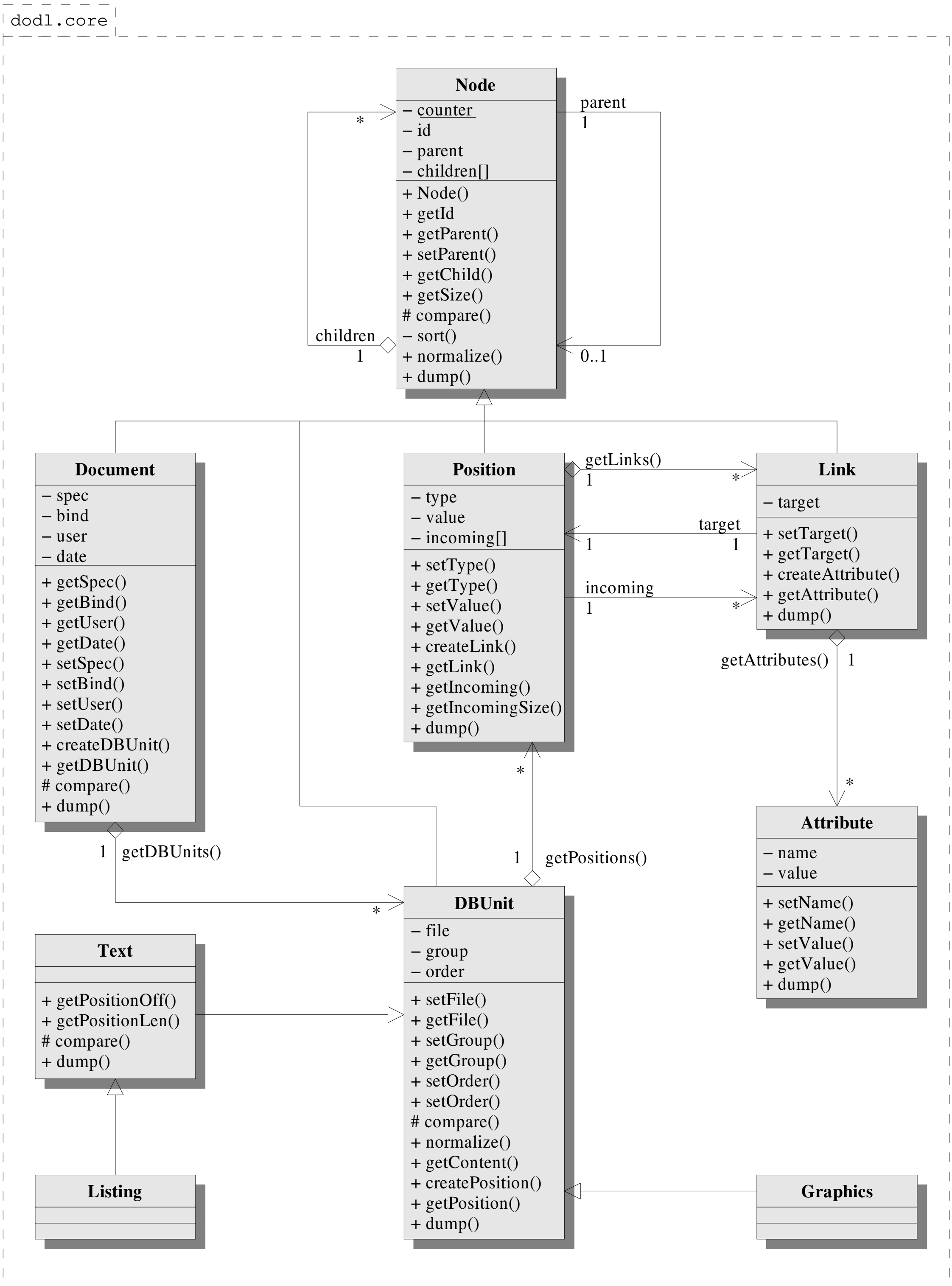

Abbildung 5.19.: Entwurf des Generators (Teil 1 von 2)

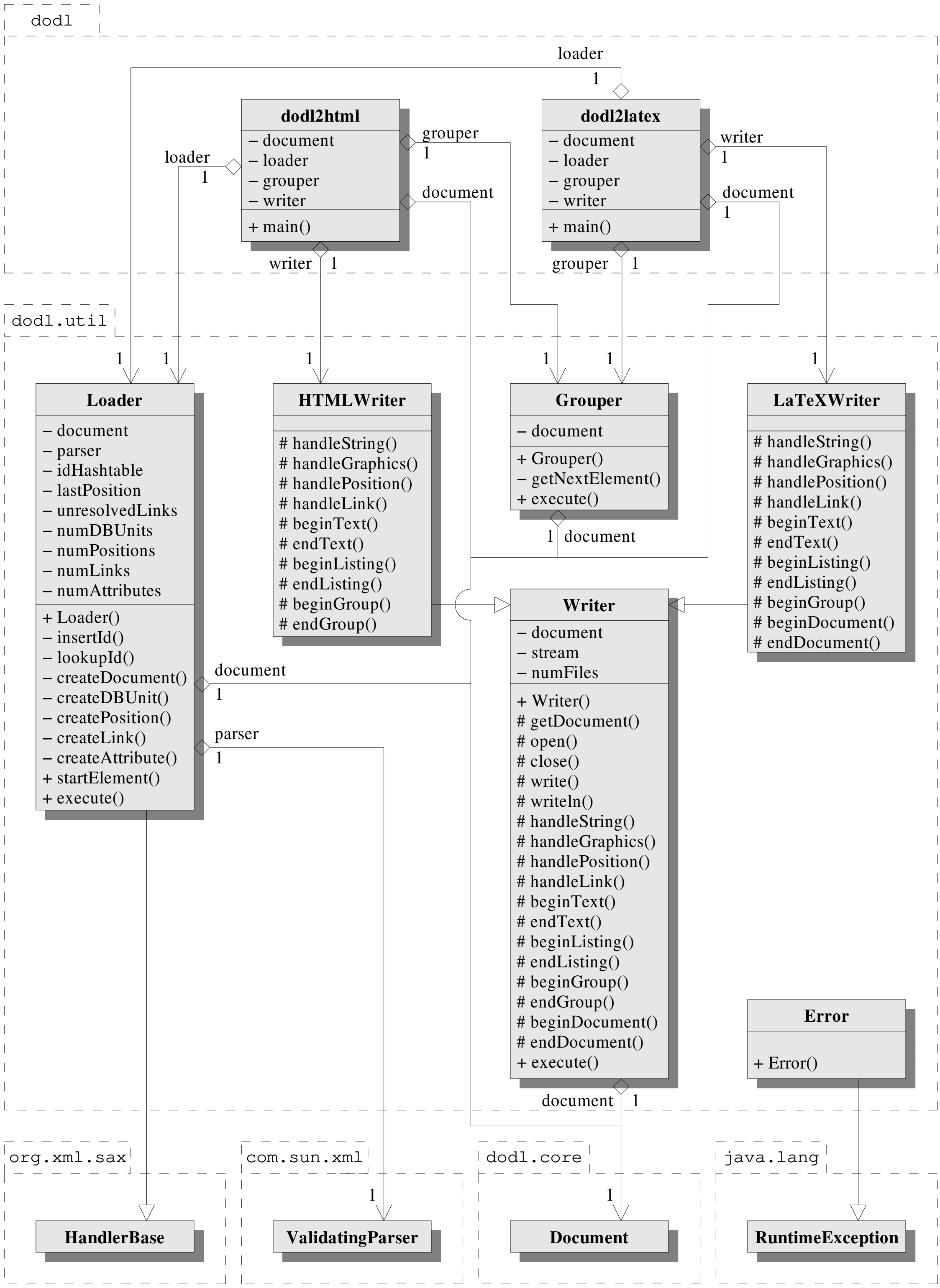

Abbildung 5.20.: Entwurf des Generators (Teil 2 von 2)

anderen Parser zu verwenden, der während der syntaktischen Analyse automatisch eine dem *Document Object Model (DOM)* [W3C98b] entsprechende Datenstruktur aufbaut. Jedoch wäre diese Struktur zu unhandlich gewesen, um daraus direkt HTML-Seiten zu erzeugen, so daß entweder das Traversieren sehr kompliziert ausgefallen wäre oder eine Überführung in die bereits bekannte Datenstruktur nötig gewesen wäre. Angesichts dessen schien es sinnvoller, den ereignisorientierten Parser zu verwenden und unmittelbar die eigene Datenstruktur aufzubauen.

