Universität Stuttgart

Fakultät Informatik

Prüfer:	Prof. Dr. rer. nat. Kurt Rothermel
Betreuer:	Dipl.−Inf. Ralf Rantzau
Beginn am:	16. Februar 1998
Beendet am:	4. September 1998
CR−Nummern:	H.2.8, H.3.5, H.5.1, I.7.2

STUDIENARBEIT NR. 1708

Datenbankgestütztes Info− und Verwaltungssystem für WWW−Inhalte

Martin Notz

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Das World Wide Web (kurz WWW) stellt das derzeit wohl beliebteste Informationssytem des Internets dar. Die Gründe für diese Beliebtheit bei den Anwendern sind die freie Verfügbarkeit von WWW−Servern und Browsern mit graphischer Benutzeroberfläche und intuitiver Bedienung, sowie die Integration bestehender Internetdienste (z. B. E−Mail, FTP, News) durch eine einheitliche Adressierung, eine konsistente Bedienung und ein standardisiertes Dokumentenformat. Das bequeme Navigieren mit Hilfe einfacher Mausklicks ermöglicht auch einem ungeübten Anwender das Informationsangebot des Internets zu nutzen.

Das World Wide Web ist kein eigener Internetdienst, sondern ein virtuelles Netzwerk im physischen Internet, das aus WWW−Objekten, WWW−Servern und WWW−Browsern besteht. WWW−Objekte sind elektronische Dokumente und Datenbestände jeder Art, wie E−Mail−Nachrichten, News−Artikel, Datenbanken und gewöhnliche Dateien. Eine besondere Rolle unter den WWW−Objekten nehmen WWW−Dokumente ein. Das sind Hypermediadokumente, die aus Text, Grafik, Ton− und Videosequenzen sowie allgemeinen Bedienungselementen (Schaltflächen, Listen und Eingabefeldern) bestehen können. WWW−Dokumente enthalten meist Verweise, sogenannte Hyperlinks, auf andere WWW−Objekte. Durch diese Verweise entsteht ein weltweit verteiltes Geflecht von Objekten, das dem WWW (weltweites Netz) seinen Namen gibt. WWW−Dokumente werden mit der Hypertext Markup Language (HTML) beschrieben und werden wegen dieser zugrundeliegenden Beschreibungssprache auch *HTML−Dokumente* genannt.

Die hohe Akzeptanz des World Wide Web hat zu einer Vielzahl von WWW−Angeboten geführt. Ein Problem ist die Aktualisierung und Pflege der Informationen auf WWW−Servern wie z. B. Personenbeschreibungen von Abteilungen. Diese Informationen sind zum Teil auf unterschiedlichen WWW−Dokumenten redundant vorhanden, sie ändern sich laufend und müssen auf allen entsprechenden *statischen WWW−Dokumenten* jeweils einzeln aktualisiert werden. Dies kann dazu führen, daß redundante Informationen nicht komplett gewartet werden und manche Seiten noch veraltete Informationen enthalten. Dazu kommen noch Zuständigkeitsprobleme einzelner Personen für die Aktualisierung durch Überschneidung der Grenzen von Verwaltungs− bzw. Verantwortungsbereichen.

Ein bessere Lösung für die Aktualisierung von WWW−Dokumenten ist, die benötigten Informationen in einer Datenbank zu speichern und bei jedem Aufruf der Dokumente durch Abfragen neu zu generieren (*dynamisch generierte WWW−Dokumente*). WWW−Dokumente, die auf diesen Datenbankinhalten beruhen, werden mit Hilfe von Programmen erzeugt, die eine Abfrage an eine Datenbank richten und das Ergebnis als HTML−Dokument aufbereiten. Solche Programme können durch unterschiedliche Techniken wie z. B. CGI und Servlets realisiert werden. Wenn die Datenbank aktualisiert wird, wirkt sich dies auf alle danach angeforderten Dokumente aus. Der Nachteil dieser Lösung ist im Gegensatz zu statischen WWW−Dokumenten, daß bei jedem Aufruf eines WWW−Dokuments durch den Anwender eine zusätzliche Kommunikation zwischen WWW−Server und Datenbank stattfindet.

Das Ziel dieser Arbeit ist die Verbindung der Vorteile statischer WWW−Dokumente mit den Vorteilen dynamischer WWW−Dokumente.

1.2 Aufgabenstellung

Der Gesamttitel dieser Arbeit beschäftigt sich mit zwei Aufgabengebieten, dem Gebiet der Informationssysteme und dem der Verwaltungssysteme für WWW−Inhalte.

Die ursprüngliche Arbeit bestand somit aus zwei Teilaufgaben, der Entwicklung eines datenbankgestützten Fakultäts−Informationssystems und eines Verwaltungssystems für WWW−Inhalte. Die erste Teilaufgabe der Informationssysteme wurde in die Diplomarbeit von [Gre 98] übernommen und parallel durchgeführt. In dieser Arbeit wird dafür das Verwaltungssystem für WWW−Inhalte ausführlich behandelt.

So wird im Rahmen dieser Arbeit ein Generator entwickelt und implementiert, der eine automatische Aktualisierung und Pflege von Informationen auf WWW−Servern mit Hilfe einer zentralen Datenbank ermöglicht.

Der Generator soll die entsprechenden WWW−Dokumente laden und an den dafür vorgesehenen Stellen die Informationen in den HTML−Quelltext einfügen. Die Informationen liegen nach der Aktualisierung als statische WWW−Dokumente vor und benötigen beim Aufruf keine Kommunikation mehr zwischen WWW−Server und Datenbank. Es handelt sich hierbei um *statische WWW−Dokumente mit dynamischer Aktualisierung*.

Damit der Generator erkennt, wo und in welcher Form er die Informationen einfügen soll, ist der Entwurf einer HTML−Erweiterung notwendig. Dazu soll in dieser Arbeit analysiert werden, welche Gestaltungsmöglichkeiten (z. B. Tabellen) HTML für die Datenbankinformationen bietet und welche davon sinnvoll in der HTML−Erweiterung zu realisieren sind.

1.3 Aufbau der Arbeit

Im folgenden Kapitel werden die Grundlagen dieser Arbeit behandelt. Zunächst wird kurz das World Wide Web mit seiner Auszeichnungssprache HTML erläutert. Im sich daran anschließenden Abschnitt werden alternative Möglichkeiten für datenbankgestützte Informationssysteme im Vergleich zu dem zu entwickelnden Generator dargestellt.

Im dritten Kapitel wird das Konzept des Generators vorgestellt und die funktionalen Anforderungen der zu entwerfenden HTML−Erweiterung analysiert und entwickelt. Anhand einer Beispieldatenbank werden dann die Möglichkeiten der HTML−Erweiterung verdeutlicht.

Im vierten Kapitel wird der Entwurf des Generators detailliert besprochen. Dazu zählen die Schnittstellen und die entworfene Grammatik für die HTML−Erweiterung.

Das fünfte Kapitel stellt die Programmierumgebung vor, die zur Implementierung des Generators benutzt wurde.

Im abschließenden Kapitel erfolgt dann eine zusammenfassende Betrachtung der Ergebnisse mit ihren zukünftigen Möglichkeiten.

2 Grundlagen

Die vorliegende Arbeit stellt eine Verbindung zwischen dem World Wide Web und einem Datenbanksystem her. Deshalb soll zu Anfang auf die Grundlagen des World Wide Web mit seiner Auszeichnungssprache HTML eingegangen werden. Danach werden alternative Mög−lichkeiten für datenbankgestützte Informationssysteme samt ihrer Vor− und Nachteile vorge−stellt.

2.1 Das World Wide Web

2.1.1 Allgemeines

Das World Wide Web ist ein auf der vorhandenen Hardware und Infrastruktur des Internets basierendes verteiltes Informationssystem. Es integriert verschiedene früher unabhängige ne−beneinander existierende Dienste in einer einheitlichen graphischen Umgebung durch eine einheitliche Adressierung, eine konsistente Bedienung und ein standardisiertes Dokumenten−format.

Wie schon in der Einleitung erwähnt, sind die Informationseinheiten des World Wide Web WWW−Objekte, die WWW−Dokumente und Datenbestände jeder Art, wie E−Mail−Nach−richten, News−Artikeln, Datenbanken und gewöhnliche Dateien darstellen können. Eine be−sondere Bedeutung nehmen in dieser Arbeit die WWW−Dokumente ein. Das sind Hyperme−diadokumente, die aus strukturiertem Text, Grafik, Ton− und Videosequenzen sowie allge−meinen Bedienungselementen (Schaltflächen, Listen und Eingabefeldern) bestehen können. WWW−Dokumente enthalten meist Verweise, sogenannte Hyperlinks, auf andere WWW−Objekte.

WWW−Server sind zum einen Server, die WWW−Dokumente zur Verfügung stellen (HTTP−Server) und zum anderen Server für die im WWW integrierten Dienste (z. B. E−Mail, FTP, News).

WWW−Browser dienen zur Navigation im WWW, zur Darstellung von WWW−Dokumenten und zur Interaktion mit WWW−Servern. Die derzeit am weitesten verbreiteten Browser sind Microsoft Internet Explorer, Mosaic und Netscape Navigator. Browser rufen WWW−Objekte von Servern ab und senden vom Benutzer eingegebene Daten an die Server.

2.1.2 Der Uniform Resource Locator

Für die Anforderung eines WWW−Objekts vom Server zum Browser ist es notwendig, das gewünschte Objekt eindeutig zu benennen. Dazu wird ein Uniform Resource Locator (URL) verwendet, der den physischen Ort adressiert, an dem das WWW−Objekt gespeichert ist. Der allgemeine Aufbau eines URL ist je nach Typ unterschiedlich, entspricht aber der Form

```
Protokoll://Hostname:Port/Pfad/Seite#Marke ,
```

wovon nicht immer alle Elemente angegeben werden müssen.

Im vorderen Teil wird das Protokoll spezifiziert, das für die Übertragung der Ressourcen (WWW−Objekte) vom WWW−Server verwendet werden soll. Für HTML−Dokumente ist es das Hypertext Transfer Protokoll (http). Andere mögliche Protokolle sind unter anderem ftp, file oder mailto. Im Hostnamen wird die IP−Adresse (Internet Protocol) oder der DNS−Name

(Domain Name System [RFC 1034, RFC 1035, RFC 1591]) des Rechners angegeben, auf dem der WWW−Server installiert ist. Der Pfad gibt den genauen Ort des Dokuments innerhalb des Dateisystems des Rechners an, die Marke eine bestimmte Stelle im Dokument.

Beispiele:

`http://www.informatik.uni-stuttgart.de/menschen/MeineHomepage.html` adressiert das WWW−Dokument `MeineHomepage.html`, das in dem Verzeichnis `menschen` auf dem HTTP−Server `www.informatik.uni-stuttgart.de` abgelegt ist.

Der E−Mail−Verweis *`mailto:notzmn@hermes.informatik.uni-stuttgart.de`* öffnet bei den meisten Browsern ein Fenster zum Schreiben einer E−Mail an den Empfänger `notzmn@hermes.informatik.uni-stuttgart.de`

2.1.3 HTML

HTML (HyperText Markup Language) [Mün 98] ist eine sogenannte Auszeichnungssprache (Markup Language) für das Word Wide Web. Sie hat die Aufgabe, die logischen Bestandteile eines Dokuments zu beschreiben. Als Auszeichnungssprache enthält HTML daher Befehle zum Markieren typischer Elemente eines Dokuments, wie Überschriften, Textabsätze, Listen, Tabellen oder Grafikreferenzen. Befehle werden in HTML auch Tags genannt, die in spitze Klammern (< und >) eingeschlossen sind.

HTML ist ein sogenanntes Klartext−Format. HTML−Dokumente sind ganz gewöhnliche ASCII−Text−Dateien, die mit jedem beliebigem Text−Editor erstellt werden können. Da HTML ein Klartext−Format ist, läßt es sich auch hervorragend mit Hilfe von Programmen generieren. Von dieser Möglichkeit machen beispielsweise CGI−Programme Gebrauch.

Befehle können auf verschiedene Weise angewendet werden:

```
<Befehl> Text </Befehl>
<Befehl Attribut=X> Text </Befehl>, oder nur
<Befehl>
```

Einige Befehle benötigen einen Start− und Endbefehl, die sich auf alles das beziehen, was von ihnen eingeschlossen wird. Daneben gibt es einige wenige Einzel−Befehle, das sind Befehle, die keine Einleitung für den folgenden Text darstellen und deshalb keinen Endbefehl haben (Bsp.: `<BR>`). Zusätzlich können einige Start− und Einzel−Befehle genauere Angaben in Form von Attributen enthalten.

Grundaufbau eines HTML−Dokuments

Ein gewöhnliches HTML−Dokument besteht grundsätzlich aus folgenden zwei Teilen:

- Header (Kopfteil): `<HEAD>...</HEAD>`
  (enthält allgemeine Angaben wie Titel u.ä.)
- Body (Textkörper): `<BODY>...</BODY>`
  (enthält den eigentlichen Text mit Überschriften, Verweisen, Grafikreferenzen usw.)

Beide Teile werden in die Befehle `<HTML>` bzw. `</HTML>` eingeschlossen.

Beispiel:

```
<HTML>
<HEAD>
<TITLE>Text des Titels</TITLE>
</HEAD>
<BODY>
Darzustellender Text
</BODY>
</HTML>
```

Dieses HTML−Dokument schreibt »Text des Titels« in den Titelbalken des Browsers und »Darzustellender Text« in das Anzeigefenster.

