1 Ziele

- Beschleunigung der Erstellung von HTML Seiten durch Faktor > 5 gegenüber Erstellung per Hand
- Automatische Generierung von PDF-Dateien
- Einheitliches Layout für alle Seiten
- Einfaches Ändern des Layouts. Dies wirkt sich dann auf alle HTML Seiten bzw der PDF Datei aus.
- Automatisch Erstellen von Inhalts- bzw. Abbildungsverzeichnis und Navigationsleisten.
- Unabhängig von verwendeten Betriebssystem. (Bis jetzt allerdings nur auf Windows getestet).
- Unabhängig von teuren Tools. Es reicht ASCII Editor. Sinnvoll ist ein einfacher xml Editor

2 Die Architektur

Die folgende Abbildung zeigt die Grobarchitektur von CoCoDil. (Stand Version 0.7)

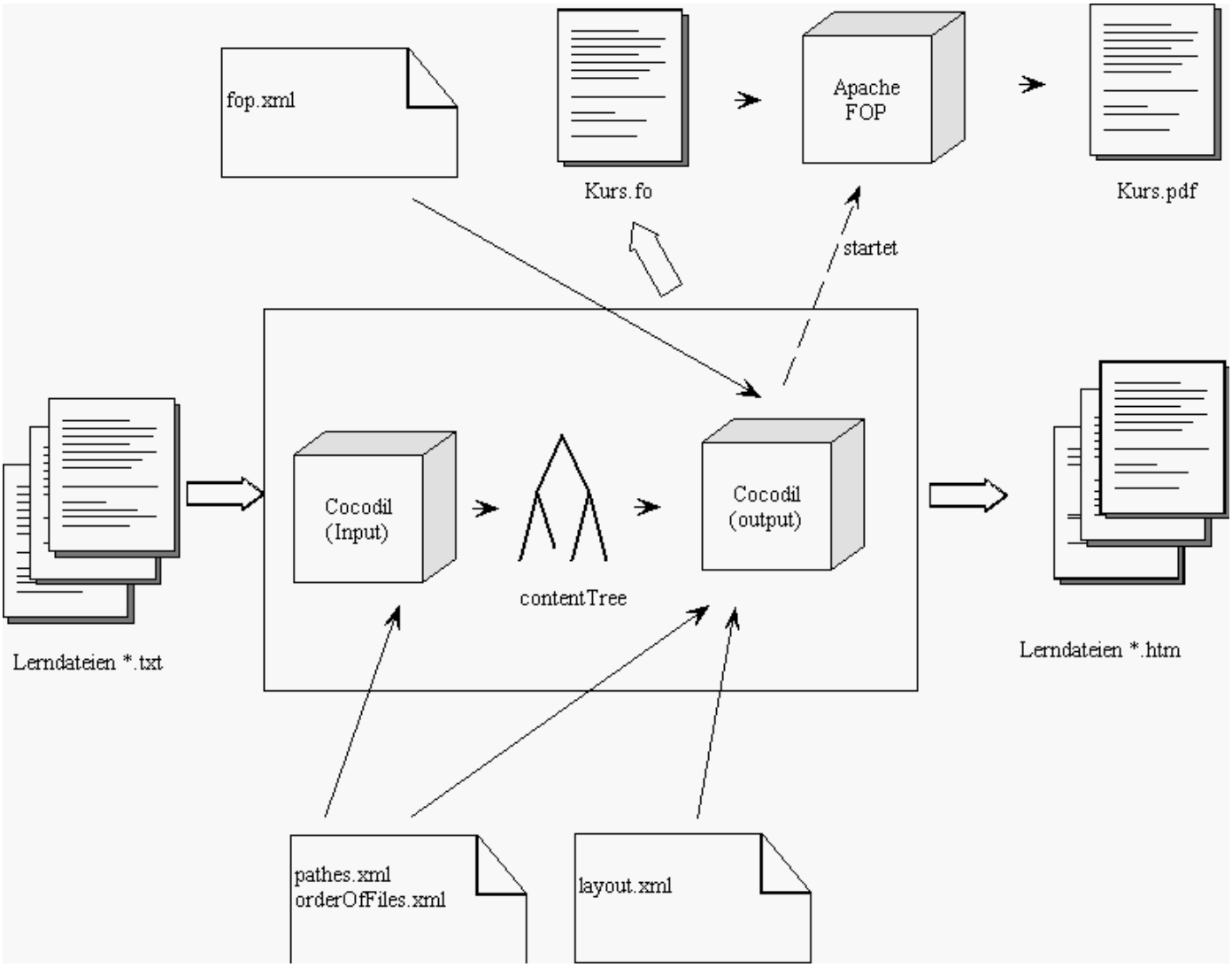

Abb. 2.1: Grobarchitektur CoCoDil

Lerndateien Hier steht die eigentliche Lerninhalte. Für jede Lerndatei wird eine eigene

HTML-Datei erstellt. Damit die Software diese interpretieren kann, müssen Sie einer bestimmten Syntax entsprechen. Diese ist einfach zu erlernen. Hier geht es zu den Beispielen.

pathes.xml Um nicht immer exakte und lange Pfadnamen angeben zu müssen, werden in dieser xml Datei Pfaden einen Begriff zugeordnet. Hier steht auch die Pfade der anderen Steuerdateien orderOfFiles.xml bzw. layout.xml. Beachte der Dateiname kann auch anderst heissen, er muss bei Start von CoCodil als Parameter mitgegeben werden. Siehe das Beispiel

orderOfFiles.xml Hier wird die Reihenfolge bestimmt, in der die Lerndateien interpretiert werden. Siehe Beispiel . Beachte dass sich der Dateiname auch anderst sein kann. Er muss in pathes.xml unter dem Begriff orderOfFiles abgespeichert sein.

ContentTree Der gesamte Lerninhalt wird in einer Baumstruktur festgehalten. Der ContentTree wird von CoCoDil erzeugt und vom Anwender nicht verändert. Der normale Kursautor der sich nur für den Inhalt interessiert braucht sich um den ContentTree nicht zu kümmern. Dijenigen die das Layout verändern wollen, brauchen aber ein rudimentäres Verständnis über den Aufbau des ContentTrees .

layout.xml Diese Datei muss nur geändert werden, falls der default Layout nicht den eigenen Ansprüchen genügt. Zum Ändern dieser Datei sind HTML Kenntnisse sowie eine Verständnis des prinzipiellen Aufbau des ContentTrees . notwendig. Der Dateiname layout.xml kann verändert werden. Der genaue Pfad muss unter pathes.xml unter dem Begriff layout abgespeichert sein. Hier ist der Aufbau von layout.xml beschrieben.

Html Dateien Die Html heissen wie die Lerndateien nur mit der Endung *.htm. Die Links zwischen den HTML Dateien untuntereinander und zu bestehen Bildern bzw Dokumente im FileSystem sind alle relativ. So kann man diese Dateien einfach in einen anderen Ordner oder auf einen anderen Rechner verschieben.

fop.xml Diese Datei hat denselben Aufbau wie die Datei **layout.xml**, und wird i.a. nur geändert wenn der Layout der generierten PDF-Datei nicht den eigenen Ansprüchen genügt. Diese Datei beschreibt, wie aus dem ContentTree eine fo-Datei erzeugt werden, die später mit Hilfe von APACHE FOP in eine pdf Datei umgewandelt wird. Zum Editieren dieser Datei sind Kenntnisse in XSL-FO notwendig. Eine sehr gute Beschreibung ist in den Tutorien *XSL Formatting Objects (XSL-FO) basics* und *XSL-FO advanced techniques* im IBM Developerworks

Kurs.fo Eine FO-Datei besteht aus einem Formating Objects Baum. Während HTML für Internetseiten konzipiert sind, orientiert sich XSL Formating Objects an ausdruckbare Seiten. Die *Kursname*.fo wird von CoCoDiL aus dem ContentTree erzeugt. Danach startet CoCoDiL **Apache FOP** um daraus eine pdf-Datei zu generieren.