5.10. Erweiterungen

Erweiterungen für den Generator sind in vielfältiger Form möglich, wobei hier die Unterstützung zusätzlicher Ausgabeformate natürlich besonders naheliegt. Als erstes Beispiel wurde deshalb bereits im Rahmen der Diplomarbeit der zweite Generator für LaTeX entwickelt. Die wenigen Zeilen Quellcode, die dafür nötig sind, verdeutlichen, wie schnell Generatoren für andere Formate implementiert sind.

Auch das leichte Hinzufügen neuer Medientypen wurde bereits mehrfach als Vorteil des Systems herausgestellt. Es wäre also denkbar, weitere Typen, etwa für Ton- oder Videodateien, in Form von spezialisierten Nachkommen von `DBUnit` zu definieren und die Ausgabeklassen entsprechend anzupassen. Allerdings sind diese Typen allesamt trivial, solange sie nur die Positionstypen `BEGIN` und `END` unterstützen, weil sie sich dann nicht wesentlich von `Graphics` unterscheiden: Letztlich resultieren sie in der Einbettung einer externen Datei in eine HTML-Seite. Erst bei der Unterstützung von Positionen des Typs `OCC` wird ein Medientyp wirklich interessant, weil dann beim Erzeugen der Ausgabe Ausschnitte des Medienobjektes mit dessen Positionen kombiniert werden müssen, wie das beim Typ `Text` der Fall ist. Dies erfordert allerdings eine gründliche theoretische Betrachtung und exakte Definition des Begriffs der Position für die entsprechenden Medientypen – ein Aufwand, der den Rahmen dieser Diplomarbeit übersteigt.

Eine andere Erweiterung im Bereich der Medientypen, die aber für den allgemeinen Hypermedia-Kontext relativ uninteressant ist, ist die Implementierung spezieller Nachkommen von `Listing` für einzelne Programmiersprachen. Der Generator könnte die Ausgabe dafür so aufbereiten, daß die Syntax der Sprache hervorgehoben wird. In Verbindung mit der ohnehin vorhandenen Möglichkeit, Vorkommen von Bezeichnern in Quellcode zu finden, könnte *DoDL* dann als System zur Hypertext-Dokumentation von Programmen benutzt werden. Es wäre ein leichtes, etwa alle Benutzungen einer bestimmten Prozedur oder Funktion mit Hilfe von `getOcc()` auf deren Deklaration verweisen zu lassen.

Auf der algorithmischen Seite ist es mit Sicherheit möglich, eine andere und vermutlich bessere Methode zur Gruppierung der Medienobjekte zu finden. Denkbar sind hier Lösungen, die sich auf graphentheoretische Betrachtungen stützen und etwa versuchen, den gesamten Graphen oder einzelne Zusammenhangskomponenten davon auf irgendeine Weise zu „plätten“. Allerdings setzt die Implementierung solcher Algorithmen wahrscheinlich eine erneute und etwas tiefergehende Betrachtung der verschiedenen Typen von Verweisen voraus. Die Ableitung der Reihenfolge allein aus Verweisen zwischen dem Ende eines und dem Beginn eines anderen Medienobjektes ist eine zugegebenermaßen sehr grobe Vereinfachung, da sie einen Großteil der vorhandenen Information außer acht läßt: Was ist zum Beispiel mit Verweisen, die vom Beginn eines Medienobjektes ausgehen, oder solchen, die vom Ende zum Vorkommen einer Zeichenkette führen? Es sollte auch möglich sein, aus solchen Verweisen eine Reihenfolge der Medienobjekte abzuleiten, vorausgesetzt es wird zuvor geklärt, wie diese Verweise eigentlich zu interpretieren sind. Nicht zuletzt stellt sich im Zusammenhang mit den Gruppen auch die bisher komplett vernachlässigte Frage, welche der Gruppen denn nun die ist, die dem Benutzer als Anfangs- oder Titelseite des Hyperdokumentes präsentiert wird.

Es gibt also offensichtlich genug Raum für Erweiterungen des bestehenden Systems, die allerdings allesamt den Rahmen der Diplomarbeit sprengen würden. Das nächste Kapitel zieht sich daher wieder auf das zurück, was bislang erreicht wurde, und zeigt einige Testläufe des Systems mit größeren Beispiel-Spezifikationen.

6. Test und Bewertung

Eine Lösung für ein gegebenes Problem zu implementieren ist gut und schön. Aber um den tatsächlichen Wert dieser Lösung abschätzen zu können, ist es notwendig, sie auf ihre Tauglichkeit für die Praxis zu untersuchen. Das minimale Hyperdokument, das stets als Beispiel herhalten mußte, reicht dazu sicherlich nicht aus. Dieses Kapitel unterzieht deshalb das in der Diplomarbeit implementierte und in den vorangehenden Kapiteln beschriebene Gesamtsystem aus *DoDL*-Laufzeitbibliothek, XML-basierter Zwischendarstellung und HTML-Generator einigen praxisnahen Tests und stellt die Ergebnisse dar. Aus Platzgründen werden jeweils nur Bildschirmphotos der HTML-Ausgabe gezeigt, obwohl alle Beispiel auch mit dem LaTeX-Generator erfolgreich getestet wurden.

6.1. Beispiel 1: Ein Hypermedia-Glossar

Das erste Beispiel ist unmittelbar dem Kontext der Diplomarbeit entlehnt. Es handelt sich dabei um ein kleines Glossar aus Begriffen, die dem Bereich Hypermedia entstammen. Jedes der beteiligten Medienobjekte enthält die Definition eines dieser Begriffe. Die Spezifikation sucht die Vorkommen des entsprechenden Begriffs in allen anderen Medienobjekten (außer dem, in dem der Begriff definiert ist) und erzeugt Verweise auf die Definition.

Das Glossar setzt sich aus insgesamt 16 Medienobjekten zusammen, von denen Abbildung 6.1 vier Stück zeigt. Die Abbildungen 6.2 und 6.3 zeigen die Spezifikation und die Bindungen, die das Glossar implementieren. Die Zwischendarstellung, die sich bei der Ausführung der übersetzten Spezifikation ergibt, hat bereits bei diesem kleinen Beispiel den beachtlichen Umfang von etwa 350 Zeilen und soll deshalb an dieser Stelle ausgespart werden.

Der HTML-Generator erzeugt aus den 16 Medienobjekten ebenso viele Gruppen, die jeweils in einer eigenen HTML-Datei plaziert werden. Die vier Dateien, die den bereits gezeigten Medienobjekten entsprechen, sind in Abbildung 6.4 dargestellt. Die Verweise bzw. die Positionen, von denen die Verweise ausgehen, sind durch die unterstrichene Darstellung im Browser deutlich erkennbar. Zur besseren Übersicht wurde darauf verzichtet, zusätzliche Pfeile, die das Ziel der Verweise angeben, in die Abbildung einzufügen.