Der Titel des HTML−Dokuments ist die wichtigste Angabe im Kopfteil und sollte den Inhalt des Dokuments beschreiben. Unterhalb davon im Textkörper wird dann der eigentliche Inhalt des Dokuments notiert, also das, was im Anzeigefenster des WWW−Browsers angezeigt wer−den soll.

Einige Grundbefehle

Leerzeichen:
Mehrere Leerzeichen hintereinander werden vom WWW−Browser ignoriert und zu einem einzigen zusammengefaßt.

Kommentare:
Kommentare werden durch die Zeichenfolge `<!−−` eingeleitet. Beendet wird ein einzeiliger Kommentarbereich durch die Zeichenfolge `−−>` , ein mehrzeiliger durch `//−−>`.

Absätze:
Absätze werden durch die Befehle `<p>` und `</p>` eingeschlossen. Ein Zeilenumbruch inner−halb eines Absatzes kann durch den Befehl `<BR>` erzwungen werden.

Überschriften:
HTML unterscheidet 6 Überschriftenebenen, um Hierarchieverhältnisse in Dokumenten ab−zubilden.

Beispiel:

```
<H1>Überschrift 1. Ordnung</H1>
<H3>Überschrift 3. Ordnung</H3>
```

Listen

HTML unterscheidet drei Arten von Listen:

- Aufzählungslisten (Bulletlisten)
- Numerierte Listen und
- Definitionslisten
  Diese sind für Glossare gedacht.

Beispiel einer Aufzählungsliste:

```
<UL>
<LI>Listeneintrag</LI>
<LI>anderer Listeneintrag</LI>
<LI>letzter Listeneintrag</LI>
</UL>
```

Beispiel einer numerierten Liste:

```
<OL>
<LI>Listeneintrag, bekommt "1." vorangestellt</LI>
<LI>Listeneintrag, bekommt "2." vorangestellt</LI>
<LI>Listeneintrag, bekommt "3." vorangestellt</LI>
</OL>
```

Beispiel einer Definitionsliste:

```
<DL>
<DL>Ausdruck</DT>
<DD>Definition des Ausdrucks</DD>
<DT>Anderer Ausdruck</DT>
<DD>Definition dieses Ausdrucks</DD>
</DL>
```

Tabellen

HTML unterscheidet grundsätzlich zwischen zwei Tabellenarten,

- Tabellen mit Gitternetzlinien (für tabellarische Daten) und
- Tabellen ohne Gitternetzlinien, sogenannte »blinde Tabellen« (z. B. für mehrspaltigen Text).

Beispiel einer Tabelle mit Gitternetzlinien:

```
<TABLE BORDER>
<TR>
    <TH>Kopfzelle: 1. Zeile, 1. Spalte</TH>
    <TH>Kopfzelle: 1. Zeile, 2. Spalte</TH>
    <TH>Kopfzelle: 1. Zeile, 3. Spalte</TH>
</TR>
<TR>
    <TD>Datenzelle: 2. Zeile, 1. Spalte</TD>
    <TD>Datenzelle: 2. Zeile, 2. Spalte</TD>
    <TD>Datenzelle: 2. Zeile, 3. Spalte</TD>
</TR>
<TR>
    <TD>Datenzelle: 3. Zeile, 1. Spalte</TD>
    <TD>Datenzelle: 3. Zeile, 2. Spalte</TD>
    <TD>Datenzelle: 3. Zeile, 3. Spalte</TD>
</TR>
</TABLE>
```

Verweise − Hyperlinks

Alle Verweise in HTML haben einen einheitlichen Aufbau, egal ob sie zu einem Verweisziel in oder auf ein beliebiges WWW−Dokument oder zu einem beliebigen anderen WWW−Objekt führen.

Schema:

```
<A HREF="Verweisziel">Verweistext</A>
```

Verweisziele können sein:

- Verweis in gleiches Dokument

  ```
  <A HREF="#Zieldef">...</A>
  ```

- Verweis zu einem anderen WWW−Objekt

  ```
  <A HREF="URL">...</A>
  ```

- Suchstring zu einem WWW−Dokument

  ```
  <A HREF="URL?Suchwort+Suchwort">...</A>
  ```

  Schickt den Suchstring zu einem Server. Sollen mehrere Worte gefunden werden, können diese mit einem Plus−Zeichen verbunden werden.

Einbindung von Bildern

Bilder werden in HTML−Dokumente eingebunden, indem Grafikdateien an den gewünschten Stellen innerhalb der HTML−Dokumente referenziert werden. Geeignete Dateiformate für Grafiken sind GIF und JPEG, da sie von modernen Browsern in ihrem Anzeigefenster dargestellt werden können.

Die Einbindung einer Grafikreferenz erfolgt nach folgendem Schema:

```
<IMG SRC="datei.gif"> bzw. <IMG SRC="datei.jpg">
```

Nähere Angaben für die Grafikdarstellung können in Form von Attributen innerhalb des `<IMG>`−Befehls gemacht werden. So gibt es Attribute um

- die Breite und Höhe einer Grafik anzugeben,
- die Grafik zu beschriften,
- den Textfluß um die Grafik festzulegen und
- um festzulegen, ob und mit welcher Dicke die Grafik eingerahmt werden soll.

Mit der Angabe `WIDTH=` [Pixel] wird die Breite und mit `HEIGHT=` [Pixel] die Höhe einer Grafik angegeben.

Mit dem Attribut `ALIGN=TOP`, `ALIGN=MIDDLE` oder `ALIGN=BOTTOM` wird der nachfolgende Text als Beschriftungstext [Mün 98] interpretiert und entsprechend oben, mittig oder unten zur Grafik ausgerichtet. Da Beschriftungstext, der über die aktuelle Zeile hinausreicht, unterhalb der Grafik angezeigt wird, kann dieses Attribut auch dazu benutzt werden, eine Grafik innerhalb einer Textzeile darzustellen. Die Ausrichtung bezieht sich dabei auf die Basislinie der Textzeile.

Mit den Angaben `ALIGN=LEFT` bzw. `ALIGN=RIGHT` wird der Browser veranlaßt, die Grafik

linksbündig bzw. rechtsbündig auszurichten und den darauffolgenden Fließtext rechts bzw. links neben der Grafik anzuzeigen.

Mit der Angabe `BORDER=` [Pixel] kann ein Rahmen um eine Grafik definiert werden. Standardmäßig ist einer vorhanden, er kann aber durch das Setzen auf Null Pixel unsichtbar gemacht werden.

Beispiel:

```
<IMG SRC="CDROM.GIF" ALIGN=MIDDLE WIDTH=50 HEIGHT=50 BORDER=0>
```

Die Grafik `CDROM.GIF` wird mittig in der Textzeile in einer Größe von 50 mal 50 Pixeln ohne Umrandung ausgerichtet.

2.2 Andere Ansätze für datenbankgestützte Informationssysteme

Bei statischen WWW−Dokumenten gibt es keine Möglichkeit, auf Datenbanken zuzugreifen und deren Inhalte innerhalb von WWW−Dokumenten zurückzugeben. Erst durch Programme unterhalb des WWW−Servers können anstelle statischer WWW−Dokumente dynamische WWW−Dokumente mit eingebundenen Datenbankinhalten erzeugt werden. Die erste Möglichkeit dazu bot das Common Gateway Interface (CGI) [Thi 98], [Mün 98]. Eine auf der Programmiersprache Java aufbauende server−seitige Weiterentwicklung sind Servlets [Cli 98]. Beide Konzepte sollen in diesem Kapitel im Vergleich zum Konzept des Generators, der im darauf folgenden Kapitel behandelt wird, vorgestellt werden.

2.2.1 Das Common Gateway Interface − CGI

Das Common Gateway Interface ist eine Konvention für den Aufruf server−seitiger Applikationen und die Übertragung ihrer Ergebnisse an externe Anwendungen wie z. B. WWW−Browser. Es ermöglicht die Bereitstellung von Programmen im WWW. Diese können von WWW−Dokumenten aus aufgerufen werden und senden ihre Ergebnisse in Form von HTML−Code an den WWW−Browser zurück.

Eine CGI−Schnittstelle steht zur Verfügung, wenn der installierte WWW−Server CGI unterstützt. Es gibt keine Vorschriften, in welcher Programmiersprache ein CGI−Programm geschrieben sein muß. Ein CGI−Programm kann sowohl in einer Programmiersprache als auch in einer Script−Sprache geschrieben sein. Damit es auf dem WWW−Server ausführbar ist, muß es entweder für die Betriebssystem−Umgebung des Servers als ausführbares Programm kompiliert worden sein, oder es muß auf dem Server ein Laufzeit−Interpreter vorhanden sein, der das Script−Programm ausführt.

Beispiele für mögliche Programmiersprachen sind:

- C/C++
- Pascal
- Fortran
- Visual Basic

Beispiele für mögliche CGI−Script−Sprachen sind:

- Perl

- Unix−Shell−Sprache
- Javascript
- Tcl
- Rexx

Die meisten heutigen CGI−Programme sind in der Unix−Shell−Sprache oder in Perl geschrieben.

Ein CGI−Programm das Datenbankinhalte ausgibt, muß den Datenbanktreiber laden, die Verbindung zur Datenbank herstellen, die Daten abfragen, diese aus der Ergebnismenge tupelweise auslesen und entsprechend in HTML−Code formatieren. CGI−Programme können auf folgende Arten aus einem WWW−Dokument aufgerufen werden:

- Ein URL adressiert das CGI−Programm direkt, welches als Ergebnis ein WWW−Dokument mit eingebundenen Datenbankinformationen an den Browser zurückliefert. Wurde die ursprüngliche Endung des CGI−Programms in `.HTML` umbenannt, so erscheint der Aufruf für den Benutzer von diesem, wie ein Aufruf eines normalen WWW−Dokuments.
- Durch Aufruf eines Verweises auf ein CGI−Programm aus einem schon geladenen WWW−Dokument. Dieses liefert als Ergebnis ein neu erzeugtes WWW−Dokument mit eingebundenen Datenbankinformationen an den Browser zurück.
- Aufruf eines CGI−Programms nach Absenden eines Formulars [Mün 98].

  Dies ist sinnvoll, wenn dem Anwender die Möglichkeit gegeben werden soll, sich WWW−Dokumente mit individuell ausgewählten Datenbankinformationen generieren zu lassen. Die Suchbegriffe dazu werden in den Formularfeldern angegeben und deren Werte vom CGI−Programm verarbeitet. Das Ergebnis ist wiederum ein neu erzeugtes WWW−Dokument.
- Aufruf eines CGI−Programms über automatisches Laden. Dazu enthält das WWW−Dokument einen `<META>`−Befehl [Mün 98] mit der URL−Adresse des CGI−Programms. Für ein sofortiges Ausführen des CGI−Programms muß die Anzeigedauer (`CONTENT`) auf null gesetzt sein. Ansonsten wird das eigentlich aufgerufene Dokument noch die für eingestellte Anzeigedauer angezeigt, bis es durch das Ergebnisdokument des CGI−Programmes ersetzt wird.

  Beispiel:
  ```
  <META HTTP−EQUIV="REFRESH" CONTENT="0; URL=/cgi−bin/personen.pl">
  ```
- Einbinden der Ergebnisse des CGI−Programms direkt in das aktuelle WWW−Dokument. Dies geschieht in HTML mit Hilfe von Server Side Includes (SSI) [Mün 98].

  Beispiel: `<!−− #EXEX CGI="/cgi−bin/personen.pl" −−>`

  Diese Möglichkeit kommt der Aufgabenstellung des Generators am nächsten, da in dieser wie beim Generator die erzeugten Datenbankinhalte direkt in das WWW−Dokument eingebunden werden.

Der Ablauf beim Aufruf eines CGI−Programms sieht wie folgt aus: Der Benutzer startet eine HTTP−Anfrage in seinem WWW−Browser. Diese Anfrage wird an den WWW−Server weitergeleitet. Der Server startet das CGI−Programm in einem eigenen Prozeß, empfängt die Ausgaben des Programms und leitet diese an den WWW−Browser zur Darstellung weiter.

Abbildung 1 zeigt den Ablauf im einzelnen, wie er beim *direkten* Aufruf eines CGI−Programms mit Datenbankanfrage aussieht. Dieser Fall trifft zu bei den oben dargestellten Mög−

lichkeiten der direkten Adressierung des CGI−Programms über eine URL, beim Aufruf eines Verweises, beim Aufruf nach Absenden eines Formulars und beim Aufruf über automatisches Laden. Das CGI−Programm muß hierbei das gesamte Gerüst des WWW−Dokuments samt Header und Body (vgl. Abschnitt 2.1.3) erzeugen.