Apache FOP Dies ist ein OpenSource Projekt, das einen Formating Object Baum in verschiedene Ausgabeformate umwandelt. Es werden die Format PDF,PCL,PS,SVG,XML (area tree representation), Print, AWT, MIF und TXT unterstützt. Das primäre Ausgabe Format ist PDF. Nähere Informationen unter der Apache FOP Homepage .

kurs.pdf Hier ist der Inhalt der Lerndateien als PDF umgewandelt. Dies ermöglicht den Ausdruck des Lerninhaltes.

3 Lerndateien

3.1 Absätze

Leerzeichen werden ignoriert. Ein neuer Text Abschnitt wird durch eine Leerzeile getrennt.

Beispiel:

```
Dies ist der
   Schuh des
Manitu
```

wird

Dies ist der Schuh des Manitu

Beispiel:

```
Dies ist der

Schuh des

Manitu
```

wird

Dies ist der

Schuh des

Manitu

HTML Tags werden akzeptiert. Allerdings können bei der Umwandlung in PDF Probleme auftreten.

Beispiel:

```
Dies ist der<br>
Schuh des<br>
Manitu<br>
```

Dies ist der
Schuh des
Manitu

3.2 Listen

Die Ausgabe der folgenden Beispiele sind abhängig von der Datei *layout.xml*. Es wird das Default Layout angenommen

Numerische Listen werden mit dem Zeichen # eingeleitet.
Ungeordnete Listen werden mit dem Zeichen * eingeleitet.
Die Tiefe der Liste hängt von der Anzahl der # bzw * ab.

Beispiel:

```
* Dies ist
** der Schuh
** des Manitu
```

wird

- Dies ist
  - der Schuh
  - des Manitu

Beispiel

```
# Dies ist
## der Schuh
## des Manitu
```

wird

1. Dies ist
   1. der Schuh
   2. des Manitu

Die Symbole # und * werden nur als ListItems erkannt, wenn diese am Anfang einer Zeile stehen.

Beispiel

```
# Dies ist
## der ** Schuh
   ## des ### Manitu
```

wird

1. Dies ist
   1. der ** Schuh
   2. des ### Manitu

Nach einer Leerzeile beginnt keine neue Liste.

Beispiel

```
# Dies ist
## der Schuh

# des Manitu
```

wird

1. Dies ist
   1. der Schuh
2. des Manitu

3.3 Anker

Anker werden für Cross-Referenzen innerhalb eines Kurses benutzt. Die Syntax ist

```
$A(einAnkername)
```

Der Ankername sollte innerhalb des gesamt zu erstellenden Kurses eindeutig sein. In Html können dieselben Anker in verschiedenen Seiten öfters vorkommen. Da aber pro Kurs der aus mehreren Html Seiten besteht,genau eine pdf Datei generiert wird, muss der Ankername auch innerhalb allen Html Seiten des Kurses eindeutig sein.

Eine Crossreferenz wird dann mit einem internen Link verbunden.

```
Dies ist ein [i|Link>einAnkername] zu der Stelle im Dokument an dem der Anker mit einAnkername definiert wurde.
```

3.4 Links

Ein Link hat folgende Syntax

```
[ Attributzeichen | darzustellender Name > URL oder Ankername]
```

Beispiel:

Aus

```
Hier ein Link zu [e|Cocodil>http://www.cocodil.org]
```

wird

Hier ein Link zu Cocodil

Mit Hilfe eines Attributzeichens kann man verschiedene Linktypen kennzeichnen. Im Default Layout Policy sind definiert:

- **e** externer Link (URL auf Dokumente im World Wide Web)
- **i** interner Link (Anker auf eine Stelle innerhalb des Kurses).

Die Einführung von Linktypen erleichtert es, eine Struktur in die zu erstellenden Kurse zu bringen.

Beachte:

- Alle Verweise, die von CoCoDiL berechnet werden sind relativ.

- In dieser Version muss der Anwender die Anker mit HTML-Code selbst setzen
- Inhaltsverzeichnis, Abbildungsverzeichnis und Navigationsleisten werden automatisch berechnet. Der Anwender muss dafür keine Anker setzen.

3.5 Kapitel

CoCoDil erzeugt die Kapitelnummern selbständig und erzeugt auch ein Inhaltsverzeichnis. Siehe zur Syntax folgendes Beispiel:

```
$K(Dies ist das Kapitel 1)
$KK(Dies ist ein Unterkapitel 1.1)
$KK(Dies ist das unterkapitel 1.2)
$K(Dies ist das Kapitel 2)
$KK(Dies ist das Unterkapitel 2.1)
$KKK(Dies ist das Unterkapitel 2.1.1)
$K(Dies ist das Kapitel 3)
```

Es wird in der DefaultLayout Policy eine Kapitelstruktur bis zur Tiefe 4 unterstützt.

3.6 Bilder

Um ein Bild einzufügen verwende bitte folgende Syntax.

Bsp1.: $B(Pfadname,Dateiname,Bezeichnung) oder

Bsp2.: $B(Pfadname,Dateiname,Bezeichnung, URL)

Dabei gilt:

Pfadname: Key eines Pfades, die in der Datei pathes.xml eingetragen ist.

Dateiname: Dateiname des Bildes.

Bezeichnung: Bildunterschrift. Diese wird durchgängig durchnummeriert, wobei die erste Nummer mit der Nummer des (obersten) Kapitels übereinstimmt. Die Bildunterschrift wird in das Abbildungsverzeichnis das automatisch erstellt wird übernommen. Möchte der Anwender auf eine Bildunterschrift verzichten, so sollte er **none** eingeben.

URL: Verknüpft ein Bild mit einer URL

Beispiel1:

Aus

```
$B(Icons,regel.gif,none)
```

wird

Beispiel2:

Aus

```
$B(Icons,regel.gif,Symbol fuer eine Regel)
```

wird

Abb. 3.2: Symbol fuer eine Regel

Beispiel3:

Aus

```
$B(Icons,regel.gif,Dieses Bild ist mit
einem Link verknüpft,http://www.cocodil.org)
```

wird

Abb. 3.3: Dieses Bild ist mit einem Link verknüpft

Beachte: Die Verknüpfung von Bilder mit Links funktioniert bei der PDF Umwandlung derzeit nicht.

3.7 Regionen

Regionen werden mit **$R(Name)** eingeleitet und mit **$R** abgeschlossen. Der Name kann frei vergeben werden, sollte aber in der layout Policy berücksichtigt werden.

Beispiel:

Aus

```
$R(Regel)
Hier steht eine Regel
$R\
```

wird:

Hier steht eine Regel

Es können auch Attribute zu einer Region mitgegeben werden. Die Syntax lautet dann **$R(Name>attr1=einWert,attr2=einAndererWert)**.

Beispiel:

Aus:

```
$R(Definition>begriff=Komödie)
Hier wird der Begriff Komödie definiert
$R\
```

wird

Hier wird der Begriff Komödie definiert

3.8 Quelltext

Es gibt derzeit zwei Möglichkeiten Quelltext einzugen.

Möglichkeit 1: Als Region mit dem Namen Source

Aus

```
$R(Source)
public int eine Methode(){
    int einInteger = 1;
    return einInteger
}
$R\
```

wird:

```
public int eine Methode(){
    int einInteger = 1;
    return einInteger
}
```

Der Quellcode wird mit dem HTML Kommando pre angezeigt. Die Einrückungen werden übernommen. Der Nachteil ist, dass Kommandos in diesem Quelltext interpretiert werden.

Möglichkeit 2:

Eine weitere Möglichkeit ist den Quelltext durch $S() bzw. $S\ einzuschliessen. Bei der Default Policy wird der Quelltext in einem TextEditor angezeigt.

3.9 Tabellen

Es lassen sich in CoCoDiL einfache Tabellen darstellen. Die Tabellen können nicht verschachtelt sein. Auch ist es nicht möglich benachbarte Tabellen Zellen miteinander zu verbinden.