Das Schöne an diesem Beispiel – abgesehen davon, daß es überhaupt funktioniert – ist die Tatsache, daß es eine elegante Implementierung des Glossars darstellt. Ein Blick auf

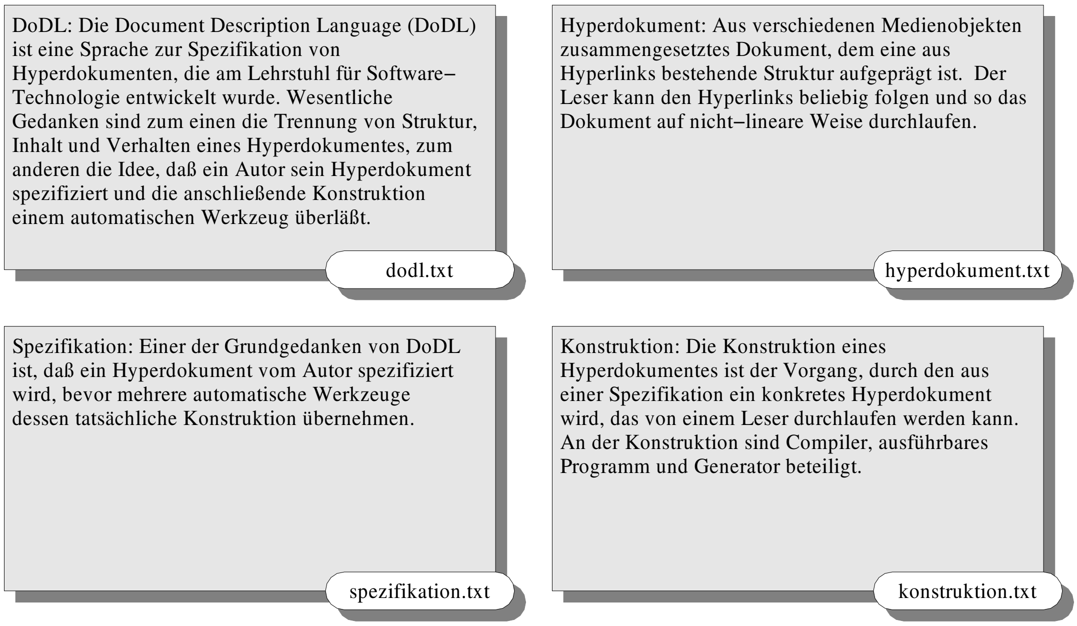

Abbildung 6.1.: Medienobjekte des Hypermedia-Glossars

Spezifikation und Bindungen zeigt, daß die Klasse `Entry` einen Eintrag des Glossars repräsentiert, der sich aus dem Stichwort und dem zugehörigen Medienobjekt zusammensetzt. Die Hauptklasse besitzt eine Liste solcher Einträge, deren Länge in der Spezifikation noch unbekannt ist, und erzeugt Verknüpfungen aller Stichwörter mit allen Vorkommen. Welche Stichwörter überhaupt vorhanden sind, ergibt sich erst aus den Bindungen. Das bedeutet insbesondere, daß allein durch Hinzufügen neuer Listenelemente in den Bindungen der Umfang des Beispiels vergrößert werden kann. Es ist nicht nötig, die eigentliche Spezifikation zu ändern oder auch nur zu kennen. Das kommt natürlich dem Autor des Glossars entgegen, denn ihm geht es sicherlich im wesentlichen um dessen Inhalt und nicht so sehr um die Realisierung.

Unschön an dem Beispiel ist, daß in vielen Fällen nur der Anfang eines Wortes zu einer Position und damit „anklickbar“ wird. Das liegt daran, daß die Methode `getOcc()` der Klasse `Text` nur exakte Vorkommen des Suchparameters findet. So ergibt zum Beispiel aus der Suche nach dem Stichwort „Hyperdokument“ die Position „Hyperdokumenten“ in der linken oberen HTML-Seite. Das ist zwar kein Fehler, denn das System arbeitet wie geplant, aber das Ergebnis ist doch etwas unschön. Es wäre also zum Beispiel darüber nachzudenken, ob `getOcc()` sein Suchergebnis nicht automatisch auf ganze Wörter ausdehnt oder, wie bereits in Abschnitt 3.8 angedacht, sogar nach regulären Ausdrücken anstelle fester Zeichenketten sucht.

```
class Glossary is Document with
  declare
    class Entry is Document with
      documents
        keyword : string ;
        content : Text;
    end Entry;

  documents
     entries : list of Entry;

  construct
    void main(void) {
      int i;
      int j;

      for (i = 0; i < size(entries); i++)
      {
        for (j = 0; j < size(entries); j++)
        {
          if (j != i)
          {
            entries[j].content.getOcc(entries[i].keyword).
              setLink(entries[i].content.getBegin());
          }
        }
      }
    }
end Glossary;
```

Abbildung 6.2.: Spezifikation des Hypermedia-Glossars

```
binding Glossary is
  in entries assign
       keyword = "Compiler";
       content = "source/compiler.txt";
     | keyword = "Datenbank";
       content = "source/datenbank.txt";
     | keyword = "Dexter";
       content = "source/dexter.txt";
     | keyword = "DoDL";
       content = "source/dodl.txt";
     | keyword = "Dokument";
       content = "source/dokument.txt";
     | keyword = "Generator";
       content = "source/generator.txt";
     | keyword = "Hyperdokument";
       content = "source/hyperdokument.txt";
     | keyword = "Hyperlink";
       content = "source/hyperlink.txt";
     | keyword = "Inhalt";
       content = "source/inhalt.txt";
     | keyword = "Konstruktion";
       content = "source/konstruktion.txt";
     | keyword = "Medienobjekt";
       content = "source/medienobjekt.txt";
     | keyword = "Position";
       content = "source/position.txt";
     | keyword = "Programm";
       content = "source/programm.txt";
     | keyword = "Spezifikation";
       content = "source/spezifikation.txt";
     | keyword = "Struktur";
       content = "source/struktur.txt";
     | keyword = "Verhalten";
       content = "source/verhalten.txt";
  end;
end;
```

Abbildung 6.3.: Bindungen des Hypermedia-Glossars

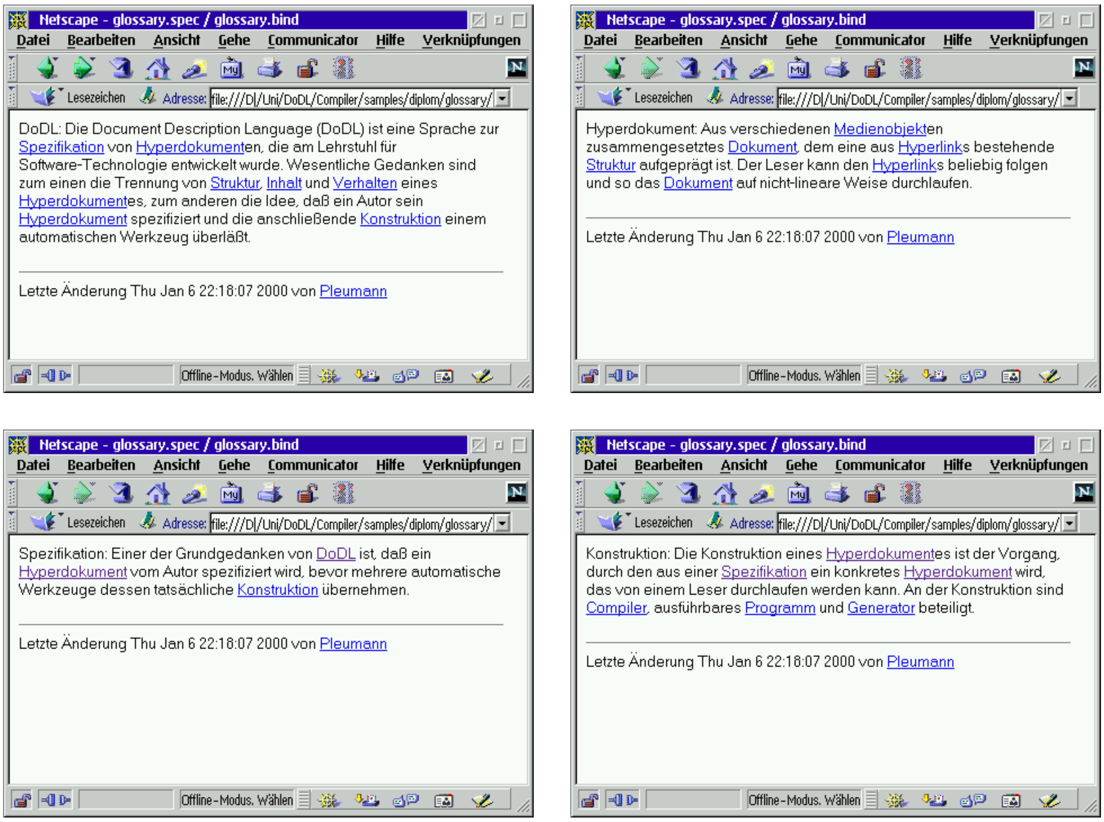

Abbildung 6.4.: HTML-Seiten des Hypermedia-Glossars

6.2. Beispiel 2: Ein Pizza-Service im Internet

Das zweite Beispiel realisiert die Web-Präsenz für einen imaginären Pizza-Service, der im Internet tätig ist. Sie führt den potentiellen Kunden durch das Angebot: Zunächst kann er eine Pizza-Sorte wählen, anschließend legt er den Weg fest, auf dem ihm die Pizza zugestellt wird. Abbildung 6.5 zeigt die Medienobjekte, die an diesem Hyperdokument beteiligt sind, sowie deren Verknüpfung.