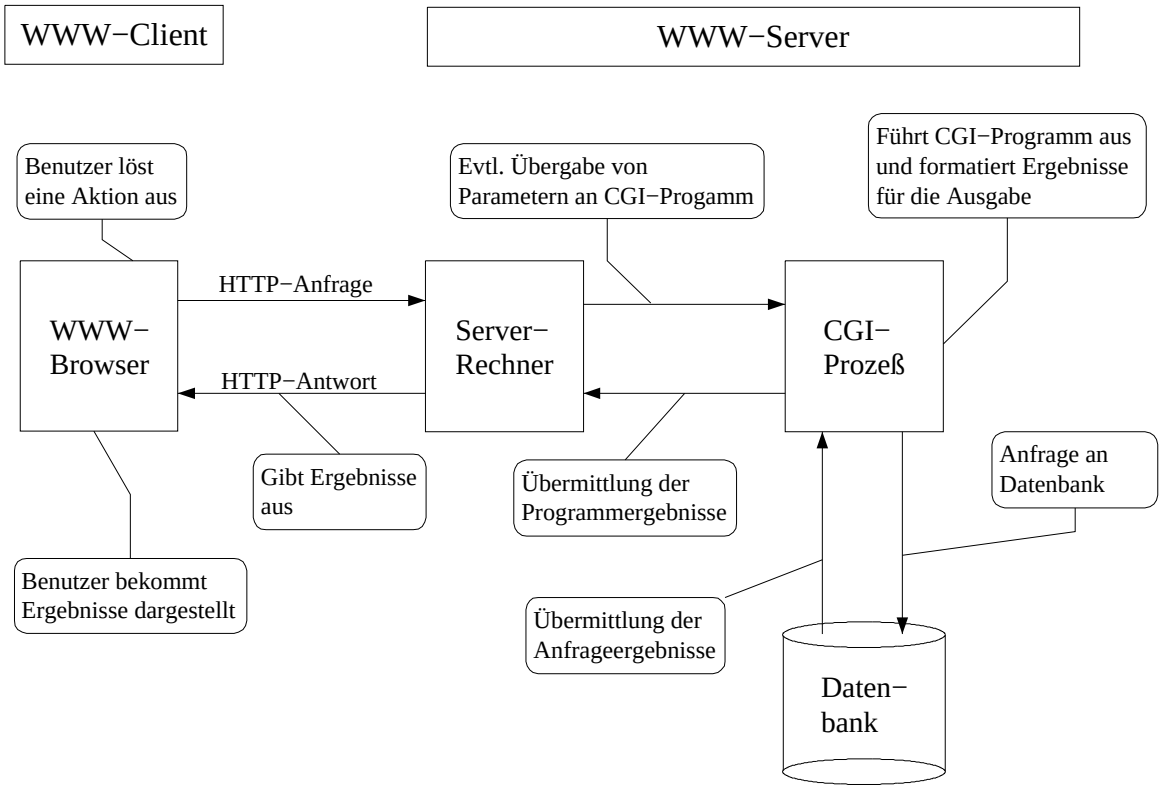

Abbildung 1: Direkter Aufruf eines CGI−Programms

Der Benutzer führt hier durch Anklicken auf eine Aktion eine HTTP−Anfrage aus, die der Browser an den zuständigen WWW−Server weiterleitet. Dieser startet für jeden Aufruf einen separaten CGI−Prozeß, in dem das CGI−Programm ausgeführt wird. Wurde ein Formular ab− gesendet, werden dem CGI−Programm zusätzlich die Werte aus den Formularfeldern überge− ben. Das CGI−Programm führt dann eine Datenbankanfrage aus und empfängt deren Ergeb− nisse. Die Ausgaben dieses Programms werden an den Server−Rechner übermittelt, der daraus ein dynamisches WWW−Dokument aufbaut und an den Browser übermittelt.

Abbildung 2 zeigt den Ablauf für den Aufruf über *Server Side Includes*. Der Unterschied zum direkten Aufruf eines CGI−Programms besteht darin, daß der CGI−Programmaufruf aus dem aktuellen WWW−Dokument ausgeführt wird und die Ergebnisse direkt an die Aufrufstelle des Dokuments geschrieben werden. Dabei wird nicht das gesamte Gerüst des WWW−Doku− ments vom Programm erzeugt, sondern lediglich die abgefragten Datenbankinformationen. Das Einbinden der Programmergebnisse führt wiederum der Server−Rechner durch.

CGI ist zwar eine einfache Möglichkeit, in WWW−Dokumente dynamische Informationen zu integrieren, doch sprechen viele Probleme mit CGI gegen den Einsatz dieser Technik. Der wichtigste Nachteil ist die schlechte Performance. Für jeden CGI−Aufruf startet der Server− Rechner einen eigenen Prozeß. Dieser muß erzeugt, geladen, ausgeführt und wieder beendet werden. Die Folge ist ein erhöhter Speicher− und Verwaltungsbedarf für das Betriebssystem. Dieser Overhead kann bei vielen fast gleichzeitig eintreffenden Anfragen den Server−Rechner blockieren.

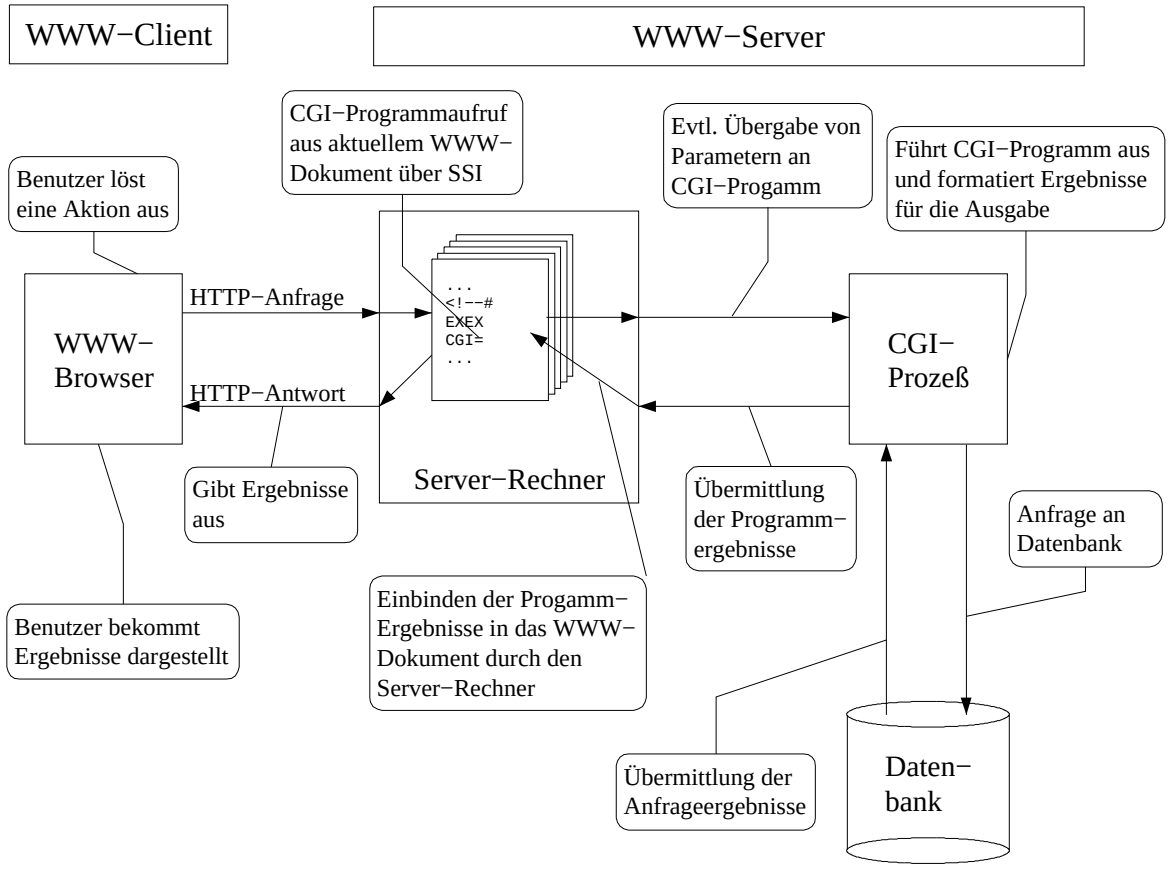

Abbildung 2: Aufruf eines CGI−Programms über Server Side Includes

Weitere Nachteile sind die Plattformabhängigkeit der CGI−Programme und die fehlende Möglichkeit, Zustandsinformationen im CGI−Programm zu speichern. CGI selbst ist zwar ein plattformübergreifender Standard, doch sind die Programme selbst in der Regel stark platt−formabhängig. So ist z. B. ein Unix−Shell−Script nur auf Unix−Rechnern ausführbar, in C/C++ geschriebene Programme müssen für die Betriebssystemumgebung des Servers com−piliert werden und sind nachher nur dort ausführbar. CGI−Prozesse können über ihre Ausfüh−rung hinaus keine Zustandsinformationen speichern. Um dies trotzdem erreichen zu können, wurde versucht dies z. B. mit Hilfe von Cookies zu lösen. Das sind Informationseinheiten, die vom WWW−Server dem Browser übermittelt und dort für eine bestimmte Zeit gespeichert werden. So können z. B. Informationen über das Verhalten der WWW−Benutzer gesammelt und ausgewertet werden.

Zusätzlich besteht die Gefahr, daß durch das fehleranfällige Feature der Zeigerarithmetik in C/C++−Programmen die Zuverlässigkeit und Stabilität des ganzen WWW−Servers beein−trächtigt werden kann.

Verbesserungen zu CGI wurden von den kommerziellen Herstellern wie Netscape oder Mi−crosoft eingeführt und machen damit der klassischen CGI−Schnittstelle zunehmend Konkur−renz. Beide Hersteller haben eigene Programmierschnittstellen (APIs) entwickelt und für ihre Server−Software−Produkte optimiert. Das Grundprinzip liegt darin, CGI−Programme als Bi−bliotheken zu schreiben, die in den WWW−Servers geladen werden. Die entscheidende Per−formanceverbesserung liegt darin, daß das Laden dieser Bibliotheken keinen Erzeugungspro−zeß erfordert. Die Zuverlässigkeit und Stabilität wird dadurch jedoch nicht verbessert. So ist eine fehlerhaft geschriebene Bibliothek immer noch in der Lage, den ganzen WWW−Server zu blockieren.

2.2.2 Servlets

Servlets [Cli 98] geben dem WWW−Server die Fähigkeit, Informationen mit externen Anwendungen wie z. B. Browsern auszutauschen. Es handelt sich hierbei um server−orientierte Java−Klassen, die vom WWW−Server bei Bedarf aufgerufen und geladen werden. Sie sind das server−seitige Äquivalent zu den Applets auf der Browserseite. Einmal geladen werden sie Teil des WWW−Servers, bis sie z. B. für eine neuere Version vom Server entfernt werden.

Servlets beheben die drei wichtigsten Schwachstellen von CGI: Performance, Plattformabhängigkeit des Codes und die fehlende Möglichkeit, Zustandsinformation im Programm zu speichern.

Sie bieten eine bessere Performance als CGI−Programme, da sie nur einmal geladen werden, anstelle bei jedem Aufruf neu erzeugt zu werden.

Die Plattformunabhängigkeit ist aufgrund der Programmiersprache Java automatisch gegeben.

Da Servlets nach dem Laden Teil des WWW−Servers werden, können sie Zustandsinformationen zwischen ihren Aufrufen speichern.

Servlets erhöhen die Zuverlässigkeit eines WWW−Servers. So ist es aufgrund der fehlenden Zeigerarithmetik von Java unwahrscheinlicher, mit einem Servlet einen WWW−Server zu blockieren als mit einem in C++ geschriebenen CGI−Programm. Außerdem erlauben es die meisten servlet−fähigen WWW−Server, Servlets in einer eigenen sogenannten Sandbox isoliert vom restlichen Server laufen zu lassen. Dies erhöht weiter die Stabilität und Sicherheit des Servers. Das Sandboxprinzip ist bei Servlets dasselbe wie bei Applets auf der Browserseite. Die Laufzeitumgebung kann so die Zugriffsrechte eines Servlets auf seine Umgebung einschränken. Somit können Servlets weitgehend isoliert auf einem lokalen Server ablaufen.

Ein weiterer wichtiger Vorteil von Servlets gegenüber CGI ist die Möglichkeit, Servlets sowohl vom lokalen Server als auch von einem beliebig entfernten Server zu laden. Dies ermöglicht z. B. in Unternehmen, Servlets für alle ihre WWW−Server zentral zu speichern und zu verwalten. CGI−Programme dagegen müssen auf einem WWW−Server in einem dafür eingerichteten Verzeichnis abgelegt sein. Benötigt ein anderer Server das gleich CGI−Programm, so muß es dorthin kopiert werden.

Um Datenbankinhalte in WWW−Dokumente einbinden zu können, ist es für Servlets genauso wie mit CGI notwendig, ein Programm zu schreiben, das diese Aufgabe erledigt. Anders als in CGI ist hier die Programmiersprache auf Java festgelegt. Diese garantiert eine Plattformunabhängigkeit der entwickelten Servlets. Servlets zum Einbinden von Datenbankinhalten können auf folgende Arten aus einem WWW−Dokument aufgerufen werden:

- über einen Servlet−Befehl in einem WWW−Dokument:

  Beispiel: `<SERVLET CLASS="ServletKlasse"></SERVLET>`

- wie mit CGI über Angabe der URL (vgl. vorherigen Abschnitt):

  Dazu wird im HTML−Befehl z. B. für Verweise oder Formulare anstelle eines CGI−Programms das URL des Servlets angegeben.

- als Teil einer Servletkette:

  Eine Servletkette, ist eine Aneinanderreihung von Servlets, die ihre Ergebnisse an andere Servlets zur Weiterverarbeitung weitergeben. Das letzte Servlet in dieser Kette sendet seine Ergebnisse in der Regel an den Browser.

Die beiden folgenden Abbildungen verdeutlichen die Architektur und das Konzept von Servlets.