Die Tabelle hat folgende Syntax:

```
$T(typ,alignString)
textSpalte_1Zeile_1 | textSpalte_2Zeile_1 | ... | textSpalte_nZeile_1 ||
textSpalte_1Zeile_2 | textSpalte_2Zeile_2 | ... | textSpalte_nZeile_2 ||
...
textSpalte_1Zeile_i | textSpalte_2Zeile_i | ... | textSpalte_nZeile_i ||
$T\
```

oder

```
$T(typ,alignString,breitenString)
textSpalte_1Zeile_1 | textSpalte_2Zeile_1 | ... | textSpalte_nZeile_1 ||
textSpalte_1Zeile_2 | textSpalte_2Zeile_2 | ... | textSpalte_nZeile_2 ||
...
textSpalte_1Zeile_i | textSpalte_2Zeile_i | ... | textSpalte_nZeile_i ||
$T\
```

- Eine Zelle wird durch die Nachbarzelle durch ein | getrennt.
- Eine Zeile wird durch doppelte Oder Zeichen || abgeschlossen.
- Jede Zeile hat genau die gleichen Anzahl von Spalten.

Der **alignString** ist genausolang wie die Anzahl der Spalten und besteht aus den Zeichen c,l,r.

Beispiel: Der **Alignstring** *cclr* sagt aus dass die Texte in der ersten und zweiten Spalte zentriert dargestellt werden. Die Texte in der dritten Spalte werden Links ausgerichtet. Die Texte in der letzten Spalte werden rechts ausgerichtet. Der Alignstring wird nur in der Html Darstellung aber nicht bei pdf-Erzeugung berücksichtigt.

Der **breitenString** ist optional und wird nur für die Tabellenzeilen in der pdf-Darstellung benötigt. Html berechnet die Breiten der Tabellenzellen selbständig. Bei der pdf-Darstellung muss man diese mitgeben. Wird kein Breitenstring mitgegeben haben alle Spalten in der pdf-Darstellung die gleiche Breite.

Beispiel: Der **breitenString** *20.20.40* besagt dass die erste und zweite Spalte 20 % der GesamtBreite einer Seite einnimmt, die dritte Spalte nimmt 40 Prozent der GesamtBreite einer Seite ein.

Man kann noch einen Typ mitgeben, um mit den Dateien layout.xml bzw. fop.xml den Tabellen verschiedenen Layouts zu geben. In den Beispielen sind folgende Layouts definiert:

- simple - Alle Zellen werden gleich dargestellt
- caption - Die erste Zeile wird fett gedruckt.

Beispiel:

```
$T(simple,clr)
Spalte1 | Spalte2 | Spalte3 ||
1 | 2 | 3 ||
100 | 200000000 | 3000000000000000 ||
a | b | c ||
$T\
```

Spalte1	Spalte2	Spalte3
1	2	3
100	200000000	3000000000000000
a	b	c

```
$T(caption,clr,10.30.60)
```

```
Spalte1 | Spalte2 | Spalte3 ||
1 | 2 | 3 ||
100 | 200000000 | 3000000000000000 ||
a | b | c ||
$T\
```

Spalte1	Spalte2	Spalte3
1	2	3
100	200000000	3000000000000000
a	b	c

4 Die Datei pathes.xml

Diese Datei dient dazu auf eine einfache Art und Weise alle Pfadnamen der verwendeten Dateien zu verwalten.

Siehe dazu folgendes Beispiel:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<pathes>
   <path key="Kurs" value="F:\Cocodil\Software\"/>
   <path key="Bilder" value="%Kurs%Bilder\"/>
   <path key="Icons" value="%Kurs%Icons\"/>
   <path key="Inhalt" value="%Kurs%content.htm"/>
   <path key="Abbildungen" value="%Kurs%pictureContent.htm"/>
   <path key="Control" value="%Kurs%Control\"/>
   <path key="orderOfFiles" value="%Control%orderOfFiles.xml"/>
   <path key="layout" value="%Control%layout.xml"/>
   <path key="Logging" value="%Kurs%Logging\"/>
   <path key="HtmlTree" value="%Logging%htmlTree.xml"/>
   <path key="foLayout" value="%Control%flop.xml"/>
   <path key="foFile" value="%Control%cocodil.fo"/>
   <path key="pdfFile" value="%Control%cocodil.pdf"/>
</pathes>
```

Folgende Bemerkungen:

- Der Kurs befindet sich unter dem Pfad F:\Cocodil\Software\
- Der Ordner mit den Bildern ist unter F:\Cocodil\Software\Bilder\
- Mit dem Prozentzeichen wird auf dem Key eines vorherigen Path Elements verwiesen.
- Die Verweisung mit Prozentzeichen auf einen anderen Pfad geht über mehrere Ebenen.
- Die Reihenfolge der path Elemente ist entscheident
- Man kann mit einem key einen Ordner oder eine Datei spezifizieren.

Beispiel für eine Verwendung:

```
$B(Icons,bild.gif,Ein Bild)
```

Es wird das Bild F:\Cocodil\Software\Icons\bild.gif eingebunden.

Bemerkungen zu den einzelnen Einträgen

- **KUrs** In diesem Ordner befinden sich alle Dateien für einen Kurs
- **Bilder** Hier sind alle Bilder drin, die in diesem Kurs eingebunden werden
- **Icons** Darin befinden sich Icons wie Regel.gif bzw. Definition.gif
- **Inhalt** In dieser Datei wird das Inhaltsverzeichnis abgelegt
- **Abbildungen** In dieser Datei wird das Abbildungsverzeichnis abgelegt.
- **Control** Hier befinden sich alle xml Dateien
- **orderOfFiles** xml Datei spezifiziert die Reihenfolge in der die Dateien abgearbeitet

werden.
- **layout** xml Datei die das Layout steuert
- **Logging** Pfad in der Log Datei wie Fehlermeldungen abgespeichert werden.
- **HtmlTree** Speichert Struktur mit erzeugten Baum, der die Lerninhalte repräsentiert ab.
- **foLayout** xml Datei die den Layout der zu generierenden pdf Datei steuert
- **foFile** fo-Datei die von CocoDiL erzeugt wird. Eingabe für Apache fop.
- **pdfFile** die generierte pdf Datei.

5 Die Datei orderOfFiles.xml

Die Datei bestimmt, in welcher Reihenfolge die Lerndateien abgearbeitet werden, und wo diese im Dateisystem zu finden sind.