Das Interessante an diesem Hyperdokument ist natürlich nicht der Inhalt, sondern die Struktur, die in einigen Punkten komplexer ist als jene, die bisher verwendet wurden. Im Gegensatz zu dem bekannten Minimal-Beispiel und dem Hypermedia-Glossar verwendet das Beispiel des Pizza-Service die folgenden Arten von Verweisen:

- Mehrere Verweise, die von einer Position ausgehen, die durch `getOcc()` gefunden wurde.
- Mehrere Verweise, die vom Ende eines Medienobjektes ausgehen, so daß dieses Medienobjekt keinen eindeutigen Nachfolger besitzt.
- Mehrere Verweise, die auf den Beginn eines Medienobjektes zeigen, so daß dieses Medienobjekt keinen eindeutigen Vorgänger besitzt.

Die Frage ist nun, wie das System mit diesen Verweisen umgeht. Es sind vor allem zwei Komponenten des Systems, die bezüglich der ungewöhnlichen Verknüpfungsstruktur besondere Aufmerksamkeit erhalten sollten: Zum einen könnte die Gruppierungs-Automatik fälschlicherweise mehrere Medienobjekte zu einer Gruppe zusammenfassen, obwohl keine zwei Medienobjekte des Beispiels die Kriterien für eine Gruppierung erfüllen (vergleiche dazu Abschnitt 5.5). Das Hyperdokument kann also als Testfall für korrekte Implementierung der Gruppierung gemäß der Definition angesehen werden. Zum anderen könnte es dem HTML-Generator Probleme bereiten, mehrere Verweise von einer Position, also *1:n*-Verweise, ausgehen zu lassen. Aus Platzgründen und auch, weil aus ihnen keine wesentlichen Erkenntnisse gewonnen werden können, sollen Spezifikation, Bindungen und Zwischendarstellung für dieses Beispiel unberücksichtigt bleiben. Stattdessen wird direkt das Endresultat betrachtet.

Abbildung 6.6, die Darstellung der erzeugten HTML-Seiten, zeigt, daß die geäußerten Bedenken unbegründet sind. Die Gruppierung hat offenbar korrekt funktioniert, denn jedes Medienobjekt befindet sich in einer eigenen Datei, also auch in einer eigenen Gruppe. Die Bildschirmphotos zeigen außerdem, wie der Generator mit den *1:n*-Verweisen umgeht: Er macht nicht die Position selbst, also zum Beispiel das Vorkommen einer gesuchten Zeichenkette, anklickbar. Stattdessen fügt er hinter der eigentlichen Position für jeden ausgehenden Verweis eine Zahl an, die anstelle des eigentlichen Textes angeklickt werden kann. Auf diese Weise kann er – zumindest theoretisch – mit unbegrenzt vielen ausgehenden Verweisen pro Position umgehen.