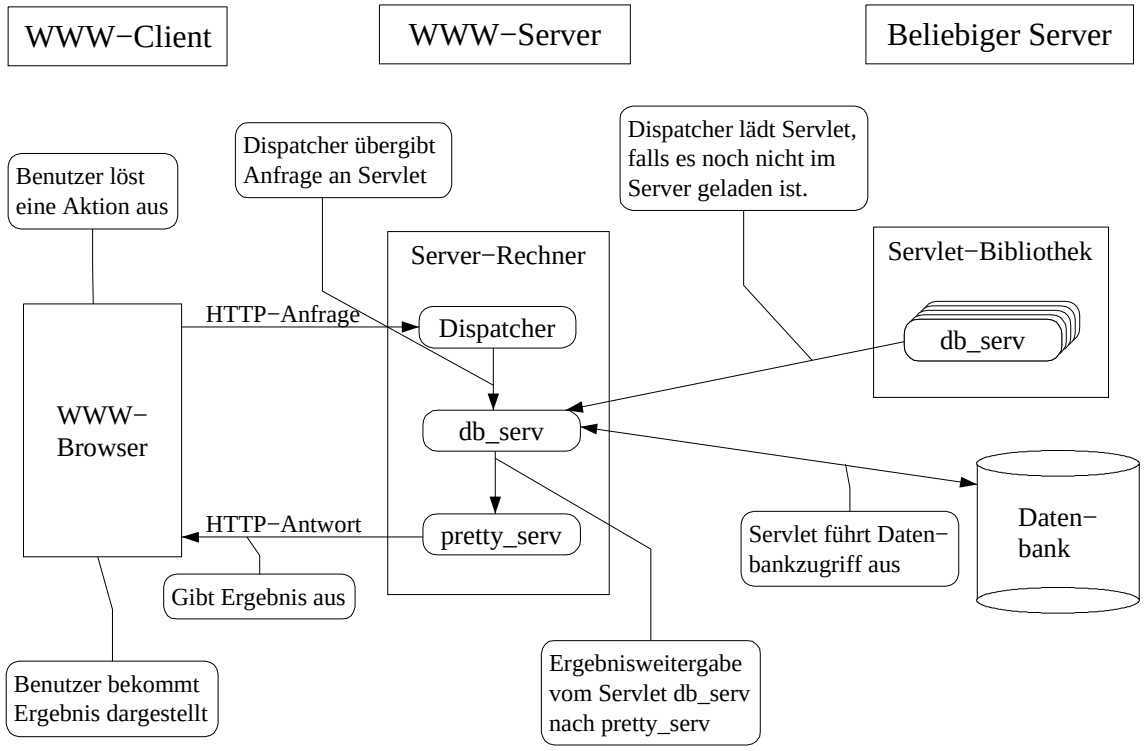

Abbildung 3: Direkter Aufruf eines Servlets mit Datenbankanfrage

Abbildung 3 zeigt einen möglichen Ablauf, wie er bei einem *direkten* Aufruf eines Servlets mit Datenbankanfrage aussehen kann. Ein direkter Aufruf ist eine Anfrage, bei der die zu la− dende URL ein Servletprogramm adressiert. Dieser gilt z. B. beim Aufruf eines Verweises oder beim Abschicken eines Formulars. Das Servlet muß hierbei das gesamte Gerüst des WWW−Dokuments samt Header und Body (vgl. Abschnitt 2.1.3) erzeugen. Der Dispatcher ist der Teil des Servers, der ankommenden HTTP−Anfragen Threads zuweist und diese zu den entsprechenden Servlets sendet. Er überprüft auch, ob sich ein aufgerufenes Servlet schon auf dem Server befindet oder noch von einer Servlet−Bibliothek geladen werden muß.

Der Benutzer löst hier durch Anklicken auf eine Aktion eine HTTP−Anfrage aus. Diese wird an den zuständigen servlet−fähigen WWW−Server weitergeleitet und dort vom Dispatcher entgegengenommen. Dieser erkennt, daß das benötigt Servlet (`db_serv`) noch nicht im Ser− ver−Rechner geladen ist. In diesem Fall lädt er es von der zentralen Servlet−Bibliothek und übergibt ihm die Anfrage. `Db_serv` führt die Anfrage aus und greift dabei auf die Datenbank zu. Seine Ausgaben werden dabei an `pretty_serv` zur Weiterverarbeitung übergeben. Die− ses Servlet formatiert `db_serv`'s Ausgabe in HTML und sendet sie an den Browser. Die An− einanderreihung der beiden Servlets bilden dabei eine Servletkette.

Einen Servlet−Aufruf aus einem WWW−Dokument über einen *eingebetteten* Servlet−Befehl zeigt Abbildung 4. Der Unterschied zum direkten Aufruf besteht darin, daß der Servlet−Auf− ruf aus dem aktuellen WWW−Dokument stattfindet und die Programmergebnisse direkt in die Aufrufstelle hineingeschrieben werden. Vom Programm werden lediglich die abgefragten und formatierten Datenbankinformationen erzeugt und nicht das gesamte Gerüst eines WWW− Dokuments. Das Einbinden der Programmergebnisse in das angeforderte Dokument über− nimmt der Serverrechner. Der Ablauf über die Servlet−Kette ist derselbe wie beim direkten

Aufruf in der vorherigen Abbildung.

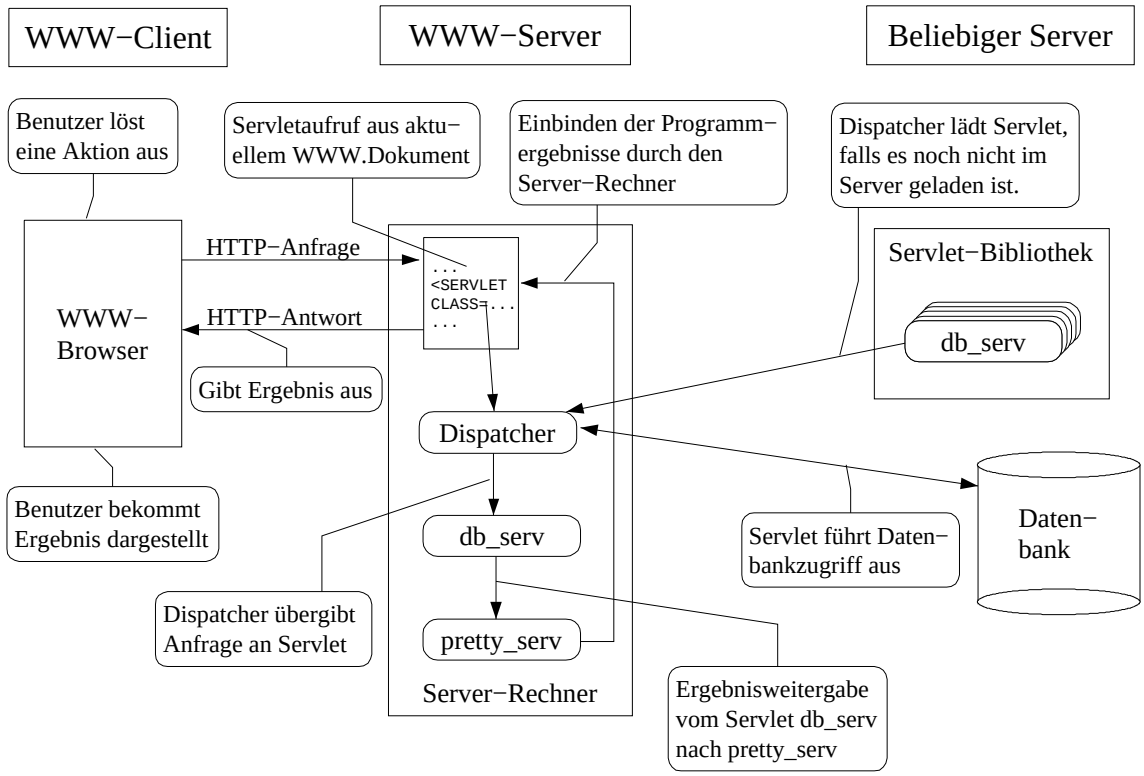

Abbildung 4: Servlet−Aufruf über einen eingebetteten Servlet−Befehl

3 Aufgaben des Generators

Der Generator stellt eine Verbindung zwischen den WWW−Dokumenten in WWW−Servern und einer zentralen Datenbank dar. Er sorgt als eigenständiges Programm für eine Synchronisation der Informationen in der Datenbank mit denen der WWW−Dokumente.

Im ersten Abschnitt dieses Kapitels wird zunächst das Konzept des Generators vorgestellt, danach werden die funktionalen Anforderungen einer HTML−Erweiterung analysiert und darauf aufbauend die HTML−Erweiterung entwickelt und beschrieben. Danach werden die Möglichkeiten dieser Erweiterung anhand einer Beispieldatenbank verdeutlicht. Anschließend wird kurz auf die Benutzerschnittstelle und die Benutzung der Erweiterung eingegangen. Den Abschluß dieses Kapitels bildet eine Betrachtung des möglichen Fehlerverhaltens des Generators.

3.1 Das Konzept des Generators

Die Grundidee des Generators ist, daß Datenbankinhalte mit Hilfe einer HTML−Erweiterung direkt in die WWW−Dokumente eingebunden werden. Diese HTML−Erweiterung ermöglicht dem Benutzer oder Web−Designer die Angabe einer Datenbankanfrage an beliebiger Stelle innerhalb eines WWW−Dokuments samt Gestaltungsangaben für das Anfrageergebnis. Als Gestaltungsmöglichkeiten werden mehrere Darstellungsformen wie z. B. Tabellen oder Listen angeboten. Der Generator fügt die Datenbankinformationen, entsprechend der angegebenen Darstellungsform formatiert, an den dafür vorgesehenen Stellen in den HTML−Quelltext ein. Für eine ausführliche Analyse und Beschreibung der HTML−Erweiterung sei auf den nächsten Abschnitt verwiesen.

Die in Kapitel 2 vorgestellten Ansätze, dynamische Datenbankinformation in WWW−Dokumente einzubinden, haben eines gemeinsam: Sie generieren bei jedem Dokumentenaufruf die Datenbankinformationen durch eine Anfrage neu, auch wenn diese sich gar nicht geändert haben. Somit sind dort für einen Dokumentenaufruf immer zwei Kommunikationsphasen notwendig:

1. Die Kommunikation zwischen dem Browser und dem Server (Dokumentenaufrufphase) und
2. die Kommunikation zwischen Server und Datenbank zur Abfrage der Datenbankinhalte.

Durch den Generator werden beide Phasen getrennt und zu unterschiedlichen Zeiten ausgeführt. Der WWW−Server stellt WWW−Dokumente mit HTML−Erweiterung als normale statische Dokumente zur Verfügung, die bei Bedarf durch den Generator dynamisch aktualisiert werden. Somit wird der Kommunikationsaufwand beim Aufruf eines Dokuments auf das Minimum reduziert, wie es auch bei statischen WWW−Dokumenten notwendig ist. Der Kommunikationsaufwand mit der Datenbank entfällt beim Aufruf. Datenbankanfragen werden bei Bedarf unabhängig von den Dokumentenaufrufen durch den Aktualisierungsvorgang des Generators durchgeführt. Der Aufrufer des Generators bestimmt, wann dies notwendig ist und kann den Generatorlauf auf einen Zeitpunkt mit geringer Auslastung des Servers verlagern.

Abbildung 5 zeigt die Phase des Dokumentenaufrufs. Der Benutzer ruft hier in seinem Browser ein WWW−Dokument auf, das die HTML−Erweiterung enthält. Dies führt zu einer HTTP−Anfrage, die der Browser an den zuständigen WWW−Server schickt. Der Server liest das angeforderte Dokument von seinem lokalen Dateisystem und schickt es an den Browser. Dieser stellt es dann dem Benutzer in seinem Anzeigefenster dar. Der Generator und die Da−

tenbank treten nicht in Aktion, da die statischen Dokumente die erforderlichen Datenbankin−halte schon enthalten.