```
<orderOfFiles>
   <file path="Kurs" file="einleitung.txt"/>
   <file path="" file="lerndateien.txt"/>
   <file path="" file="pathes.txt"/>
   <file path="" file="orderOfFiles.txt"/>
</orderOfFiles>
```

- Das Element **file** besteht aus einem attribut **file** und optional aus einem Attribut **path**.
- Der Wert des Attributs **path** sollte in der Datei pathes.xml enthalten sein.
- Fehlt das Attribut **path** oder besteht es aus einem Leerstring, so wird der path des obigen Elements angenommen.
- Das erste Element sollte ein Attribut **path** enthalten.

6 Aufbau des ContentTrees

Aus den Lerndateien erzeugt CoCoDiL einen ContentTree der die Lerninhalte repräsentiert. Erst danach wird aus diesem HtmlTree mit Hilfe der LayoutPolicy die eigentlichen Html-Seiten generiert.

Jeder Knoten hat eine Menge von Attribute. Jedes Attribut besteht aus einem Schlüssel und einem Wert. Jeder Knoten hat ein Attribut befehl, sowie ein Attribut startTemplate und ein Attribut endTemplate. Die Werte von startTemplate und endTemplate werden in der Datei layout.xml spezifiziert.Der Wert von startTemplate ist ein parametrisierbarer String der vor dem Arbeiten der KinderKnoten ausgegeben wird. Der Wert von endTemplate ist ein parametrisierbarer String, der nach dem Abarbeiten der Kinder ausgegeben wird. startTemplate und endTemplate werden mit den anderen Attributen des Kontens parametrisiert. Mit Hilfe dieser Templates werden die HtmlDateien gebildet.

6.1 Die Grundstruktur

Der oberste Knoten hat immer den Befehl root. Darunter kommen Knoten die einen Seitenwechsel repräsentieren. Die Attribute current, previous, next, first und last werden für die Bildung der Navigationsleiste benötigt.

In der unteren Abbildung wird davon ausgegangen, daß der Kurs aus den 3 Dateien datei1, datei2 bzw datei3 besteht.

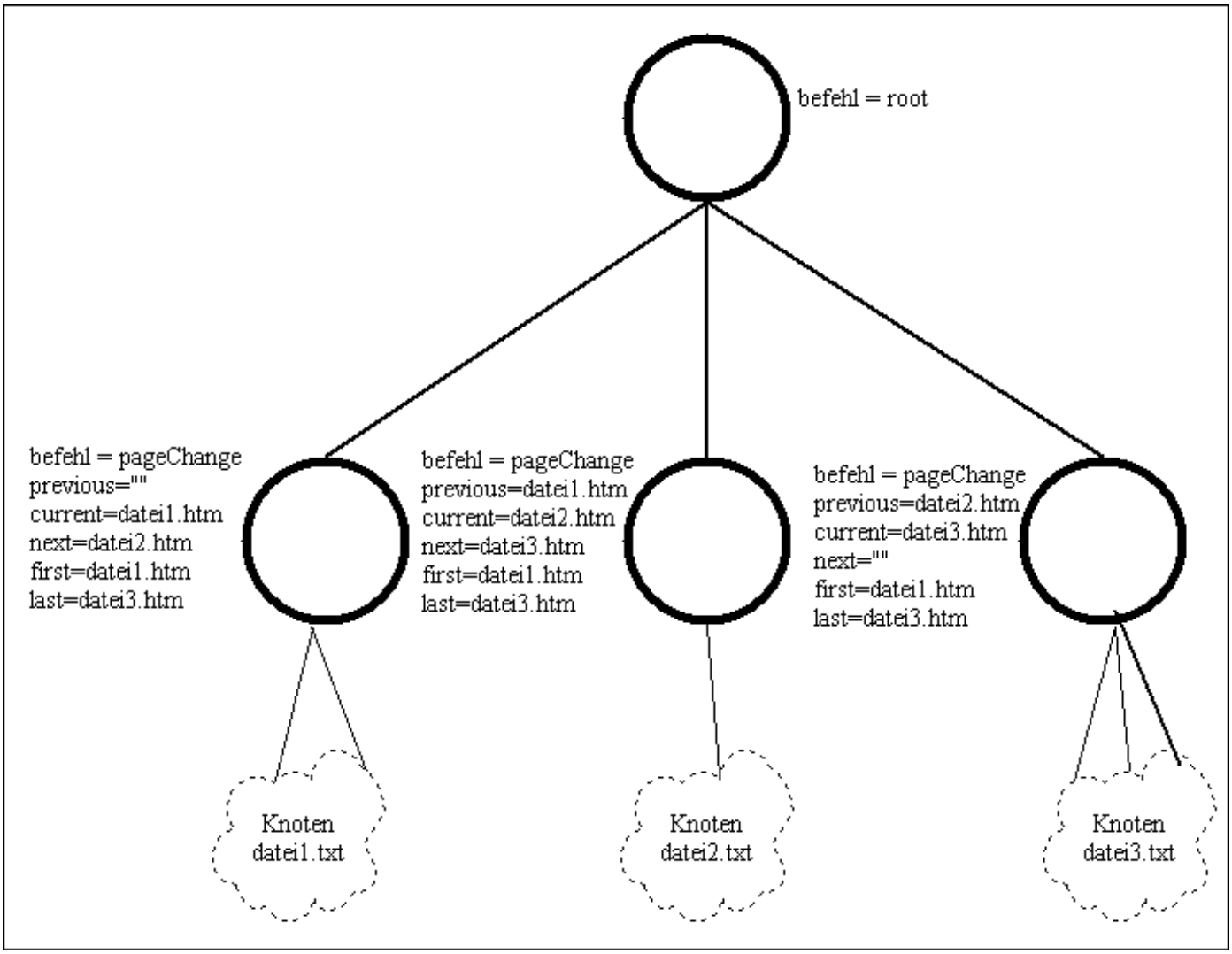

Abb. 6.1: Die obersten Knoten des Baums

Beachten Sie, daß die Attribute startTemplate und endTemplate im Bild nicht dargestellt werden.

Hinweis!

Die Attribute current, previous, next, first und last werden erst während dem Ausgabeprozeß berechnet. Deshalb sind diese Attribute bei der Analyse des Baums direkt nach dem Parsen nicht sichtbar.

6.2 Repräsentation von Texten

Die Repräsentation von Texten wird in der folgenden Abbildung deutlich. Es wird folgender Text repräsentiert: *Dies ist ein Text mit einem Verweis in der Mitte*.

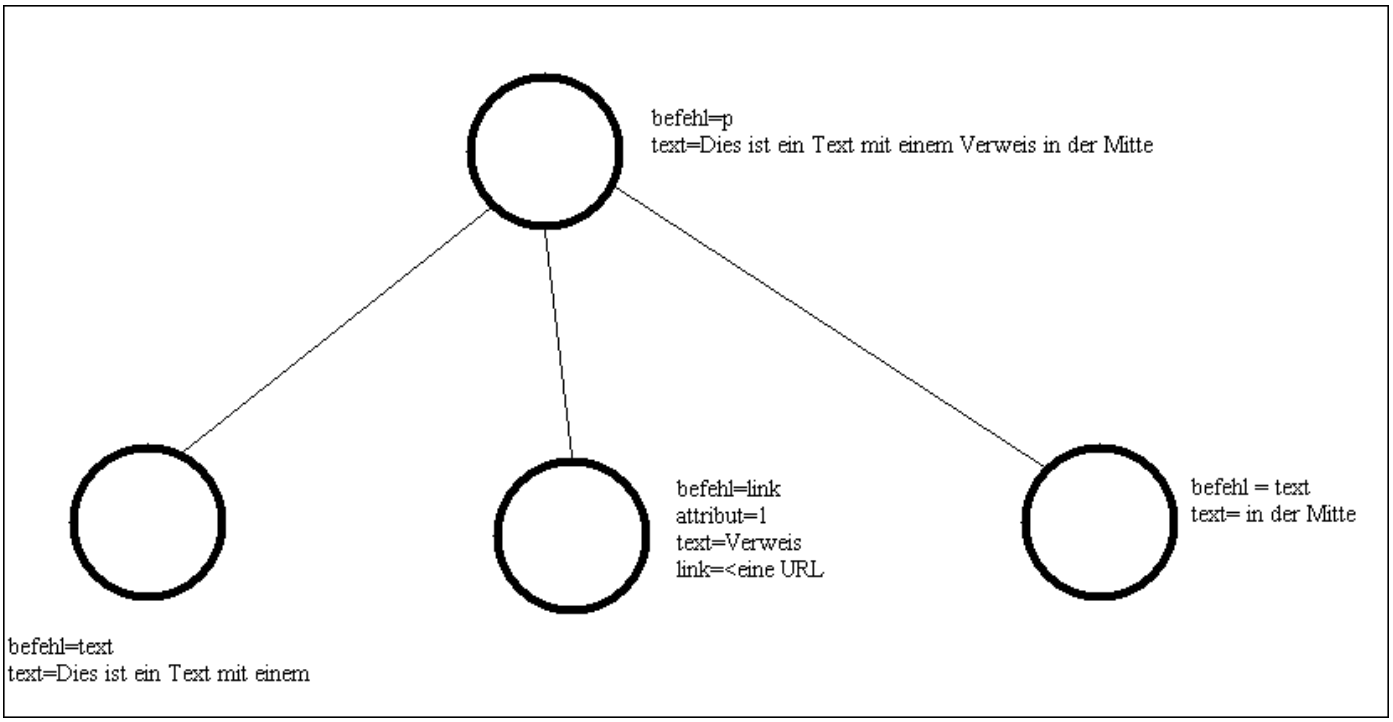

Abb. 6.2: Repräsentation eines Textes mit Verweis

Beachte dabei:

- Ein Knoten mit dem Befehl text, steht immer für einen puren Text ohne Verweis
- Ein Knoten mit dem Befehl p, hat immer mindestens einen Unterknoten text, auch wenn in dem Text keine Verweise sind.
- Ein Knoten mit dem Befehl link steht fuer einen Verweis. Das Attribut <attribut> kann folgende Werte annehmen
  - 1 fuer einen internen Link in dasselbe Dokument
  - 2 fuer einen externen Link

6.3 Repräsentation von Listen

Das folgende Beispiel zeigt wie Listen repräsentiert werden.

- Eine
  - verschachtelte Liste
  - mit Link

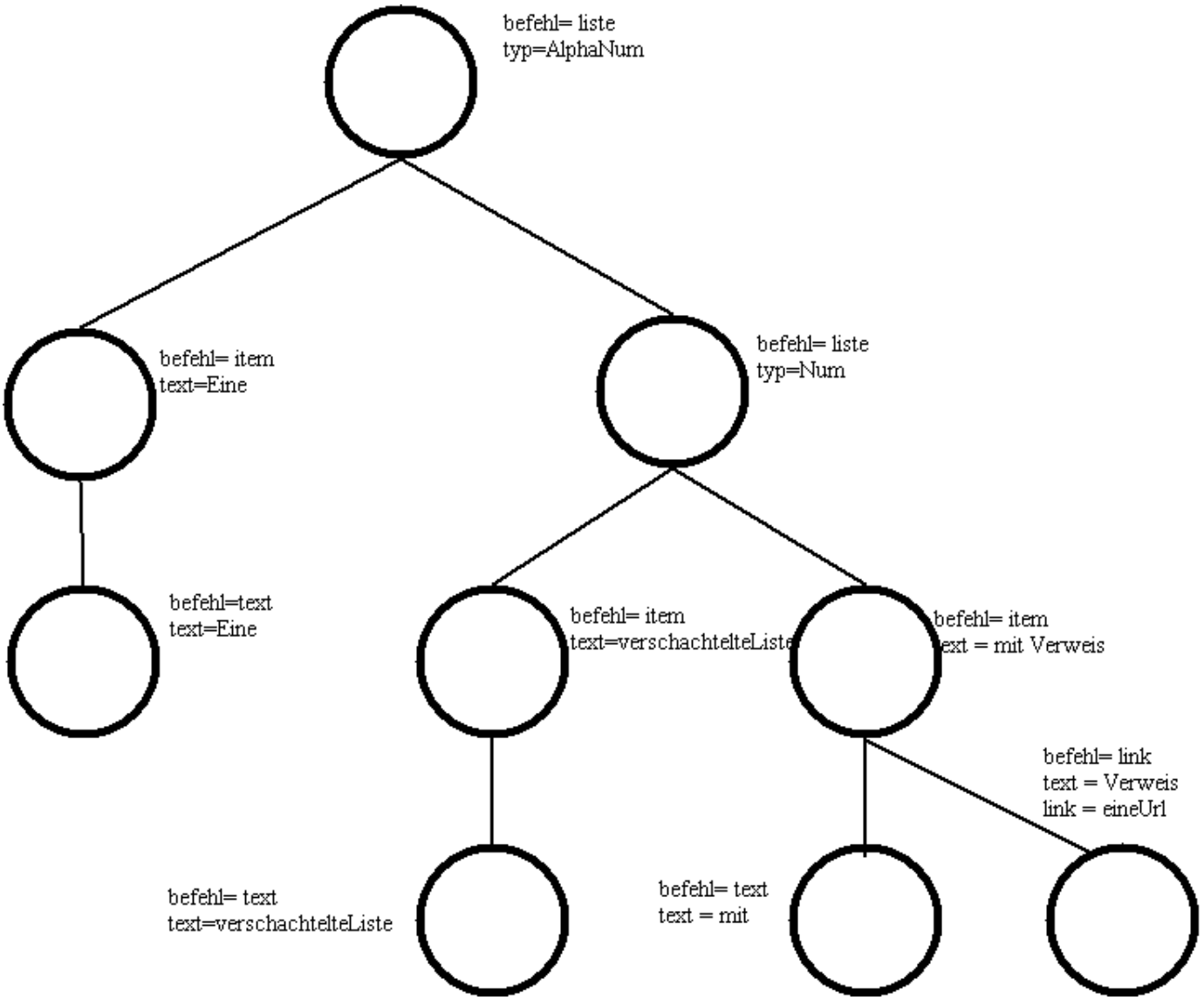

Abb. 6.3: Darstellung von Listen

Beachte dabei:

- Ein Knoten mit dem Befehl text, steht immer für einen puren Text ohne Verweis
- Ein Knoten mit dem Befehl item, hat immer mindestens einen Unterknoten text, auch wenn in dem Text keine Verweise sind.
- Ein Knoten mit dem Befehl item, hat als Vaterknoten einen Knoten mit Befehl liste

6.4 Repräsentation von Kapiteln

Ein Kapitelknoten hat folgende Attribute:

- **befehl** kapitel
- **text** Name des Kapitels mit Nummerierung
- **tiefe** Kapiteltiefe

Beispiel:

Das Kapitel *2.1.2 Ein nettes Kapitel* hat folgende Attribute

befehl=kapitel
text=2.1.2 Ein nettes Kapitel
tiefe=3

Beachte: Die Inhalte eines Kapitels befinden sich nicht unter dem Kapitelknoten, sondern rechts neben dem Kapitelknoten.

6.5 Repräsentation von Bilder

Ein Bildknoten hat folgende Attribute

befehl=picture
relativePath=relativer Pfad von der aktuellen Seite zum Bild
description=Bildunterschrift ohne Nummerierung falls nicht vorhanden hat es den Wert none
caption=Bildunterschrift mit Nummerierung. Dieses Attribut wird erst während der Ausgabe berechnet
link=url-Link falls nicht vorhanden hat er den Wert none

6.6 Repräsentation von Regionen

Hier ein Beipsiel für die Definition einer Region