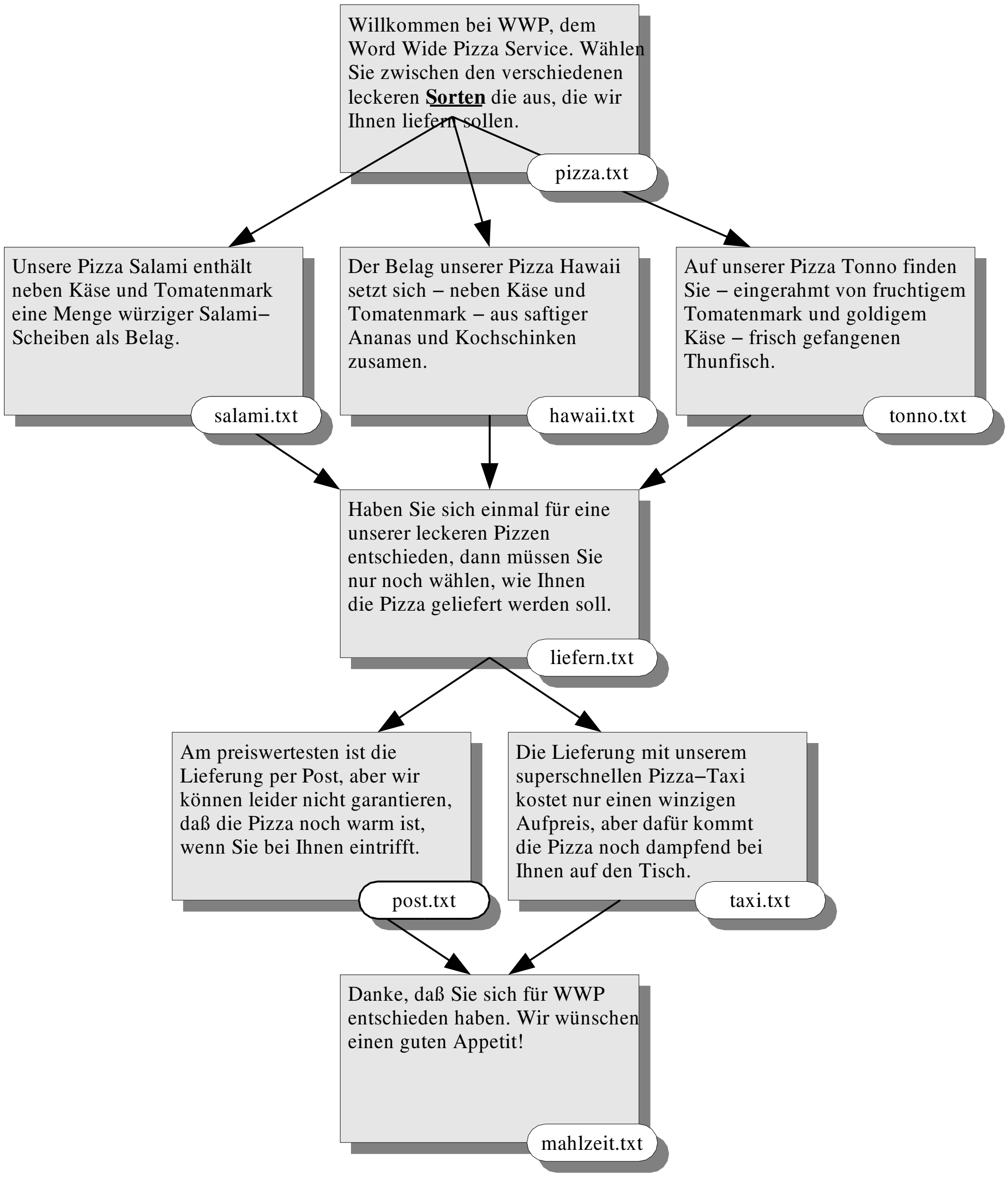

Abbildung 6.5.: Medienobjekte des *World Wide Pizza Service*

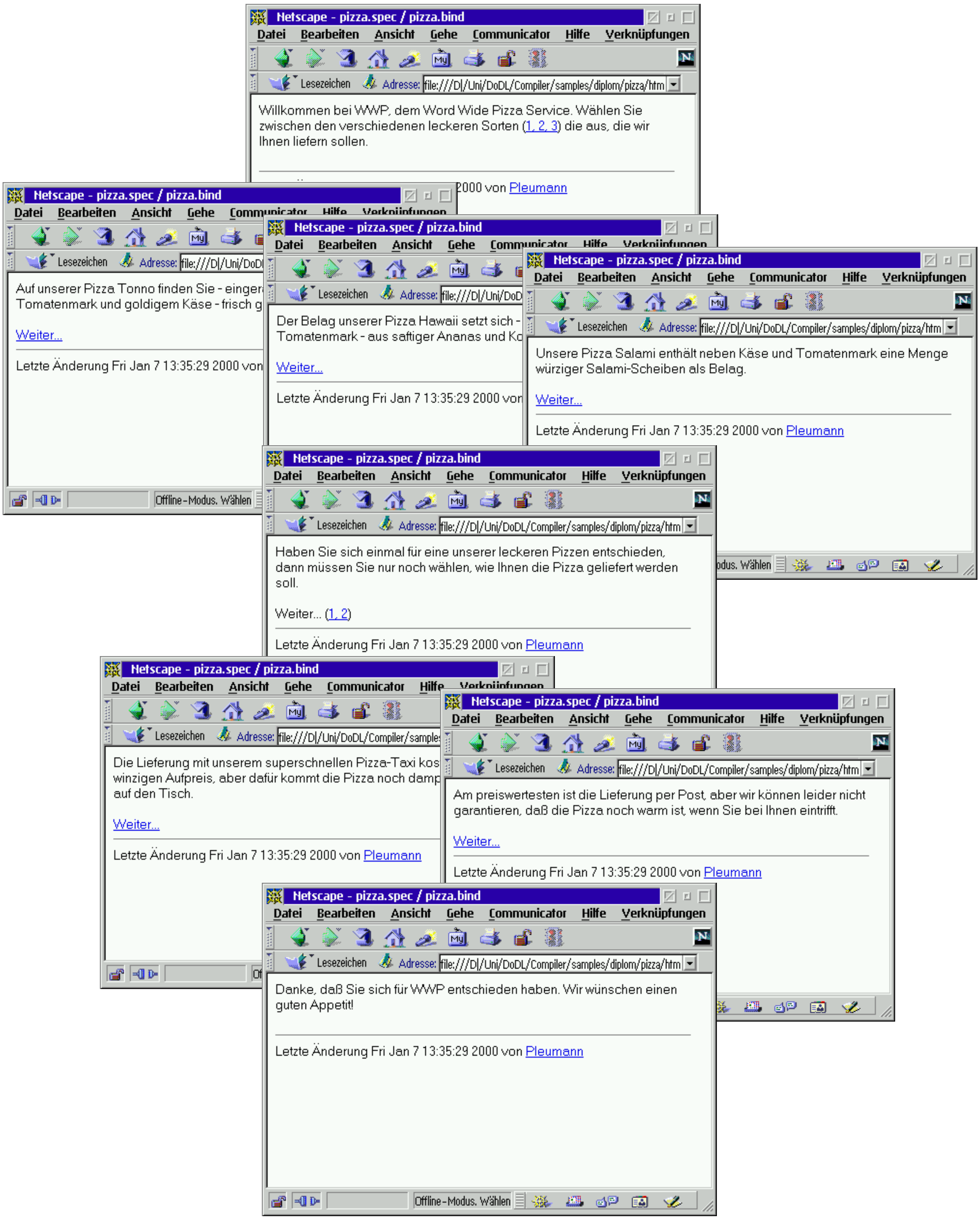

Abbildung 6.6.: HTML-Seiten des *World Wide Pizza Service*

Das Beispiel zeigt auch, wie der Generator automatisch den Text „Weiter...“ einfügt, wenn ein Verweis vom Ende einer Gruppe, also von deren letztem Medienobjekt ausgeht. Auch dies entspricht den Ideen, die in Abschnitt 5.5, der sich mit der Gruppierung beschäftigt, geäußert wurden. Unschön an dem Beispiel ist allerdings, daß die automatisch eingefügten Verweise „Weiter...“ und „1, 2, 3, ...“ keinen Aufschluß darüber geben, was sich hinter ihnen verbirgt. Dies ist vielleicht ein Punkt, der verbesserungswürdig ist.

6.3. Beispiel 3: Ein astrologisches Lexikon

Das dritte und letzte Beispiel implementiert ein Hypertext-Lexikon zum Thema Astrologie. Es setzt sich aus einer ganzen Reihe von kleinen Texten zusammen, die sich mit den Sternzeichen, den Planeten und deren Beziehung zueinander beschäftigen[1]. Von der Struktur her ähnelt das Lexikon dem Hypermedia-Glossar aus Abschnitt 6.1, jedoch ist es ungleich größer: Mit etwa 500 Medienobjekten und einem Gesamtumfang von fast 370 KB ist es schon allein deshalb ein interessanter Testfall, weil es dem *DoDL*-System abverlangt, mit einer sehr großen Datenmenge, die zum Beispiel mehrere tausend Positionen umfaßt, umzugehen.

Aufgrund des Umfangs sollen hier weder Spezifikation noch Bindungen, schon gar nicht die 250 KB große Zwischendarstellung, im Detail dargestellt werden. Auch für das Endergebnis, das sich aus 25 größeren HTML-Seiten zusammensetzt, soll nur mit Hilfe von Abbildung 6.7 ein Existenzbeweis erbracht werden, ohne daß jeder einzelnen Seite ein komplettes Bildschirmphoto gewidmet wird. Beachtung soll stattdessen in diesem Fall den Laufzeiten der beiden Programme geschenkt werden. Bei den anderen Beispielen waren diese Laufzeiten aufgrund des geringen Umfangs nicht meßbar. Jedes Programm hatte seine Aufgabe in weniger als einer Sekunde beendet, und wieviel von der Zeit auf das Laden des Programms oder den Start der Java-Maschine entfiel, ist schwer zu sagen. Bei diesem Beispiel sieht das Ergebnis jedoch etwas anders aus:

- Das *DoDL*-Programm, also die übersetzte Spezifikation, benötigt zum Erzeugen der Verknüpfungsstruktur 33.61 Sekunden.
- Der Generator benötigt zum Erzeugen der HTML-Seiten 8.41 Sekunden.

Natürlich sind die gemessenen Zahlenwerte Schall und Rauch, da sie vom verwendeten Rechner abhängen. Bei dem nach wie vor steten Wachstum der Rechenleistung moderner Personal Computer dürften sie damit in einem oder zwei Jahren ebenfalls in einem

[1] Die inhaltliche Ausrichtung des Beispiels erlaubt keinerlei Rückschlüsse auf die Art und Weise, wie während der Diplomarbeit Entscheidungen bezüglich des Konzeptes oder der Realisierung getroffen wurden. Vielmehr ist es so, daß die astrologischen Fachtexte aufgrund eines Projektes, an dem der Autor vor etwa 10 Jahren beteiligt war, noch auf der Festplatte schlummerten. Allerdings deutet der Jupiter im zehnten Haus zufällig darauf hin, daß der Kombination aus XML und Java eine große Zukunft bevorsteht...

Bereich liegen, der nicht mehr ohne weiteres zu messen ist. Interessant und in gewisser Weise erstaunlich ist jedoch das Verhältnis der beiden Zahlenwerte zueinander, denn der Generator benötigt nur ein Viertel der Laufzeit der ausführbaren Spezifikation. Da beide Programme aber letztlich mit der gleichen Datenmenge umzugehen haben und der Generator zudem in Java implementiert ist (und somit als Bytecode ausgeführt wird), wäre eher ein Ergebnis zugunsten des *DoDL*-Programms zu erwarten gewesen. Überlegt man sich allerdings, welche Operationen die beiden Programme auf den vorhandenen Daten durchführen, dann erklären sich die gemessenen Laufzeiten von selbst:

- Der HTML-Generator betrachtet zum Schreiben der Ausgabedateien jedes Medienobjekt genau einmal. Aufgrund der normalisierten Darstellung (siehe Abschnitt 5.6) durchläuft er das Medienobjekt sequentiell und schreibt abwechselnd Positionen und den Text, der sich zwischen diesen Positionen befindet, in die Zieldatei.

- Das *DoDL*-Programm hingegen muß jedes Medienobjekt so oft betrachten, wie die Methode `getOcc()` darauf angewendet wird. Es muß das ganze Medienobjekt auf Vorkommen der gesuchten Zeichenkette überprüfen. Dadurch und durch die schiere Anzahl der Medienobjekte wächst die Laufzeit entsprechend an.

Gerade die Betrachtung der Laufzeiten legt die Vermutung nahe, daß die Methode `getOcc()` nicht effizient implementiert ist. Das ist auch richtig, denn die Methode zieht sich schlicht auf die Funktion `strstr()` der ANSI-C-Standardbibliothek zurück. Wenn *DoDL* also tatsächlich zur Verarbeitung vieler großer Medienobjekte eingesetzt werden soll, dann wäre es sicherlich lohnend, über eine effizientere Implementierung von `getOcc()` nachzudenken. Hier bieten sich zum Beispiel die Algorithmen von Knuth, Morris und Pratt oder von Boyer und Moore an, die beide zum Beispiel in [Sedg83] beschrieben sind.