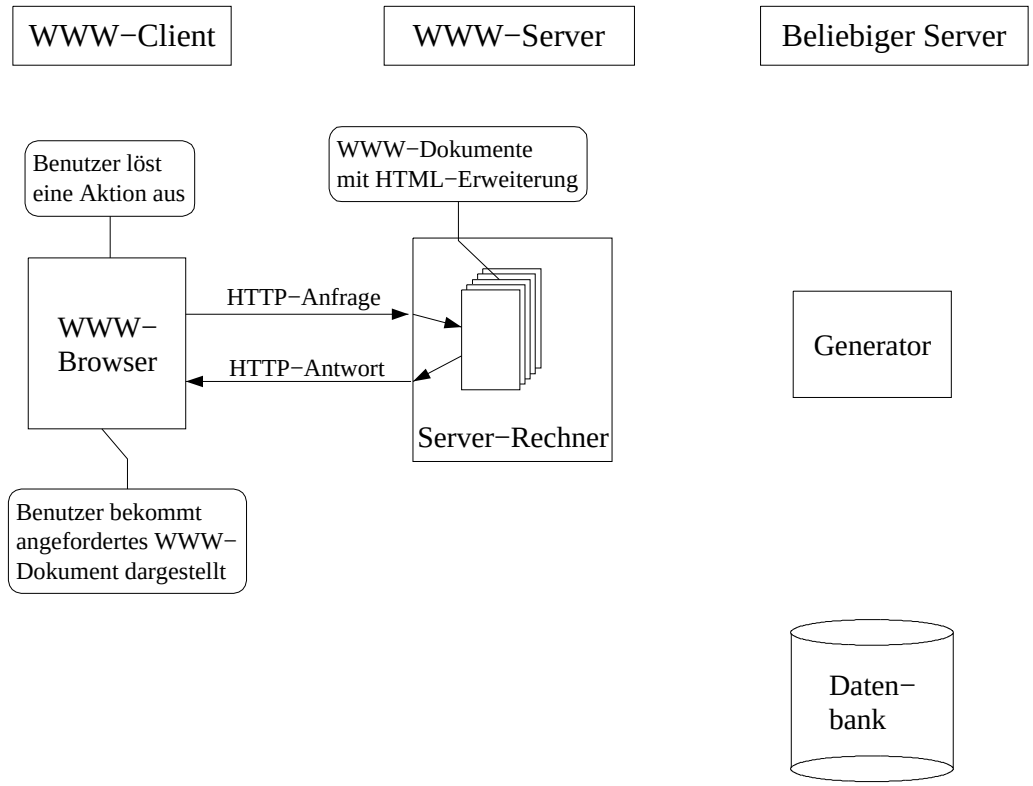

Abbildung 5: Die Dokumentenaufrufphase unter Verwendung eines Generators

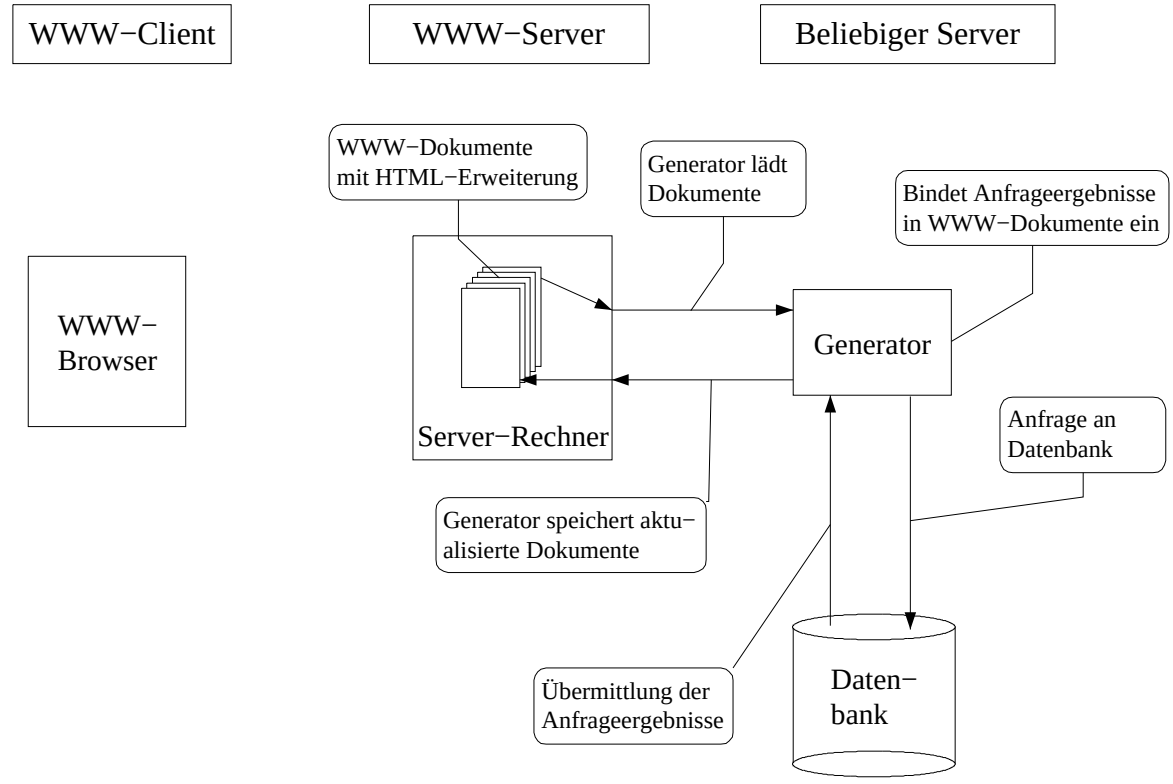

Abbildung 6: Der Aktualisierungsvorgang der statischen WWW−Dokumente durch den Generator

Abbildung 6 zeigt den Aktualisierungsvorgang der statischen WWW−Dokumente durch den Generator, der unabhängig von Browseranfragen stattfindet. Vor jedem Aktualisierungsvorgang baut der Generator einmalig eine Verbindung zur Datenbank auf. Danach lädt er die Dokumente des WWW−Servers und durchsucht sie nach HTML−Erweiterungen. Bei jeder gefundenen HTML−Erweiterung führt er die enthaltene Datenbankanfrage aus und schreibt deren Anfrageergebnisse in der angegebenen Darstellungsform an die angegebenen Stellen hinein. Jedes aktualisierte Dokument wird danach im Dateisystem des Servers überschrieben. Zum Schluß des Aktualisierungsvorgangs wird die Datenbankverbindung durch den Generator geschlossen.

Die Vorteile des Generators gegenüber CGI sind:

- Die gleiche Performance wie bei statischen WWW−Dokumenten ohne Datenbankinhalte. Ein Aufruf startet keinen Prozeß auf dem Server und führt keine Datenbankanfrage samt Verbindungsaufbau und −abbau durch.
- Die Stabilität und Zuverlässigkeit des Servers wird nicht beeinflußt.

Der Vorteil des Generators gegenüber Servlets ist:

- Ein Aufruf startet keine Datenbankanfrage samt Verbindungsaufbau und −abbau.

Der Generator stellt gegenüber CGI und Servlets eine benutzer− und änderungsfreundliche Alternative dar. Er befreit den Benutzer oder Web−Designer von der sonst notwendigen Programmierarbeit.

Die HTML−Erweiterung des Generators ermöglicht die Angabe einer Datenbankanfrage samt Gestaltungsangaben für das Anfrageergebnis durch einfache Angaben, die vom Benutzer leicht zu erlernen sind. Änderungen in diesen Angaben sind schnell durchgeführt und die HTML−Codeerzeugung übernimmt der Generator.

Ein Nachteil des Generators ist, daß in den WWW−Dokumenten nicht immer die aktuellen Datenbankinhalte enthalten sind. Dies hängt davon ab, wann z. B. ein Administrator es für notwendig hält, einen Generatorlauf durchzuführen.

3.2 Funktionale Anforderungen

Der HTML−Quelltext der WWW−Dokumente soll für den Generator Informationen enthalten, aus denen er die gewünschten Datenbankinhalte an die gewünschte Stelle z. B. als Tabelle oder Liste in den HTML−Quelltext einsetzt. Dazu ist eine entsprechende *HTML−Erweiterung* notwendig, die auch die Angabe einer SQL−Abfrage für die Informationserzeugung erlaubt. Dazu sollen in dieser Arbeit zunächst die Gestaltungsmöglichkeiten analysiert werden, die HTML für Datenbankinhalte bietet. Danach wird darauf aufbauend die HTML−Erweiterung entwickelt.

3.2.1 HTML−Gestaltungsmöglichkeiten für Datenbankinhalte

HTML bietet folgende Gestaltungsmöglichkeiten, alphanumerische Datenbankinhalte in den HTML−Quelltext einzufügen:

1. Tabelle im präformatierten Text definieren
   HTML−Befehle: `<PRE>`, `<LISTING>`, `<XMP>` oder `<PLAINTEXT>`

   Präformatierter Text wird immer in nicht−proportionaler Schrift dargestellt. Zeilenumbrü−

che, Leerzeichen und Tabulatoren werden so wiedergegeben, wie sie beim Editieren eingegeben wurden. Neben dem Befehl <PRE> gibt es noch drei weitere Befehle für präformatierten Text, nämlich <LISTING>, <XMP> und <PLAINTEXT>. Alle drei sollen jedoch laut Empfehlung des W3−Konsortiums [W3] nicht mehr verwendet werden und werden deshalb hier nicht weiter betrachtet.

Der <PRE>−Befehl wird für nicht proportionalen Text verwendet. Entsprechender Text wird im Rahmen dieser Arbeit zur Gestaltung von Datenbankinhalten nicht umgesetzt.

2. Aufzählungslisten (Bulletlisten) mit verschiedenen Aufzählungszeichen (Bulletypen)
   HTML−Befehl: <UL>

   Sie sind für einspaltige Tabellen geeignet, wenn zum Hervorheben der einzelnen Punkte Aufzählungszeichen gewünscht werden. Da kaum ein Unterschied zur Menüliste <MENU> und Directoryliste <DIR> besteht, reicht hier die Betrachtung der Aufzählungslisten aus.

3. Numerierte Listen mit verschiedenen Numerierungsarten
   HTML−Befehl: <OL>

   Sie sind geeignet, wenn Datenbankinformationen geordnet werden sollen. Dies ist der Fall, wenn z. B. ein Überblick über die Anzahl der Informationen gewünscht wird.

4. Definitionslisten
   HTML−Befehl: <DL>

   Definitionslisten werden in der Regel für Glossare verwendet. Sie bestehen aus einer Liste von Einträgen, die einen Ausdruck wie z. B. einen Fachbegriff mit dazugehöriger Definition beschreiben. Sie werden im Rahmen dieser Arbeit nicht weiter verfolgt.

5. nur Textabsätze (entspricht einer einspaltigen Tabelle)
   HTML−Befehl: <P>

   Eine Möglichkeit ist, jedes Datenbanktupel in einen eigenen Textabsatz mit daraus resultierendem Absatzabstand für jede Tupelzeile zu schreiben. Eine andere ist die, alle Tupel in einen einzigen Textabsatz mit einem Zeilenumbruch nach jedem Tupel zu schreiben (kompakte Darstellung).

   Beide Möglichkeiten sind gut für einspaltige Tabellen geeignet, wenn keine Aufzählungszeichen gewünscht werden.

6. Tabelle
   HTML−Befehl: <TABLE>

   Da im relationalen Datenmodell eine Datenbank aus einer Anzahl sog. Relationen besteht, die sich näherungsweise als Tabellen auffassen lassen, sind Tabellen die geeignetste Form Datenbankinformationen darzustellen. HTML unterscheidet Tabellen mit und ohne Gitterzellen. Tabellen mit Gitternetzlinien sind gut geeignet für mehrspaltige Tabellen, bei denen viele Informationen durch die Gitterlinien getrennt gut lesbar sein sollen. Tabellen ohne Gitternetzlinien sind eine Darstellungsform, bei denen Gitternetzlinien übertrieben wirken würden.

Im obigen Überblick ist ersichtlich, daß Tabellen im präformatierten Text und Definitionslisten für die Darstellung von Datenbankinformationen nicht weiter bearbeitet werden. Somit werden in dieser Arbeit *Aufzählungslisten*, *Numerierte Listen*, *Textabsätze* und *Tabellen* mit und ohne Gitternetzlinien als Gestaltungsformen für Datenbankinformationen in WWW−Dokumenten realisiert.

3.2.2 HTML−Erweiterung

Die HTML−Erweiterung ist eine einfache Sprache, die in den HTML−Kommentar geschrieben wird, damit sie bei der Darstellung des WWW−Dokuments im Browser ignoriert wird. Diese Erweiterung soll die Angabe einer SQL−Anfrage ermöglichen, deren Ergebnis entweder in Textabsätzen, Listen oder Tabellen dargestellt werden kann.

Um die SQL−Anfrage für die HTML−Erweiterung formulieren zu können, ist es notwendig, diese dort einzubetten. Dafür gibt es zwei Möglichkeiten:

1. Einbindung der SQL−Anfrage in die Erweiterung oder
2. Auslagerung der SQL−Anfrage in eine eigene Datei und diese über einen Verweis in die Erweiterung einbinden.

Da die Einbindung der SQL−Anfrage in die HTML−Erweiterung einen direkten Bezug zwischen den in der SQL−Anfrage selektierten Attributen und ihrer Anordnung in der Darstellungsform erlaubt, wird die erste Alternative realisiert. Zur besseren Abgrenzung in der HTML−Erweiterung wird die SQL−Anfrage in eckige Klammern eingeschlossen.

Wie im vorherigen Abschnitt analysiert, sollen die SQL−Anfrageergebnisse in Textabsätzen, Aufzählungslisten, numerierten Listen oder Tabellen dargestellt werden können. Die Angabe der gewünschten Darstellungsform findet in einer Art Prozeduraufruf statt, in der die gewünschte Anordnung der selektierten Attribute angegeben wird. In dieser Anordnung soll es möglich sein, jedes Attribut durch textuelle Ergänzungen einzuschließen.

Darstellungsform	**Aufruf in HTML−Erweiterung**
Textabsatz	`Textabsatz (Attributliste)`
TextabsatzKompakt	`TextabsatzKompakt (Attributliste)`
Aufzählungsliste	`Liste (Attributliste)`
Numerierte Liste	`Liste (Attributliste)`
Tabelle (auch blinde Tabelle)	`Tabelle (Attributliste)`

Um den generierten Datenbankinhalt vom restlichen Inhalt eines HTML−Dokuments besser abgrenzen zu können, wird dieser in einen Startbefehl mit SQL−Anfrage und Darstellungsform und einen Endebefehl eingeschlossen. Da sich in HTML eine Aufzählungsliste von einer numerierten Liste nur durch den Anfangs− und Endebefehl unterscheiden (vgl. Kapitel 2) und ihre Listeneinträge dieselben Befehle verwenden, bietet es sich an, lediglich die Listeneinträge vom Generator durch Aufruf von `Liste` erzeugen zu lassen. Um welchen Listentyp es sich dann wirklich handelt, bestimmt der Benutzer durch den entsprechenden Anfangsbefehl wie `<UL>` oder `<OL>`. Somit können auch innerhalb dieser Anfangsbefehle außerhalb der HTML−Erweiterung spezielle HTML−Formatierungen für die Listen angegeben werden. Analog wird auch mit Tabellen verfahren. Ob es sich dabei um eine blinde Tabelle oder um eine mit Gitternetzlinien handelt, entscheidet eine Angabe im Anfangsbefehl (`<TABLE>`) vor der HTML−Erweiterung.