```
$R(Definition>begriff=komödie)
Es wird der Begriff Komödie definiert
$R\
```

Dies wird folgendermassen repräsentiert:

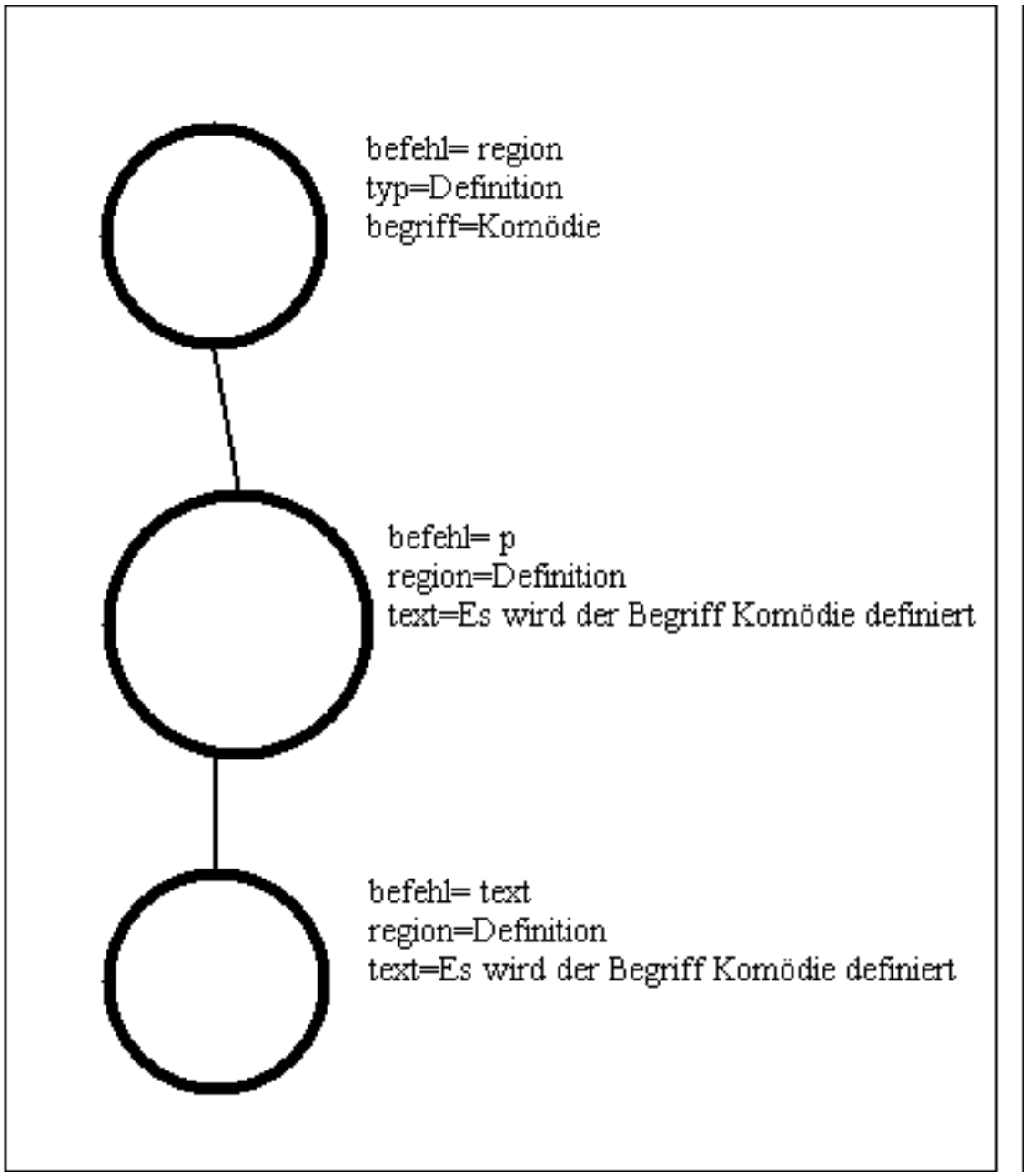

Abb. 6.4: Repräsentation einer Region

- Der Region Knoten hat den Befehl region
- Die Art der Region wird unter dem Attribut typ gehalten.
- Der Inhalt der Region wird als Unterknoten zu dem Regionknoten repräsentiert
- Alle Unterknoten der Region haben als Attribut region als Wert den Region typ
- Verschachtelte Regionen sind derzeit nicht möglich

6.7 Repräsentation von Quellcode

Ein innerhalb von $S() und $S\ eingebetteter Code wird in einem einzigen Knoten repräsentiert. Dieser Knoten hat folgende Attribute

befehl=source
text=eingebetteter Code
rows=Anzahl der Zeilen des eingebetteter Code

6.8 Repräsentation von Tabellen

Die folgende Struktur zeigt eine Tabelle mit 2 Spalten und 3 Zeilen, sowie die Attribute die einzelnen Knoten haben.

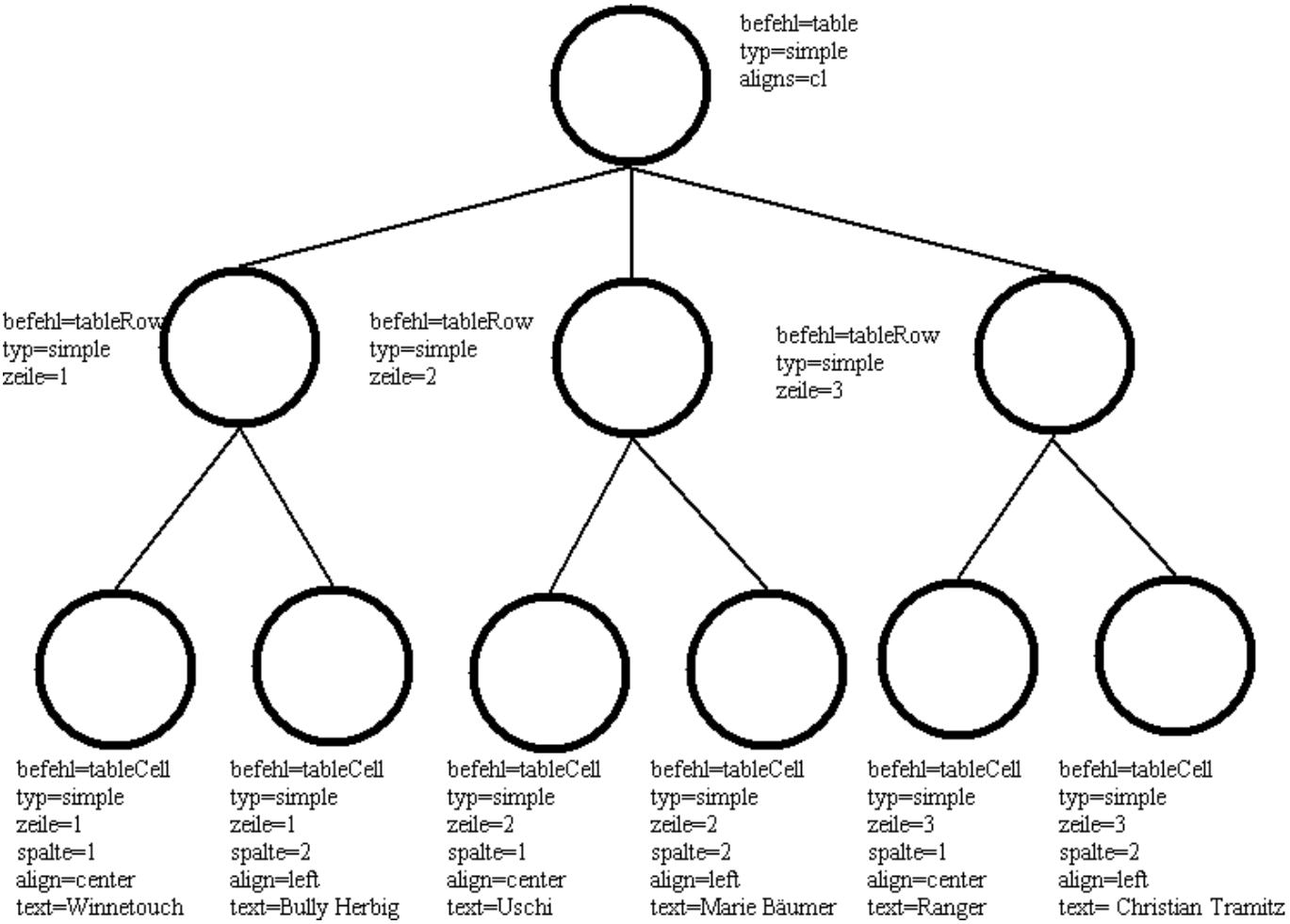

Abb. 6.5: Repräsentation von Tabellen

Der oberste Knoten hat den Befehl *table*. Dieser Knoten hat für jede Zeile der Tabelle einen Tochterknoten mit dem Befehl *tableRow*. Unter diesen Knoten sind eine Ebenen von Knoten mit dem Befehl *tableCell*.

7 Die Layout Dateien

Die Datei layout.xml bestimmt das Layout der generierten HtmlSeiten,die Datei fop.xml bestimmt das Layout der generierten pdf-Datei. Diese Dateien werden i.a. nur einmal oder sehr selten erstellt bzw. geändert. Der Kursautor muss sich normalerweise nicht darum kümmern.

7.1 Reihenfolge der Abarbeitung der Knoten im ContentTree

Hier ein Beispiel die Repräsentation einer 2-fach verschachtelten Liste.

- ein Text

- ein Text
- ein Text

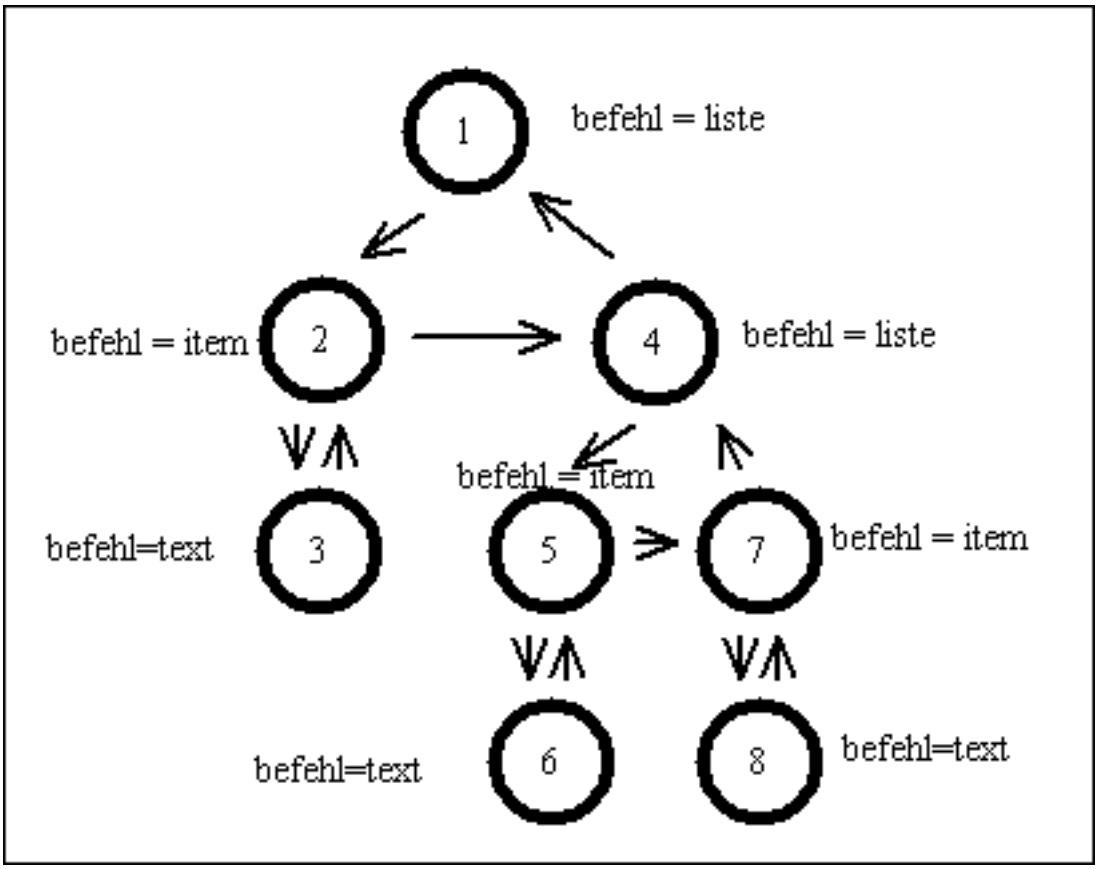

Abb. 7.1: Reihenfolge der Abarbeitung der Knoten

Zuerst wird in den obersten Knoten hineingegangen, dann wird der am weitesten links stehende Knoten mit all seinen Kindern abgearbeitet, danach die Geschwisterknoten. (Tiefensuche)

Beim Einstieg in einen Knoten wird in der Datei layout.xml das StartTemplate fuer den Knoten gesucht, ausgewertet und in der aktuellen HTML-Datei ausgegeben.

Beim Ausstieg in einen Knoten wird in der Datei layout.xml das EndTemplate fuer den Knoten gesucht, ausgewertet und in der aktuellen HTML-Datei ausgegeben.

Dies ergibt folgende Reihenfolge der Verarbeitetung:

1. startTemplate Knoten 1
2. startTemplate Knoten 2
3. startTemplate Knoten 3
4. endTemplate Knoten 3
5. endTemplate Knoten 2
6. startTemplate Knoten 4
7. startTemplate Knoten 5
8. startTemplate Knoten 6
9. endTemplate Knoten 6
10. endTemplate Knoten 5
11. startTemplate Knoten 7
12. startTemplate Knoten 8
13. endTemplate Knoten 8

14. endTemplate Knoten 7
15. endTemplate Knoten 4
16. endTemplate Knoten 1

7.2 Die Struktur der layout Dateien