Ein anderer Punkt, der bei dem letzten Beispiel auffällt, ist das Verhältnis der großen Anzahl von urspünglich 500 Medienobjekten zu den nur 25 HTML-Seiten im endgültigen Hyperdokument. Dieser große Unterschied ist darauf zurückzuführen, daß sich die ebenfalls 25 Textdateien, die dem Lexikon eigentlich zugrunde liegen, aus vielen Absätzen und Zwischenüberschriften zusammensetzen. Wäre jede der Dateien als ein Medienobjekt behandelt worden, dann wäre das Ergebnis im HTML-Browser, der ja mehrere Leerzeichen und -zeilen zu einem Leerzeichen zusammenzieht, als ein riesiger Absatz erschienen und somit nicht mehr lesbar gewesen. Um dem zu entgehen, wurden aus den Überschriften und Absätzen viele Medienobjekte gebildet, die dann mit Verweisen zwischen Ende und Beginn aneinandergehängt wurden, um später wieder eine Seite zu bilden.

Diese Notwendigkeit offenbart tatsächlich eine Schwäche in *DoDL* selbst, das keinerlei Unterstützung für solche längeren Texte bietet, sondern stattdessen dem Autor die eigentlich völlig überflüssige Aufgabe aufbürdet, erst den Text in kleine Schnipsel aufzuteilen, damit *DoDL* ihn wieder vernünftig zusammensetzen kann.

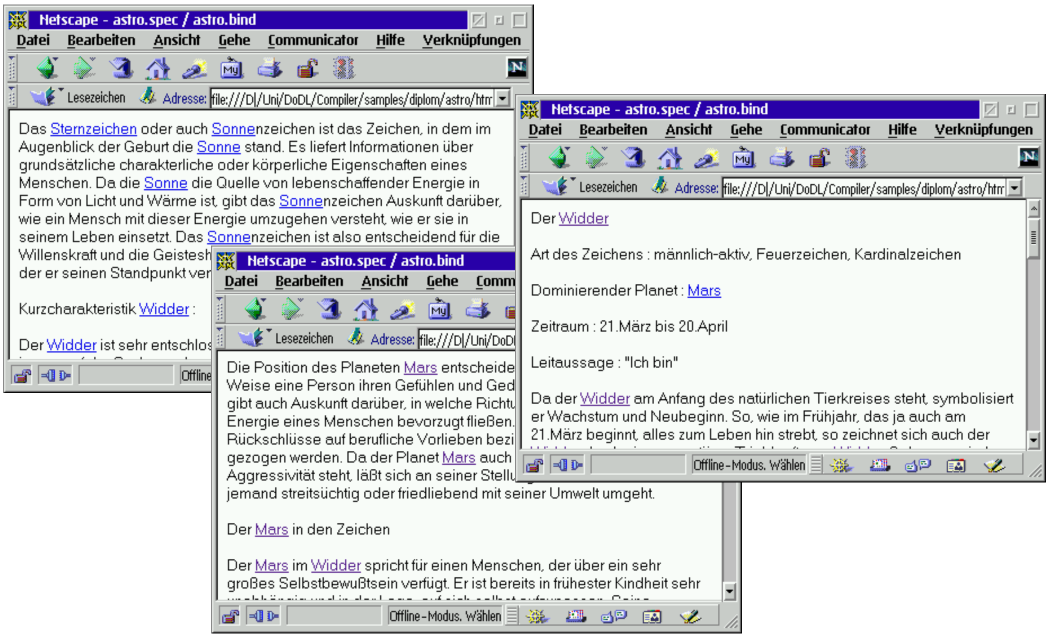

Abbildung 6.7.: HTML-Seiten des Astrologie-Lexikons

6.4. Bewertung

Wie die Testläufe zeigen, erfüllt das im Rahmen der Diplomarbeit implementierte und im vorliegenden Dokument beschriebene System die gestellte Aufgabe. Die Laufzeitbibliothek versetzt eine Spezifikation bzw. ein *DoDL*-Programm in die Lage, die Verknüpfungsstruktur eines Hyperdokumentes zu konstruieren und in eine externe Datei zu schreiben. Der HTML-Generator liest diese Datei und erzeugt daraus die gewünschten HTML-Seiten. Gerade der letzte Testlauf zeigt außerdem, daß das System vor größeren Hyperdokumenten nicht kapituliert, auch wenn die Laufzeit durchaus verbesserungswürdig ist.

Trotzdem kommt den einzelnen Komponenten eher der Status eines Prototypen als der eines fertigen Produkts zu. Von den verschiedenen kleinen „Teufeln im Detail“, die noch auszutreiben wären, bevor sich das System wirklich für den praktischen Einsatz eignet, seien die folgenden drei hier stellvertretend genannt:

- Die Übersetzung der Medienobjekte in die HTML-übliche Zeichencodierung ist nur rudimentär unterstützt. Tatsächlich werden nur die HTML-Sonderzeichen und die deutschen Umlaute übersetzt. Umlaute aus anderen Sprachen werden nicht berücksichtigt. Auch geht der Generator davon aus, daß die Medienobjekte selbst in der üblichen Zeichencodierung der Plattform vorliegen, auf welcher er ausgeführt wird. Auf dem System des Autors ist dies die IBM-OS/2-Zeichentabelle *850*, auf Unix- und Windows-Systemen ist es die ISO-Zeichentabelle *Latin-1*.

- Auch wenn das System mehrere Verweise unterstützt, die von der gleichen Quellposition ausgehen, ist es in dieser Hinsicht sicherlich nicht absolut „wasserdicht“. Es ist zum Beispiel möglich, eine Spezifikation zu konstruieren, die geschachtelte Positionen produziert. Wie der Generator auf eine solche Struktur reagiert, ist nicht getestet worden. Im günstigsten Fall wird vermutlich „nur“ ungültiger HTML-Code erzeugt. Im ungünstigsten Fall wird der Generator abstürzen.

- Die Aufteilung der Medienobjekte auf Gruppen, also HTML-Seiten, und die Bestimmung ihrer Reihenfolge ist ein erster Schritt hin zu einem automatisierten Layout des Hyperdokumentes. Es sind aber Fragen offen geblieben, die von der momentanen Implementierung nicht berücksichtigt werden. Dazu zählt zum Beispiel die Frage nach dem Anfang eines Hyperdokumentes. Die Namen der HTML-Seiten werden zufällig bestimmt und lassen somit weder Rückschlüsse auf die Reihenfolge noch solche auf den Inhalt einer Seite zu. Als Konsequenz daraus muß der Leser alle HTML-Seiten ausprobieren, um die richtige Einstiegsseite zu finden.

Diese Unzulänglichkeiten liegen aber eher im Detail und lassen sich beheben, wenn Bedarf dazu besteht. Das Gesamtkonzept der Diplomarbeit hat sich nach Meinung des Autors als tragfähig erwiesen. Insbesondere die konsequent objektorientierte Implementierung des Generators und der Laufzeitbibliothek sorgen dafür, daß das System für nahezu beliebige Erweiterungen offen ist. Das nun folgende Kapitel gibt abschließend einen kleinen Ausblick auf einige dieser möglichen Erweiterungen.

7. Ausblick

Wohin kann die Reise im Anschluß an die Diplomarbeit gehen? In den Kapiteln 3 bis 5 wurden bereits zahlreiche Erweiterungen genannt, die sich anbieten, wie auch Alternativen, die sich zur Lösung der entsprechenden Teilaufgabe angeboten haben, aber aus dem einen oder anderen Grund verworfen wurden.