Für den Endebefehl der Listen oder Tabellen gibt es zwei Möglichkeiten:

1. Der Benutzer schreibt lediglich den Beginnbefehl der Liste bzw. Tabelle vor die HTML−Erweiterung; den Endebefehl dazu schreibt der Generator.
2. Der Benutzer schreibt sowohl den Beginnbefehl der Liste bzw. Tabelle vor die HTML−Erweiterung als auch den entsprechenden Endebefehl hinter die HTML−Erweiterung.

Da die erste Möglichkeit den Listenaufruf der HTML−Erweiterung in zwei getrennte Aufrufe für Aufzählungslisten und Numerierungslisten spalten würde, wird die zweite Möglichkeit gewählt. Den Endebefehl für Tabellen könnte der Generator ohne weiteres erzeugen, doch wird der Einheitlichkeit wegen auch hier die zweite Möglichkeit gewählt.

Da Hyperlinks einen wichtigen Bestandteil von HTML−Dokumenten bilden, sollen auch sie mit Hilfe von selektierten Attributen gebildet werden können. Ihr Aufruf in der HTML−Erweiterung benötigt als Parameter das Verweisziel und den dazugehörigen Verweistext, der beliebig viele Attribute enthalten darf. Er soll wie folgt aussehen:

```
Ref(Verweisziel; Verweistext)
```

WWW−Dokumente sind Hypermediadokumente, die neben Text (alphanumerische Datenbankinhalte) auch Grafiken, Ton− und Videosequenzen enthalten können. Eingebundene Grafiken gehören mittlerweile zum Standard und können von jedem modernen Browser dargestellt werden. Sie sollen deshalb in der Darstellungsform mit Hilfe von selektierten Attributen eingebunden werden können. Ton− und Videosequenzen benötigen dagegen für ihre Anzeige entsprechend installierte Plugins. Plugins sind in der Regel herstellerabhängig und nicht jeder Anwender hat in der Regel alle verfügbaren Plugins installiert. Da eine Anzeige im Gegensatz zu Grafiken nicht bei jedem Anwender garantiert ist, soll in dieser Arbeit auf die Einbindung von Ton− und Videosequenzen verzichtet werden.

In HTML werden Grafiken durch Grafikreferenzen eingebunden (vgl. Kapitel 2). Der Aufruf in der HTML−Erweiterung benötigt als Parameter die Zieldatei und Angaben für die Eigenschaften einer Grafik. Zu den Eigenschaften einer Grafik zählen Attribute, die zum Formatieren von dieser notwendig sind. So kann man in diesen Attributen Angaben über die Größe der Grafik machen, den Textfluß um die Grafik und die Ausrichtung der Grafik im Text bestimmen. Ohne die Möglichkeit dieser Angaben würden immer die Standardeinstellungen gelten. Die Eigenschaften werden im `<IMG>`−Befehl den entsprechenden Attributen zugewiesen.

Für die Angabe der Grafikeigenschaften gibt es zwei Möglichkeiten:

1. Die Attributwerte des `<IMG>`−Befehls werden als Parameter übergeben.

   Hierbei müssen die Attribute in ihrer Reihenfolge festgelegt werden. Andere Attributangaben, insbesondere neue HTML−Standards, sind ohne Codeänderung des Generators nicht möglich.

2. Alle gewünschten Attribute samt ihrer Werte werden am Stück als String übergeben.

   Da hier keine Festlegung auf bestimmte Attribute und deren Reihenfolge nötig ist und jedes Attribut mit seinem Wert angegeben wird, bietet diese Lösung eine hohe Flexibilität. Hierbei können auch Attribute neueren HTML−Standards ohne Programmänderung übergeben werden.

Da die zweite Möglichkeit eine hohe Flexibilität sowohl für den Benutzer als auch für neue HTML−Standards bietet, soll diese realisiert werden. Somit sieht der Grafikreferenzaufruf wie folgt aus:

```
RefGrafik(Zieldatei; Eigenschaften)
```

Formatierungen in den Darstellungsformen werden durch die HTML−Erweiterung nicht unterstützt, sondern sie finden weiterhin über die HTML−Befehle statt. Formatierungen, die die gesamte Liste oder Tabelle betreffen, sind somit möglich. Zusätzlich ist es möglich, selektierte Attribute über die textuellen Ergänzungen mit HTML−Befehlen zu formatieren. Dagegen sind Formatierungen, die bei Tabellen einzelne Zellen, Zeilen oder Spalten betreffen nicht möglich. Würde man diese realisieren wollen, so wären viele Formatierungsparameter für

einzelne Zellen, Zeilen und Spalten notwendig. Dies würde einen großen Aufwand darstellen, der über diese Arbeit hinausgeht.

Im folgenden soll dieses Problem verdeutlicht und gezeigt werden, warum Formatierungen einzelner Zellen, Zeilen oder Spalten in dieser Lösung nicht möglich sind.

Um in HTML Formatierungen für einzelne Zeilen anzugeben, sind Attributangaben im einleitenden <TR>−Befehl der entsprechenden Zeile notwendig. Formatierungen für einzelne Zellen werden im <TD>−Befehl angegeben. HTML bietet keine Möglichkeit, Formatierungen für ganze Spalten anzugeben. Diese müssen durch Angaben in allen Zellen der entsprechenden Spalte gemacht werden. Da die <TR>− und <TD>−Befehle zum Generatorergebnis gehören und in der Darstellungsform keine Attributangaben aus obigem Grund vorgesehen sind, hat der Benutzer keine Möglichkeit, Formatierungen dieser Art anzugeben. Die beiden folgenden Beispiele zeigen Formatierungen, die in der Darstellungsform Tabelle nicht möglich sind.

- Festlegen der Hintergrundfarbe für einzelne Zeilen oder Zellen einer Tabelle:

```
<TABLE BORDER>
   <TR BGCOLOR=#CCFFFF>
      <TD>Zeile 1 Spalte 1 = hellblau (gilt für Zeile)</TD>
      <TD>Zeile 1 Spalte 2 = hellblau (gilt für Zeile)</TD>
   </TR>
   <TR BGCOLOR=#CCFFCC>
      <TD BGCOLOR=#CCFFCC>Zeile 2 Spalte 1 = hellgrün</TD>
         (gilt für diese Zelle)
      <TD BGCOLOR=#FFCCFF>Zeile 2 Spalte 2 = hellviolett</TD>
         (gilt für diese Zelle)
   </TR>
</TABLE>
```

- Ausrichtung der Zelleninhalte von einzelnen Zeilen oder Zellen einer Tabelle:

```
<TABLE BORDER>
   <TR>
      <TH ALIGN=LEFT>Kopfzelle: 1. Zeile, 1. Spalte</TH>
      <TH>Kopfzelle: 1. Zeile, 2. Spalte</TH>
      <TH ALIGN=RIGHT>Kopfzelle: 1. Zeile, 3. Spalte</TH>
   </TR>
   <TR>
      <TD>Datenzelle: 2. Zeile, 1. Spalte</TD>
      <TD ALIGN=CENTER>Datenzelle: 2. Zeile, 2. Spalte</TD>
      <TD ALIGN=RIGHT>Datenzelle: 2. Zeile, 1. Spalte</TD>
   </TR>
</TABLE>
```

Der HTML−Quelltext soll beliebig viele nicht geschachtelte HTML−Erweiterungen enthalten dürfen. So wird es dem Benutzer ermöglicht, beliebig viele SQL−Anfrageergebnisse in sein HTML−Dokument einzubinden.

Beschreibung der HTML−Erweiterung

Die Grundstruktur der HTML−Erweiterung besteht aus einem Startbefehl, der Angabe der gewünschten Darstellungsform, einer SQL−Anfrage und einem Endebefehl. Für den Inhalt dazwischen ist nur der Generator zuständig. Er generiert das SQL−Anfrageergebnis in der angegebenen Darstellungsform dort hinein. Es dürfen dort keine weiteren HTML−Befehle stehen, da diese sonst beim nächsten Generatorlauf entfernt werden. Eigene HTML−Befehle (z. B. eine Listenüberschrift) kann der Benutzer vor die HTML−Erweiterung schreiben.

Leerzeichen zwischen den einzelnen Befehlen sind erlaubt.

Die vier Aufrufsmöglichkeiten der Darstellungsformen in der HTML−Erweiterung:

- Textabsatz:

```
<!--[\Start Textabsatz (Attibutliste)
    [SQL−Anfrage]
//−−>
hier steht das Generatorergebnis:
<P> . . . </P>
<P> . . . </P>
...       ...
<!--[\Ende−−>
```

- TextabsatzKompakt:

```
<!--[\Start TextabsatzKompakt (Attibutliste)
    [SQL−Anfrage]
//−−>
hier steht das Generatorergebnis:
<P>
. . . <BR>
. . . <BR>
. . .
</P>
<!--[\Ende−−>
```

- Liste für eine Aufzählungsliste bzw. Numerierungsliste:

```
<OL> bzw. <UL> für eine Numerierungsliste bzw. Aufzählungsliste
Es sind beliebige Formatierungen in beiden Tags erlaubt.
<!--[\Start List (Attibutliste)
    [ SQL−Anfrage ]
//−−>
hier steht das Generatorergebnis:
<LI> . . .</LI>
<LI> . . .</LI>
. . .
<!--[\Ende−−>
</OL> bzw. </UL>
```

- Tabelle:

```
<TABLE>
<TR>     <!--mögliche Kopfzeilendefinition vom Benutzer−−>
         <TH>...</TH>
           :  :
           .  .
</TR>
<!--[\Start Tabelle (Attibutliste; Attributliste; ...)
    [ SQL−Anfrage ]
//−−>
hier steht das Generatorergebnis:
```

```
<TR>
        <TD> ...</TD>
        <TD> ...</TD>
        ...
</TR>
<TR>
        <TD> ...</TD>
        <TD> ...</TD>
        ...
</TR>
 ...
<!--[\Ende-->
</TABLE>
```

Die Attributliste:

Die Attributliste gibt an, wie und in welcher Reihenfolge die in der SQL−Anfrage selektierten Attribute in der Darstellungsform angeordnet werden. Dabei können selektierte Attribute beliebig oft in der Darstellungsform verwendet werden. Zusätzlich können die Attribute in textuelle Ergänzungen eingeschlossen werden. Textuelle Ergänzungen werden in Anführungszeichen gesetzt. Um dem Benutzer die Möglichkeit zu geben, in einer textuellen Ergänzung Anführungszeichen zu verwenden, sollen die beide Typen »"..."« und »'...'« unterschieden werden. In textuellen Ergänzungen können außerdem HTML−Befehle zur Formatierung bestimmter Attribute als Strings angegeben werden.

In einer Attributliste werden alle Attribute durch Kommas getrennt. Die Aufrufe der Darstellungsformen `Textabsatz`, `TextabsatzKompakt` und `Liste` werden als einspaltige Tabelle aufgefaßt und können genau eine Attributliste mit beliebig vielen Attributen enthalten. Der Aufruf der Darstellungsform `Tabelle` darf dagegen beliebig viele Attributlisten durch Semikolons getrennt enthalten. Jede Attributliste wird in einer eigenen Tabellenspalte angeordnet. Somit werden durch Kommas getrennte Attribute in einer eigenen Tabellenspalte angeordnet. Zusätzlich kann jede Attributliste auch Hyperlinks (Hypertext−Referenzen) und Grafikreferenzen enthalten.

Hyperlinks:

Hypertext−Referenzen ermöglichen die Erzeugung von Referenzen im Generatorergebnis. Das Aufrufschema sieht wie folgt aus:

```
Ref(Verweisziel; Verweistext)
```

Das Verweisziel kann genau ein Attribut enthalten, der Verweistext beliebig viele. Dieser Referenzaufruf wird in folgenden HTML−Befehl umgesetzt:

```
<A HREF="Verweisziel">Verweistext</A>
```

Mit diesem Referenzaufruf lassen sich alle in HTML üblichen Verweisziele realisieren:

- Verweis in gleiches Dokument: `"#Zieldef"`
- Verweis auf eine Datei im gleichen Verzeichnis: `"datei.html"`
- Verweis auf WWW−Adresse: `"http://www.informaitk.uni-stuttgart.de"`
- Verweis zu einer FTP−Adresse: `"ftp://ftp.uni-stuttgart.de/"`
- E−Mail−Verweis: `"mailto:s.muenz@euromail.com"`

- E−Mail−Verweis mit vordefiniertem Subjekt:
  `"mailto:s.muenz@euromail.com?subjekt=Feedback"`

Grafikreferenzaufruf:

Grafikreferenzen ermöglichen das Einbinden von Bildern in das Generatorergebnis. Sie haben folgendes Aufrufschema:

```
RefGrafik(Zieldatei; Eigenschaften)
```

Dieser Grafikreferenzaufruf wird in folgenden HTML−Befehl umgesetzt:

```
<IMG SRC="Zieldatei" Eigenschaften>
```

Die Zieldatei darf sich in einem beliebigen Verzeichnis des lokalen Rechners oder auf einem beliebigen WWW−Server befinden. In den Eigenschaften können nähere Angaben für die Grafikdarstellung in Form von Attributen innerhalb des `<IMG>`−Befehls gemacht werden. Alle angegebenen Attribute werden zusammen als String in den `<IMG>`−Befehl geschrieben und vom Browser interpretiert. Wichtige Attribute für die Darstellung von Grafiken innerhalb von Datenbankinhalten sind:

- `WIDTH=` [Pixel] für die Breite und `HEIGHT=` [Pixel] für die Höhe einer Grafik
- `ALIGN=TOP`, `ALIGN=MIDDLE` oder `ALIGN=BOTTOM` richten die Grafik auf die Basislinie der Textzeile entsprechend oben, mittig oder unten aus. Diese Attribute sind geeignet, um eine Grafik innerhalb von alphanumerischen Attributen anzuordnen.
- `BORDER=` [Pixel] definiert ein Rahmen um die Grafik.