Die untere Abbildung zeigt die Grundlegende Struktur der Datei layout.xml bzw fop.xml. Die Layoutbeschreibung besteht aus mehreren Policies. Für genau jeden möglichen Befehl (wie picture, item, text, p, source, region, pageChange, root, link, table) gibt es genau eine Policy. Eine Policy besteht aus einem oder mehreren Items. Diese haben die startTemplates bzw. endTemplates die bei der Abarbeitung eines HtmlTrees für jeden Knoten verarbeitet werden. Welches dieser Items verwendet werden, hängt von den Conditions ab.

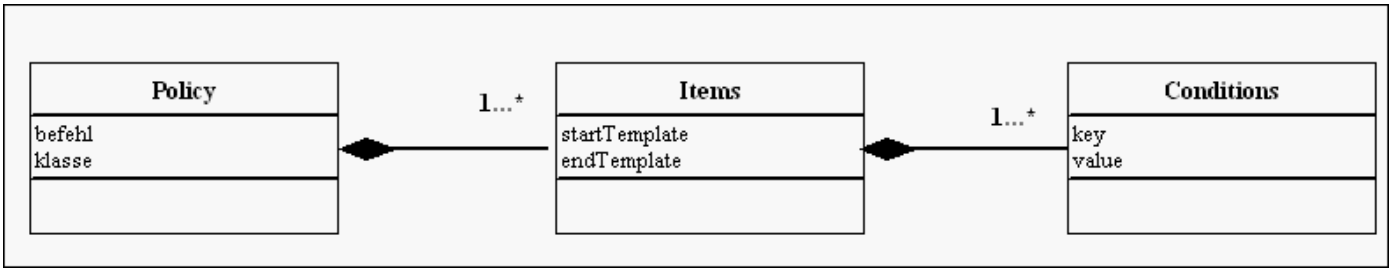

Abb. 7.2: Aufbau der Layout.xml

7.3 Finden des start- bzwl. EndTemplates

Schritt 1: Finden der Policy

Dies ist besonders einfach, da es zu jedem möglichen Befehl eines Knotens genau eine Policy gibt. Die Reihenfolge der Policies in der Datei layout.xml spielt keine Rolle. Ich empfehle diese nach den befehlen zu alphabetisch zu ordnen, um beim Editieren die richtige Policy schnell zu finden.

Schritt2: Finden der Klasse die den Knoten bearbeitet

CoCoDiL ist sehr flexibel in der Verarbeitung des ContentTrees. Man kann optional eine Klasse angeben, die die Verarbeitung des Knotens eines bestimmten Befehls übernimmt, und so das Standardverhalten der Bearbeitung und Ausgabe seinen eigenen Bedürfnissen anpassen.

Dies wird standardmässig schon sehr häufig gemacht. So wird z.B bei dem Befehl pageChange die Navigationsleiste berechnet. Bei Links wird z.B. nach einem Anker gesucht usw.

Der Klassenname muss voll qualifiziert (d.h mit der gesamten Paketstruktur) angegeben werden. Die Klasse sollte eine Unteklasse der abstrakten Klasse **cocodil.output.OutputStrategy** sein. Folgende Hookup Methoden können (müssen aber nicht) überschrieben werden.

Methode	Beschreibung
actionBegin(ContentNode)	wird aufgerufen bevor der aktuelle Knoten bearbeitet wird
output_begin(ContentNode, I_PrintWriter)	Ausgabe bevor die Kinder des Knoten bearbeitet werden

output_end(ContentNode, I_PrintWriter)	Ausgabe nachdem die Kinder des Knoten bearbeitet werden
action_end(ContentNode)	wird aufgerufen nachdem der aktuellen Knoten bearbeitet wurden

In einem ContentNode kann man beliebige Attribute speichern um dann in der Ausgabe darauf zuzugreifen. Die typische Vorgehensweise ist es, in der actionBegin Methode Attribute zu berechnen, um dann in der output_begin bzw output_end Methode darauf zuzugreifen.

Schritt 3: Finden des richtigen Items einer Policy

Eine Policy besteht aus einem oder mehreren Items. CoCoDiL muss genau ein Item finden. Dazu werden die Attribute des Knotens mit den key value Werten der Conditions eines Items verglichen. Sind alle Conditions erfolgreich, so wird die Suche abgebrochen und das dazugehörige Item genommen. Die Reihenfolge der Items innerhalb einer Policy ist also entscheident.

Beispiel (Auszug aus layout.xml):