Eine Alternative, die bislang nicht genannt wurde, leistet ähnliches wie der Generator, bedient sich dabei jedoch komplett anderer Mittel: Es handelt sich um die *Document Style, Semantics and Specification Language (DSSSL)* [ISO96] und deren Implementierung *Jade* [Clar99]. Mit DSSSL ist es möglich, strukturell verschiedene SGML-Dateien ineinander zu überführen, wobei die Abbildung der Struktur mit Hilfe von Scheme [AbSu85] realisiert wird. Da XML eine Untermenge von SGML ist, sollte es auch möglich sein, mit DSSSL die Zwischendarstellung aus Kapitel 4 in HTML-Seiten zu überführen. DSSSL ist ein ISO-Standard, der sich allerdings nie wirklich durchsetzen konnte, was möglicherweise auf die Verwendung von Scheme zurückzuführen ist. Bessere Chancen hat allerdings die *Extensible Stylesheet Language (XSL)* [W3C99c], eine neuere Entwicklung des W3C, die in enger Verbindung mit XML steht. XSL soll, analog zu DSSSL, die Überführung von strukturell verschiedenen XML-Dokumenten ineinander ermöglichen. Jedoch basiert XSL nicht auf Scheme, sondern auf Java und JavaScript, was vermutlich zu einer höheren Akzeptanz führen wird. Der Ansatz von XSL klingt vielversprechend und verdient sicherlich weitere Beachtung. XSL ist aber noch in der Entwicklung, so daß zum jetzigen Zeitpunkt keine abschließende Beurteilung möglich ist. Insbesondere existiert keine komplette Spezifikation oder Referenzimplementierung, weshalb in der Diplomarbeit darauf verzichtet wurde, diesen Weg weiterzuverfolgen.

Bei den Erweiterungen des bestehenden Systems liegt natürlich die Integration zusätzlicher Medientypen nahe. Für den praktischen Einsatz von *DoDL* wäre es sehr interessant, über Überschriften, Listen, Tabellen, Hervorhebungen und ähnliche Layout-Mittel zu verfügen. Damit ließen sich komplexere und – vor allem – ansprechendere Hyperdokumente erzeugen, als es bisher möglich ist. Listen und Tabellen unterscheiden sich allerdings teilweise recht erheblich von den doch eher „simplen“ Medientypen `Text` und `Graphics`. Für ein Medienobjekt des Typs `Text` wird angenommen, daß seine innere Struktur einfach nur ein Strom von Zeichen ist. Die innere Struktur von `Graphics` wird bei der aktuellen Implementierung gar nicht betrachtet. Bei Listen und Tabellen ist eine Betrachtung der aus Elementen oder Zellen bestehenden inneren Struktur jedoch von sehr großer Bedeutung. Damit einher geht die Notwendigkeit, entweder Importmöglichkeiten für bestehende Dateiformate zu schaffen (etwa spezielle Medienobjekte für Excel-Tabellen) oder auf

Textdateien zurückzugreifen, in denen eine eigene Syntax zum Markieren von Elementen oder Zellen verwendet wird.

Ebenso nahe wie das Hinzufügen neuer Möglichkeiten auf der Eingabeseite des Systems liegt natürlich dessen Erweiterung um neue Möglichkeiten auf der Ausgabeseite, also die Unterstützung zusätzlicher Zielformate. Mit HTML und LaTeX stehen derzeit zwei Zielformate zur Verfügung, die sich zur Anzeige in einem Browser bzw. zum Ausdruck des gesamten Hyperdokumentes eignen. Sinnvoll wäre als Ergänzung eine Unterstützung des *Portable Document Format (PDF)* von Adobe, das die Vorteile der beiden anderen Formate vereint, da es für Bildschirm und Ausdruck gleichermaßen geeignet ist. Wie ein interessanter Artikel [Kirs00] (leider erst kurz vor Ende der Diplomarbeit) gezeigt hat, unterstützen bereits einige moderne TeX-Compiler die Erzeugung von PDF, wenn die Eingabe bestimmte Bedingungen erfüllt. Damit bietet es sich an, eine spezialisierten Nachkommen von `LaTeXWriter` zu entwickeln, der für diesen Zweck geeignet ist. Weitere spezialisierte Nachkommen von `Writer` könnten eine Schnittstelle zu anderen Hypermedia-Systemen öffnen.

Eine sinnvolle Ergänzung des Gesamtsystems wäre auch eine graphische Oberfläche, die sämtliche Werkzeuge, die auf dem Weg von der Spezifikation bis zum fertigen Hyperdokument benötigt werden, integriert. Diese Oberfläche könnte insbesondere zur Verwaltung der Medienobjekte in einer Datenbank eingesetzt werden, was ja dem urspünglichen Gedanken von *DoDL* entspräche. Nicht zuletzt könnte die Oberfläche die visuelle Entwicklung von Spezifikationen ermöglichen. Dieser Gedanke ist für den praktischen Einsatz eines Systems wie *DoDL* von besonderer Bedeutung, denn im Normalfall wird der Autor eines Hypertextes zwar mit Fachkenntnissen seines Spezialgebietes, jedoch nicht unbedingt mit Programmierkenntnissen ausgestattet sein.

Wie unschwer zu erkennen ist, sind noch viele Erweiterungen möglich, aber auch nötig, um das System *DoDL* dem Einsatz in der Praxis näherzubringen. Der Autor hofft, hierzu mit seiner Diplomarbeit einen kleinen Beitrag geleistet zu haben.

Literaturverzeichnis

[AbSu85] H. Abelson, G. J. Sussman: *Structure and Interpretation of Computer Programs,* MIT Press (1985)

[ArGo96] K. Arnold, J. Gosling: *Die Programmiersprache Java,* Addison Wesley (1996)

[Bahn98] T. Bahnes et. al.: *Kolibri – Kooperatives Lernen mittels Internet-basierter Informationstechniken,* Abschlußbericht einer Projektgruppe am Lehrstuhl für Automaten- und Schaltwerktheorie, Universität Dortmund (1998)

[BeMi98] H. Behme, S. Mintert: *XML in der Praxis,* Addison Wesley (1998)

[Brat86] I. Bratko: *PROLOG Programming for Artificial Intelligence,* Addison Wesley (1986)

[Broc11] F. A. Brockhaus (Hrsg.): *Brockhaus' Kleines Konversations-Lexikon,* Brockhaus-Verlag (1911)

[Bush45] V. Bush: *As We May Think,* The Atlantic Monthly (Juli 1945)

[Dobe95] E.-E. Doberkat: *A Language for Specifying Hyperdocuments,* Technischer Bericht des Lehrstuhls für Software-Technologie, Universität Dortmund (1995)

[Dobe96a] E.-E. Doberkat: *A Language for Specifying Hyperdocuments,* Artikel in *Software – Concepts and Tools*, Springer Verlag (1996)

[Dobe96b] E.-E. Doberkat: *Browsing a Hyperdocument,* Technischer Bericht des Lehrstuhls für Software-Technologie, Universität Dortmund (1996)

[Dobe98] E.-E. Doberkat: *Using Logic for the Specification of Hypermedia Documents,* Artikel in *Classifications, Data Analysis and Data Highways*, Springer Verlag (1998)

[Fron99] A. Fronk: *Support for Hypertext Maintenance,* IEEE Computers 32, VI, S. 8-9 (1999)

[FrPl99] A. Fronk, J. Pleumann *Der DoDL-Compiler,* Technischer Bericht des Lehrstuhls für Software-Technologie, Universität Dortmund (1999)

[Gamm95] E. Gamma et. al.: *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison Wesley (1995)

[GoJo97] J. Gosling, B. Joy, G. Steele: *Java - Die Sprachspezifikation,* Addison Wesley (1997)

[Haas99] M. Haase-Arndt: *Untersuchung eines Slot-Mechanismus in einer objektorientierten Sprache,* Diplomarbeit am Lehrstuhl für Software-Technologie, Universität Dortmund (1999)

[HaSc94] F. Halasz, M. Schwartz: *The Dexter Hypertext Reference Model,* Communications of the ACM 37, II, S. 30-39 (1994)

[KeRi90] B. W. Kernighan, D. M. Ritchie: *Programmieren in C, Zweite Ausgabe, ANSI-C,* Carl Hanser Verlag (1990)

[Kirs00] S. M. Kirsch: *TEX akrobatisch –* `pdftex` *erzeugt PDF ohne Umwege,* c't Magazin für Computertechnik, Ausgabe 2, S. 180-187, Heise Verlag (2000)

[Knut93] D. E. Knuth: *The Stanford Graphbase: A Platform for Combinatorial Computing,* Addison Wesley (1993)

[Lamp94] L. Lamport: *LATEX – A Document Preparation System,* Addison Wesley (1994)

[Maur96] H. Maurer: *HyperWave: The Next Generation Web Solution,* Addison Wesley (1996)