Die Attribute `ALIGN=LEFT` bzw. `ALIGN=RIGHT` veranlassen den Browser die Grafik linksbündig bzw. rechtsbündig auszurichten und den darauffolgenden Fließtext rechts bzw. links neben der Grafik anzuzeigen. Da beide die Grafik genau am Rand ausrichten, sind sie für die Darstellung von Datenbankinhalten unbrauchbar.

3.3 Die Beispieldatenbank

Im folgenden soll die Verwendung der HTML−Erweiterung anhand einer Beispieldatenbank verdeutlicht werden. Die gewählte Beispieldatenbank enthält Personenbeschreibungen, wie sie üblicherweise in Abteilungen, Instituten oder Fakultäten auftreten. Dafür enthält die Datenbank eine Beispielrelation Person mit den Attributen Pnr (Personalnummer), Name, Vname (Vorname), Abteilung, Raum, Tel (Telefon), Fax, Email (E−Mail) und Hpage (Verweis auf Homepage). Das Attribut Pnr bildet den Primärschlüssel, da sich jede Person allein durch die Angabe ihrer Personalnummer eindeutig identifizieren läßt.

Person								
Pnr	**Name**	**Vname**	**Abtei−lung**	**Raum**	**Tel**	**Fax**	**Email**	**Hpage**
4675676	Berger	Uwe	ZDI	0.169	321	null	Uwe.Berger@infor−matik.uni−stuttgart.de	http://www.informatik.uni−stuttgart.de/menschen/ber−ger.html
5133435	Rantzau	Ralf	AS	2.010	433	null	Ralf.Rantzau@infor−matik.uni−stuttgart.de	http://www.informatik.uni−stuttgart.de/ipvr/as/perso−nen/rantzau.html’
6876536	Schiele	Frank	AS	2.010	433	420	Frank.Schiele@infor−matik.uni−stuttgart.de	http://www.informatik.uni−stuttgart.de/ipvr/as/perso−nen/schiele.html

Beispiele für die HTML−Erweiterung

Die folgenden Beispiele zeigen die Verwendung der HTML−Erweiterung in allen fünf Dar−stellungsformen, wie Textabsatz, TextAbsatzKompakt, Aufzählungsliste, numerierte Liste und Tabelle. Dabei wird für die SQL−Anfrage die obige Beispieldatenbank Person verwendet.

Es werden in diesen Beispielen lediglich die für die HTML−Erweiterung notwendigen Befeh−le dargestellt ohne das übrige HTML−Gerüst wie Header und Body. Für nähere Informationen dazu sei auf Kapitel 2 verwiesen.

Beispiel für Textabsätze:

```
...
<H4>Namensliste</H4>
<!−−[\Start Textabsatz(Name, ", " Vname)
    [select Name, Vname
     from Person;]
//−−>
hier steht das Generatorergebnis:
<P>Berger, Uwe</P>
<P>Rantzau, Ralf</P>
<P>Schiele, Frank</P>
<!−−[\Ende−−>
...
```

Broweransicht:

Namensliste

Berger, Uwe

Rantzau, Ralf

Schiele, Frank

Beispiel für kompakte Textabsätze:

```
...
```

```
<H4>Telefonliste</H4>
<!--[\Start TextabsatzKompakt(Name, ", " Vname, " Tel.: " Tel)
    [select Name, Vname, Tel
     from Person;]
//-->
hier steht das Generatorergebnis:
<P>
Berger, Uwe Tel.: 321<BR>
Rantzau, Ralf Tel.: 433<BR>
Schiele, Frank Tel.: 433<BR>
</P>
<!--[\Ende-->
...
```

Browseransicht:

Telefonliste

Berger, Uwe Tel.: 321
Rantzau, Ralf Tel.: 433
Schiele, Frank Tel.: 433

Beispiel einer Aufzählungsliste:

```
...
<H4>Abteilungsliste</H4>
<UL type=disc>
<!--[\Start Liste
(Pnr, " " Name, ", " Vname " ", "Abteilung: " Abteilung)
    [select Pnr, Name, Vname, Abteilung
     from Person
     order by Abteilung;]
//-->
hier steht das Generatorergebnis:
<LI>4675676 Berger, Uwe Abteilung: ZDI</LI>
<LI>5133435 Rantzau, Ralf Abteilung: AS</LI>
<LI>6876536 Schiele, Frank Abteilung: AS</LI>
<!--[\Ende-->
</UL>
...
```

Browseransicht:

Abteilungsliste

- 4675676 Berger, Uwe Abteilung: ZDI
- 5133435 Rantzau, Ralf Abteilung: AS
- 6876536 Schiele, Frank Abteilung: AS

Beispiel einer numerierten Liste:

Dieses Beispiel enthält eine durchgezählte Namensliste.

```
...
<H4>Namensliste</H4>
<OL>
<!--[\Start Liste(Name, ", " Vname)
    [select Name, Vname
     from Person;]
//-->
hier steht das Generatorergebnis:
<LI>Berger, Uwe</LI>
<LI>Rantzau, Ralf</LI>
<LI>Schiele, Frank</LI>
<!--[\Ende-->
</OL>
...
```

Browseransicht:

Namensliste

1. Berger, Uwe
2. Rantzau, Ralf
3. Schiele, Frank

Beispiel einer numerierten Liste mit Homepageverweis:

```
...
<H4>Telefonverzeichnis</H4>
<OL>
<!--[\Start Liste(Ref(Hpage; Name, ", " Vname) " ", "Tel.: " Tel)
    [select Name, Vorname, Hpage, Tel
     from Person;]
//-->
hier steht das Generatorergebnis:
<LI><A HREF="http://www.informatik.uni-stuttgart.de/menschen/
berger.html">Berger, Uwe</A> Tel.: 321</LI>
<LI><A HREF="http://www.informatik.uni-stuttgart.de/ipvr/as/
personen/rantzau.html">Rantzau, Ralf</A> Tel.: 433</LI>
<LI><A HREF="http://www.informatik.uni-stuttgart.de/ipvr/as/
personen/schiele.html">Schiele, Frank</A> Tel.: 433</LI>
<!--[\Ende-->
</OL>
...
```

Browseransicht:

Telefonverzeichnis

1. Berger, Uwe Tel.: 321
2. Rantzau, Ralf Tel.: 433
3. Schiele, Frank Tel.: 433

Beispiel einer Tabelle:

Dieses Beispiel zeigt, daß in der SQL−Anfrage selektierte Attribute beliebig oft in der Darstellungsform verwendet werden können. So wird hier die E−Mail−Adresse sowohl als Verweisziel als auch als Verweistext verwendet.

```
...
<H3>Personentabelle</H3>
<TABLE border>
<TR>    <!−−Kopfzeilendefinition−−>
    <TH>Name</TH>
    <TH>E−Mail</TH>
    <TH>Raum</TH>
    <TH>Telefon</TH>
</TR>
<!−−[\Start Tabelle(Ref(Hpage; Name, ", " Vname); Ref("mailto:"
Email; Email); Raum; "Tel.: " Tel)
    [select Name, Vname, Hpage, Email, Tel, Raum
     from Person;]
//−−>
hier steht das Generatorergebnis:
<TR>
    <TD><A HREF="http://www.informatik.uni−
stuttgart.de/menschen/ berger.html">Berger, Uwe</A></TD>
    <TD><A HREF="mailto:Uwe.Berger@informatik.uni−stuttgart.
de">Uwe.Berger@informatik.uni−stuttgart.de</A></TD>
    <TD>0.170</TD>
    <TD>Tel.: 321</TD>
</TR>
<TR>
    <TD><A HREF="http://www.informatik.uni−stuttgart.de/ipvr/as/
personen/rantzau.html'">Rantzau, Ralf</A></TD>
    <TD><A HREF="mailto:Ralf.Rantzau@informatik.uni−stuttgart.
de">Ralf.Rantzau@informatik.uni−stuttgart.de</A></TD>
    <TD>2.010</TD>
    <TD>Tel.: 433</TD>
</TR>
<TR>
    <TD><A HREF="http://www.informatik.uni−stuttgart.de/ipvr/as/
personen/schiele.html">Schiele, Frank</A></TD>
    <TD><A HREF="mailto:Frank.Schiele@informatik.uni−stuttgart.
de">Frank.Schiele@informatik.uni−stuttgart.de</A></TD>
    <TD>2.010</TD>
    <TD>Tel.: 433</TD>
</TR>
<!−−[\Ende−−>
</TABLE>
...
```

Browseransicht:

Personentabelle

Name	E−Mail	Raum	Telefon
Berger, Uwe	Uwe.Berger@informatik.uni−stuttgart.de	0.170	Tel.: 321
Rantzau, Ralf	Ralf.Rantzau@informatik.uni−stuttgart.de	2.010	Tel.: 433
Schiele, Frank	Frank.Schiele@informatik.uni−stuttgart.de	2.010	Tel.: 433

Beispiel des Telefonverzeichnisses der Fakultät Informatik:

Dieses Beispiel zeigt einen kurzen Ausschnitt aus dem Telefonverzeichnis der Fakultät Infor−matik in Form einer blinden Tabelle (keine Gitternetzlinien). Das Attribut `CELLPADDING` gibt den Randabstand der Zellendaten zum Zellenrand in Pixeln an.

```
...
<H1>Telefon − Fakult&auml;t Informatik 0711/7816−...</H1>
<H2>Stand: 07.01.98</H2>
<TABLE CELLPADDING=5>
<TR>    <!−−Kopfzeilendefinition−−>
    <TH>Name</TH>
    <TH>Abt.</TH>
    <TH>Raum</TH>
    <TH>Telefon</TH>
</TR>
<!−−[\Start Tabelle
(Ref(Hpage; Name, ", " Vname); Abteilung; Raum;  Tel)
    [select Name, Vname, Hpage, Email, Tel, Raum
     from Person;]
//−−>
hier steht das Generatorergebnis:
<TR>
    <TD><A HREF="http://www.informatik.uni−
stuttgart.de/menschen/ berger.html">Berger, Uwe</A></TD>
    <TD>ZDI</TD>
    <TD>0.170</TD>
    <TD>321</TD>
</TR>
<TR>
    <TD><A HREF="http://www.informatik.uni−stuttgart.de/ipvr/as/
personen/rantzau.html">Rantzau, Ralf</A></TD>
    <TD>AS</TD>
    <TD>2.010</TD>
    <TD>433</TD>
</TR>
<TR>
    <TD><A HREF="http://www.informatik.uni−stuttgart.de/ipvr/as/
personen/schiele.html">Schiele, Frank</A></TD>
    <TD>AS</TD>
    <TD>2.010</TD>
    <TD>433</TD>
</TR>
<!−−[\Ende−−>
```

```
</TABLE>
...
```

Browseransicht:

Telefon − Fakultät Informatik 0711/7816−...

Stand: 07.01.98

Name	Abt.	Raum	Telefon
Berger, Uwe	ZDI	0.170	321
Rantzau, Ralf	AS	2.010	433
Schiele, Frank	AS	2.010	433

3.4 Benutzerschnittstelle

Eine Aktualisierung der gewünschten WWW−Dokumente soll im Rahmen dieser Arbeit zunächst kommandozeilenorientiert stattfinden. Dabei soll es die Möglichkeit geben, eine Liste bestehend aus einzelnen Dateien bzw. ganzen Verzeichnissen aktualisieren zu können.

Aufruf des Generators durch:

```
generator <files>
```

Der Parameter `<files>` enthält die zu aktualisierenden Dateien oder Verzeichnisse. Der Benutzer kann davon beliebig viele durch Leerzeichen getrennt angeben.

Beispiel:

```
generator Telefonliste.html /usr/ipvr/as/personen
```

Dieser Aufruf des Generators aktualisiert im aktuellen Verzeichnis die Datei `Telefonliste.html` und im Verzeichnis `/usr/ipvr/as/personen` alle enthaltenen Dateien.

Darüber hinaus besteht die Möglichkeit für eine automatische Aktualisierung, den Kommandozeilenaufruf in ein Unix−Shell−Script einzubinden und dieses in regelmäßigen Abständen nachts oder am Wochenende laufen zu lassen.

3.5 Benutzung der HTML−Erweiterung

Für den Benutzer oder Web−Designer kann die HTML−Erweiterung als optionale Funktion betrachtet werden. Er wird die Erweiterung benutzen, wenn er in seine WWW−Dokumente zentrale Datenbankinformationen einbinden will.