```
<policies>
      <befehl>picture</befehl>
      <klasse>cocodil.output.html.PictureOutputStrategy</klasse>
      <items>
         <conditions>
            <key>description</key>
            <value>none</value>
         </conditions>
         <conditions>
            <key>link</key>
            <value>none</value>
         </conditions>
         <startTemplate>&lt;img src="$(relativePath)" border=0 hspace=2/&gt;</startTemplate>
         <endTemplate/>
      </items>
      <items>
         <conditions>
            <key>description</key>
            <value>none</value>
         </conditions>
         <startTemplate>&lt;a href="$(link)"&gt;&lt;img src="$(relativePath)" border=0 hspace=3 /&gt;</startTempla
         <endTemplate>&lt;/a&gt;</endTemplate>
      </items>
      <items>
         <conditions>
            <key>link</key>
            <value>none</value>
         </conditions>
         <startTemplate>&lt;a name="$(caption)"&gt;&lt;img src="$(relativePath)" border=1 /&gt;&lt;/a&gt;</startTe
         <endTemplate>&lt;br&gt;&lt;caption&gt;&lt;b&gt;$(caption)&lt;/b&gt;&lt;/caption&gt;</endTemplate>
      </items>
      <items>
         <conditions>
            <key>befehl</key>
            <value>picture</value>
         </conditions>
         <startTemplate>&lt;a href="$(link)" name="$(caption)"&gt;&lt;img src="$(relativePath)" border=1 /&gt;&lt;
         <endTemplate>&lt;br&gt;&lt;caption&gt;&lt;b&gt;$(caption)&lt;/b&gt;&lt;/caption&gt;</endTemplate>
      </items>
   </policies>
```

Dasselbe ist mit einem einfachen XML Editor leichter überblickbar. Ich benutze den kosenlosen XML Editor XML Nodepad .

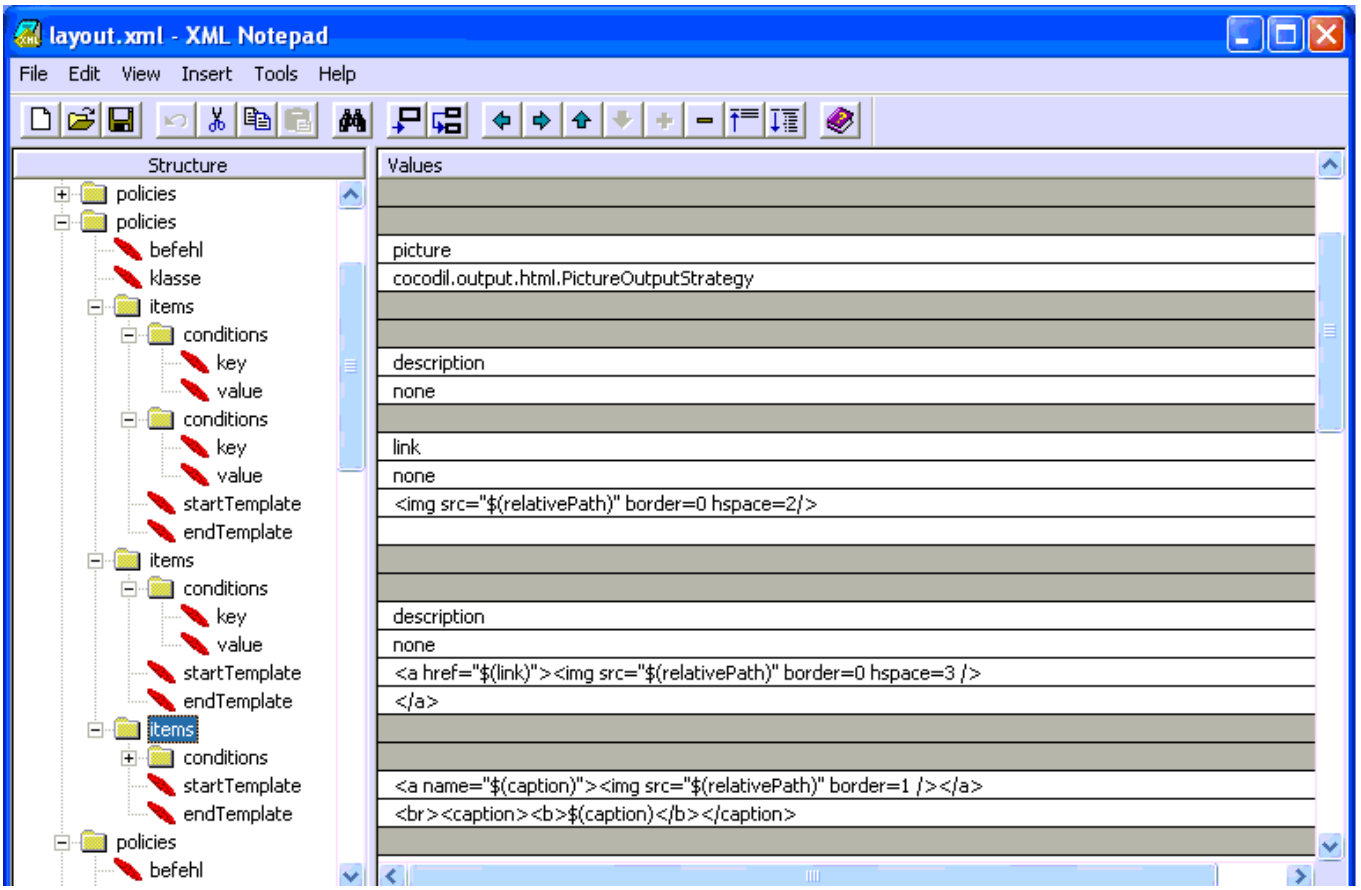

Abb. 7.3: Policy Struktur mit xml Editor

Angenommen der aktuelle Knoten hat folgende Attribute:

befehl: picture
description: none
relativePath: ./Bilder/bild.gif
link: eine URL

Das erste Items hat als erste Conditions, dass die Attribute description und link jeweils den Wert none haben. Dies ist nicht erfüllt.

Das zweite Items hat lediglich die Condition dass das Attribute description den Wert none hat. Dies ist erfüllt, damit wird nicht die weiteren Items durchsucht, sondern gleich das startTemplate und endTemplate diese Items genommen.

Das StartTemplate hat folgenden Wert:

```
<a href="$(link)"><img src="$(relativePath)" border=0 hspace=3 />
```

Das EndTemplate hat folgenden Wert:

```
</a>
```

Schritt 4: Auswerten der Templates

In den Templates gibt es Platzhalter für Attribute, die mit dem $ Zeichen eingeleitet werden. Es werden diese Platzhaltern mit den Attributwerten des aktuellen Knotens ersetzt.

Es wird für den aktuellen Knoten für das StartTemplate folgendes Ausgegeben:

```
<a href="eine URL"><img src="./Bilder/bild.gif" border=0 hspace=3 />
```

8 Installation

Zum Ablauf von CoCoDiL benötigt man

- JDK 1.4
- einen XML Parser z.B: von Apache Xerces .
- Apache FOP zur Umwandlung der fo-Datei in eine pdf Datei.

sowie die CoCoDiL Java Files .

Die Main Klasse heisst CoCoDiL und benötigt 2 Parameter:

1. Der Name des Kurses
2. Der Dateipfad an der sich die pathes.xml Datei befindet.

9 Vorschau

Folgende Erweiterungen sind bis Version 1.o geplant:

Änderungen an der Sprache der Lerndateien

- Nummerierung und Titel der Tabellen, Quelltexte, Regeln und Definitionen
- Automatische Verzeichnisse für Tabellen, Quelltexte, Regeln und Definitionen

Eine Benutzungsoberfläche

- Editieren der Dateien pathes und orderOfFiles mit Hilfe einer FileSelect-Box.
- Logging Fenster, der das Debuggen und Auffinden des Fehlern erleichtert.
- Grafische Analyse des ContentTrees. Analysieren welche Attribute an welchen Knoten definiert sind.

Einfachere Installation

- Anpassen an Unix bzw. Linux
- Erzeugen einer exe Datei für Windows

Folgendes Erweiterungen sind noch visionär:

- Eingabe der Lerndateien durch Internet. Cocodil wird ein Wiki Server.
- Wie bei WikiServer normal Volltextsuche und Versionierung der einzelnen Dateien.
- Möglichkeit eine aktuelle Version einzufrieren und auf dieser wieder aufzusetzen.