[MaWi99] V. Mattick, C.-P. Wirth: *An Algebraic Dexter-Based Hypertext Reference Model,* Forschungsbericht des Lehrstuhls für Programmiersysteme und Übersetzerbau, Universität Dortmund (1999)

[MeNa99] K. Mehlhorn, S. Näher: *The LEDA Platform of Combinatorial and Geometric Computing,* Cambridge University Press (1999)

[Newc91] S. R. Newcomb et. al.: *The „HyTime“ Hypermedia/Time-based Document Structuring Language,* Communications of the ACM 34, XI, S. 67 - 83 (1991)

[Sedg83] R. Sedgewick: *Algorithms,* Addison Wesley (1983)

[Smol92] G. Smolka: *Feature-Constrained Logics for Unification Grammars,* Journal of Logic Programming 12, S. 51-87 (1992)

[StFu89] P. D. Stotts, R. Furuta: *Petri-Net-Based Hypertext: Document Structure with Browsing Semantics,* ACM Transactions on Information Systems, 7, III, S. 29 (1989)

[Szil94] H. Szillat: *SGML – Eine praktische Einführung,* MITP (1994)

[WiLe97] U. K. Will, J. J. Leggett: *Hyperform: A Hypermedia System Development Environment,* ACM Transactions on Information Systems, 15, I, S. 1-31 (1997)

– Die folgenden Informationen sind nur online verfügbar –

[Borl99] Inprise Corporation: *Borland Delphi 5 - High Productivity Development for the Internet* URL: `http://www.borland.com/delphi/` (zuletzt gesichtet am 12.01.2000)

[Clar99] J. Clark: *Jade – James' DSSSL Engine,* URL: `http://www.jclark.com/jade/` (zuletzt gesichtet am 12.01.2000)

[FBI99a] Universität Dortmund, Fachbereich Informatik: *COMRIS: Co-Habited Mixed-Reality Information Spaces,* URL: `http://www-ai.cs.uni-dortmund.de/FORSCHUNG/PROJEKTE/COMRIS` (zuletzt gesichtet am 12.01.2000)

[FBI99b] Universität Dortmund, Fachbereich Informatik: *The PLGraph Library,* URL: `http://guiness.cs.uni-dortmund.de/projects/METAFrame/plgraph` (zuletzt gesichtet am 12.01.2000)

[FBI99c] Universität Dortmund, Fachbereich Informatik: *Projektgruppe HEU: Hypermedia-Entwicklungsumgebung,* URL: `http://ls10-www.informatik.uni-dortmund.de/PGs/pg326/Intro.shtml`, (zuletzt gesichtet am 12.01.2000)

[IBM99] IBM Corporation: *XML Parser for Java,* URL: `http://www.alphaworks.ibm.com/ tech/xml4j` (zuletzt gesichtet am 12.01.2000)

[ISO96] International Standards Organization: *DSSSL – Document Style, Semantics and Specification Language,* ISO/IEC 10179:1996, URL: `ftp://ftp.ornl.gov/ pub/sgml/WG8/DSSSL/` (zuletzt gesichtet am 12.01.2000)

[Sun99] Sun Microsystems, Inc.: *Java API for XML Parsing,* URL: `http://developer.java.sun.com/developer/earlyAccess/xml/` (zuletzt gesichtet am 12.01.2000)

[Pass99] Universität Passau: *Graphlet – A Toolkit for Graph Editors and Graph Algorithms,* URL: `http://www.fmi.uni-passau.de/Graphlet/` (zuletzt gesichtet am 12.01.2000)

[Wolf98] G. Wolf: *The Curse Of Xanadu,* URL: `http://wn-1.wired.com/wired/archive/3.06/xanadu_pr.html` (zuletzt gesichtet am 12.01.2000)

[W3C98a] The World Wide Web Consortium: *XML – Extensible Markup Language 1.0,* URL: `http://www.w3.org/TR/1998/REC-xml-19980210` (zuletzt gesichtet am 12.01.2000)

[W3C98b] The World Wide Web Consortium: *DOM – The Document Object Model Level 1,* URL: `http://www.w3.org/TR/REC-DOM-Level-1` (zuletzt gesichtet am 12.01.2000)

[W3C99a] The World Wide Web Consortium: *HTML – HyperText Markup Language 4.01,* URL: `http://www.w3.org/TR/html4/` (zuletzt gesichtet am 12.01.2000)

[W3C99b] The World Wide Web Consortium: *XHTML – Extensible HyperText Markup Language 1.0,* URL: `http://www.w3.org/TR/xhtml1/` (zuletzt gesichtet am 12.01.2000)

[W3C99c] The World Wide Web Consortium: *XSL – Extensible Stylesheet Language* URL: `http://www.w3.org/TR/WD-xsl/` (zuletzt gesichtet am 12.01.2000)

A. DTD der Zwischendarstellung

```
<!-- Beginn der Datei -->

<?xml version="1.0"?>

<!ELEMENT document (content,structure,browsing)>
<!ATTLIST document
          spec      CDATA #IMPLIED ""
          bind      CDATA #IMPLIED ""
          user      CDATA #IMPLIED ""
          date      CDATA #IMPLIED "">

<!ELEMENT content (dbunit*)>

<!ELEMENT dbunit EMPTY>
<!ATTLIST dbunit
          id        ID    #REQUIRED
          file      CDATA #REQUIRED
          type      CDATA #REQUIRED>

<!ELEMENT structure (position*)>

<!ELEMENT position (link*)>
<!ATTLIST position
          id        ID    #REQUIRED
          ref       IDREF #REQUIRED
          value     CDATA #REQUIRED>

<!ELEMENT link EMPTY>
<!ATTLIST link
          id        ID    #REQUIRED
          target    IDREF #REQUIRED>

<!ELEMENT browsing (attribute*)>

<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
          id        ID    #REQUIRED>
          ref       IDREF #REQUIRED
          name      CDATA #REQUIRED
          value     CDATA #REQUIRED>

<!-- Ende der Datei -->
```

B. Inhalt der CD-ROM

Die folgende Tabelle gibt einen Überblick über die Verzeichnisstruktur der beiliegenden CD-ROM. Um das System zu installieren, ist es sinnvoll, zunächst den gesamten Inhalt der CD-ROM auf die Festplatte zu kopieren. Anschließend sollte der *DoDL*-Compiler nach den Anweisungen eingerichtet werden, die in [FrPl99] angegeben sind. Im letzten Schritt sollte die Umgebungsvariable `CLASSPATH` so geändert werden, daß die Dateien bzw. Verzeichnisse `Java/jaxp/jaxp.jar` und `DoDL/Generator/src` darin enthalten sind. Der Generator selbst wird über `java dodl.dodl2html` oder `java dodl.dodl2latex` ausgeführt.

Verzeichnis	Inhalt
`/DoDL/Compiler`	Der geänderte *DoDL*-Compiler.
`/DoDL/Compiler/bin`	Das ausführbare Programm des Compilers sowie die zur Ausführung benötigten Dateien `DoC.sys` und `DoC.rtl`.
`/DoDL/Compiler/lib`	Die Laufzeitbibliothek des Compilers.
`/DoDL/Compiler/samples`	Verschiedene Beispiel-Spezifikationen zum Testen des Compilers.
`/DoDL/Compiler/src`	Der Quellcode des Compilers.
`/DoDL/Compiler/tmp`	Temporärer C-Quellcode, der während der Ausführung des Compilers benötigt wird.
`/DoDL/Diplom`	Der Text der Diplomarbeit als Postscript-Datei.
`/DoDL/Generator`	Der HTML- bzw. LaTeX-Generator.
`/DoDL/Generator/etc`	Die DTD der Zwischendarstellung.
`/DoDL/Generator/samples`	Die Beispiel-Spezifikationen für den Generator, die in der Diplomarbeit verwendet wurden.
`/DoDL/Generator/src`	Die Quellen und die `class`-Dateien des Generators.
`/DoDL/Generator/src/dodl`	Das Paket `dodl`.
`/DoDL/Generator/src/dodl/core`	Das Paket `dodl.core`.
`/DoDL/Generator/src/dodl/util`	Das Paket `dodl.util`.
`/Java`	Die verwendete XML-Bibliothek von Sun.