Die Kommandozeilenbenutzung gibt dem Seitendesigner die Möglichkeit seine neu erstellten WWW−Dokumente sofort auf eventuell enthaltene Fehler und gewünschtes Aussehen zu testen, ohne auf eine allgemein stattfindende Aktualisierung warten zu müssen.

3.6 Fehlerverhalten

Treten beim Aktualisieren eines WWW−Dokuments Fehler auf, so sind für den Benutzer in erster Linie diejenigen Fehler relevant, die ihm beim Schreiben der HTML−Erweiterung unterlaufen sind. Um diese beheben zu können, benötigt er möglichst konkrete und aussagekräftige Fehlermeldungen.

Bei den für den Benutzer relevanten Fehlertypen handelt sich entweder um

- Syntax−Fehler, die beim Parsen der HTML−Erweiterung auftreten können, oder um
- SQL−Fehler, die vom SQL−Interpreter gemeldet werden.

Bei Fehlermeldungen sollen beide Fehlertypen deutlich unterscheidbar sein.

Weiterhin existieren zwei unterschiedliche Anforderungen des Generatorbenutzers an die Fehlerausgabe.

- Ein Kommandozeilenbenutzer, der sein WWW−Dokument auf eventuell enthaltene Fehler und gewünschtes Aussehen testet, wünscht eine sofortige Rückmeldung in seinem Kommadozeilenfenster.
- Bei Fehlern, die bei einer automatischen Aktualisierung auftreten, machen direkte Bildschirmmeldungen keinen Sinn, da diese in der Regel keinen Benutzer erreichen. Hier ist es sinnvoll, die Fehlermeldungen zu speichern, damit die Benutzer oder Administratoren Probleme bei fehlerhaften WWW−Dokumenten zu einem späteren beliebigen Zeitpunkt nachvollziehen können.

Für die Speicherung der Fehlermeldungen gibt es mehrere Möglichkeiten:

- Nachricht per E−Mail an den Ersteller des WWW−Dokuments.

  Ein Problem besteht jedoch darin, den Ersteller des WWW−Dokuments zu ermitteln. Darüber hinaus ist für einen Aufrufer des Dokuments nicht ersichtlich, wenn es veraltete Informationen enthält.

- Ausgabe aller Meldungen in eine zentrale Datei.

  Hierbei besteht das Problem, wer die Meldungen zu welcher Zeit liest. Es besteht die Gefahr, daß viele Meldungen von den Erstellern der WWW−Dokumente spät oder gar nicht gelesen werden. Somit können fehlerhafte Dokumente mit veralteten Informationen lange unerkannt bleiben.

- Ausgabe der Meldungen in eine temporäre Datei, die vom Generator beim Aktualisieren angelegt wird. Diese wird im Normalfall gelöscht und bleibt lediglich bei einem aufgetretenen Fehler erhalten.

  Hierbei besteht die Gefahr, daß eventuell vorhandene temporäre Dateien unerkannt bleiben. Damit würden auch hier fehlerhafte Dokumente mit veralteten Informationen lange erhalten bleiben.

- Ausgabe der Meldungen in das original WWW−Dokument innerhalb eines Kommentares.

  Auch hier werden Aktualisierungsfehler lange unerkannt bleiben, da WWW−Dokumente

in der Regel nicht im HTML−Quelltext angeschaut und Kommentare im Anzeigefenster des Browsers nicht angezeigt werden.

- Ausgabe der Meldungen in das original WWW−Dokument als normaler Dokumenteninhalt und nicht als Kommentar. Zusätzlich wird ein Hyperlink auf die alte Version des Dokuments eingefügt.

  Diese Variante ermöglicht es sowohl dem Ersteller als auch dem Aufrufer des Dokuments, Aktualisierungsfehler zu erkennen. Ein einfaches Laden des Dokumentes in den Browser ist ausreichend, das Suchen in bestimmten Dateien entfällt. Dem Aufrufer wird es trotzdem ermöglicht, die alte Version zu laden.

Aufgrund der oben dargestellten Vorteile der letzten Variante sowohl für den Ersteller als auch für den Aufrufer, wird diese in dieser Arbeit realisiert.

Würden beide Fehlerausgaben gleichzeitig im Generator realisiert werden, so würde die Änderung der Orginaldatei und das Speichern des Orginalinhaltes in einer anderen Datei den Kommandozeilenbenutzer, der sein WWW−Dokument testen will, bei vorhandenen Fehlern behindern. Er müßte die Datei mit dem Orginalinhalt wieder in den Orginalnamen zurückbenennen, die gemeldeten Fehler in ihr verbessern und diese dann neu aktualisieren. Trotzdem sollen beide Fehlerausgaben gleichzeitig im Generator realisiert werden, da in dieser Arbeit die automatische Aktualisierung im Vordergrund steht.

4 Entwurf des Generators

Im ersten Abschnitt dieses Kapitels werden zunächst die Schnittstellen des Generators dargestellt. Im anschließenden Abschnitt wird das für die Schnittstelle des Parsers notwendige Ablaufschema für das Parsen der HTML−Erweiterung anhand eines Grammatikentwurfs entwickelt.

4.1 Die Schnittstellen

Dieser Abschnitt erläutert die für den Generator entwickelten Schnittstellen. Da eine Realisierung in der Programmiersprache Java vorgesehen ist, werden die entworfenen Klassen an der Java−Syntax angelehnt. Bei der Entwicklung der Schnittstellen des Generators wurde auf Wiederverwendbarkeit Wert gelegt. So wurde die eigentliche Funktionalität des Generators in den Parser ausgelagert. Deshalb wird die Klasse `Parser` zuerst erläutert. Da das eigentliche Ablaufschema anhand einer Grammatik im folgenden Abschnitt entwickelt wird, werden in der Klasse `Parser` lediglich die für die Schnittstelle `Generator` benötigten Methoden erläutert. Im Anschluß daran wird die Klasse `Tupelzeile` vollständig behandelt. Zum Schluß dieses Abschnitts wird eine mögliche Integration der Klasse Parser am Beispiel des Generator gezeigt. Diese läßt sich auf beliebige Programme wie Editoren oder Entwurfswerkzeuge für WWW−Dokumente oder Datenbanken übertragen.

4.1.1 Die Schnittstelle des Parsers

Die Schnittstelle des Parser wurde so entworfen, daß nach dem Erzeugen eines Parser−Objekts beliebig viele WWW−Dokumente aktualisiert werden können, ohne jedesmal ein neues Objekt erzeugen zu müssen. Die Klassendefinition sieht wie folgt aus:

```
public class Parser {
  //Konstruktoren
  public Parser (String driver, String url);
  public Parser (String driver, String url, String user,
                 String password);

  //Parser−Methoden
  public void init(String file) throws FileNotFoundException;
  final public void HTMLErweiterung() throws ParseException;
  public void closeFile();
  public void handleError(ParseException e);
  public void close();
} //Parser
```

Im folgenden wird die Funktionsweise der Methoden erläutert.

`public Parser (String driver, String url)`

Dieser Konstruktor lädt den angegebenen Treiber und versucht eine Verbindung zu der in `url` angegebenen Datenbank aufzubauen. Zusätzlich bereitet er die Datenbank durch Erzeugung eines `Statement`−Objekts für das Senden von SQL−Anweisungen vor und erzeugt ein `Tupelzeilen`−Objekt für das Abspeichern der Zeilenstruktur für die in HTML zu formatierenden Datenbankinhalte. Dieser Konstruktor wird verwendet, wenn die Datenbank keinen Benutzernamen und kein Paßwort benötigt.

`public Parser (String driver, String url, String user, String password)`

Dieser Konstruktor lädt den angegebenen Treiber und versucht eine Verbindung zu der in `url` angegebenen Datenbank mit einer Identifikation des Benutzers aufzubauen. Er erzeugt auch ein `Statement`− und `Tupelzeilen`−Objekt.

`public void init(String file) throws FileNotFoundException`

Benennt die übergebene Datei, die in der Regel die Struktur <file>.html hat, in <file>.old, setzt den Eingabestrom des Parsers auf diese Datei, und legt eine temporäre Datei <file>.html für die Ausgabe des aktualisierten WWW−Dokumenteninhaltes an. Um den Eingabestrom des Parsers auf diese neue Datei setzen zu können, ist die Erzeugung eines neuen `FileInputStream` notwendig. Dies geschieht durch den Aufruf `this(new java.io.FileInputStream(file))`. Zusätzlich wird das `Tupelzeile`−Objekt mit dieser Datei initialisiert.

`final public void HTMLErweiterung() throws ParseException`

Parst die in der `init`−Methode angegebene Datei und schreibt deren aktualisierten Inhalt in die Datei <file>.html.

`public void closeFile()`

Löscht die Datei <file>.old mit ihrem ursprünglichen Inhalt und schließt die Datei <file>.html mit aktualisiertem Inhalt nach einem erfolgreichen Parsevorgang durch die Methode `HTMLErweiterung`.

Diese Methode soll nur nach einer erfolgreichen Beendigung der Methode `HTMLErweiterung` aufgerufen werden.

`public void handleError(ParseException e)`

Führt die Fehlerbehandlung nach einem Parsefehler durch. Dabei wird durch die `ParseException` übergebene Fehlermeldung sowohl auf dem Bildschirm ausgegeben als auch in die Datei <file>.html geschrieben. Die Fehlermeldung wird nicht als HTML−Kommentar geschrieben, damit sie im Browser angezeigt wird. Zusätzlich wird in die Datei <file>.html ein Hyperlink auf die alte Version <file>.old des WWW−Dokuments eingefügt und diese danach geschlossen. Die Datei <file>.old bleibt im Fehlerfall erhalten.

Diese Methode soll nur in einer catch−Anweisung zur Behandlung der aufgetretenen ParseException aufgerufen werden.

`public void close()`

Schließt das im Parser−Objekt angelegte `Statement`−Objekt und die Datenbankverbindung.

4.1.2 Die Schnittstelle der Tupelzeile

Die Tupelzeile ist eine Datenstruktur zur Abspeicherung der Zeilenstruktur für die in HTML zu formatierenden Datenbankinhalte, wie sie vom Benutzer in der Darstellungsform angegeben wurde. In ihr wird abgespeichert, wie der Generator jedes Tupel der Ergebnismenge formatieren soll.

Ein so formatiertes Tupel enthält neben Attributen Strings und HTML−Befehle, die auch als Strings aufgefaßt werden sollen. Somit muß die Datenstruktur die Elementtypen Attribute und String aufnehmen und das gesamte formatierte Tupel mit den aktuellen Attributwerten ausgeben können.

Eine dazu geeignete Datenstruktur ist eine einfach verkettete Liste, die Objekte beider unterschiedlicher Klassentypen, einmal `String` und einmal `Attribut`, aufnehmen kann. Um dies zu erreichen, wird die abstrakte Klasse `Element` eingeführt und die beiden benötigten Klassen `Attribut` und `ElemString` davon abgeleitet. Eine verkettete Liste deren Elemente vom Klassentyp `Element` sind, kann Objekte aller seiner Unterklassen aufnehmen und somit auch `Attribut` und `ElemString`.

Die Klasse `ElemString` enthält eine `write`−Methode, die den enthaltenen String sofort ausgibt, die Klasse `Attribut` eine `write`−Methode, die ihren Attributwert des aktuellen Tupels aus der Ergebnisrelation aus der Datenbank ausliest und danach ausgibt. Ein beim Auslesen aus der Datenbank aufgetretener Fehler wird in eine `ParseException` umgewandelt und an den Parser zur Weiterbehandlung weitergegeben. Die Ausgabemethode `writeTupel` gibt die gesamte Liste aus. In ihr wird die `write`−Methode der Klasse `Element` aufgerufen. Dynamisches Binden sorgt dafür, daß immer die `write`−Methode des tatsächlichen Klassentyps für das enthaltene Objekt, entweder `String` oder `Attribut`, aufgerufen wird.

Die Methode `initFile` ermöglicht, daß die `Tupelzeile` mit einer neuen Datei initialisiert werden kann. Die Methoden `initListe` und `initResultSet` ermöglichen, daß innerhalb eines WWW−Dokuments die Zeilenstrukturen weiterer HTML−Erweiterungen abgespeichert werden können. Die so beschriebene Klassendefinition der `Tupelzeile` sieht wie folgt aus:

```
class Tupelzeile {

  private Element zeile, letzteElem;
  private FileWriter fr;
  private ResultSet rs;

  //Verwendung des Default−Konstruktors, deshalb hier keiner
angegeben

  public initFile(FileWriter fr) {
    this.fr = fr;
  } //initFile

  //löscht alle Elemente der Tupelzeile
  public void initListe() {
    zeile = null; letzteElem = null;
  } // initListe

  //weist neuen ResultSet zu
  public void initResultSet(ResultSet rs) {
    this.rs = rs;
  } //initResultSet

  public void addAttribut(String name) {
    Attribut a = new Attribut();
    a.next = null;

    if (zeile == null) {  //leere Liste
       letzteElem = a; zeile = a;
```

```
    } else {  //Objekt in Liste einhaengen
       letzteElem.next = a; letzteElem = a;
    } // endif

    a.attribut = name;
  } // addAttribut

  public void addString(String s) {
    ElemString es = new ElemString();
    es.next = null;

    if (zeile == null) { //leere Liste
       letzteElem = es; zeile = es;
    } else {
       letzteElem.next = es; letzteElem = es;
    } // endif

    es.string = s;
  } //addString

  public void writeTupel() throws ParseException {
    Element e = zeile;

    while (e != null) {
       e.write(); e = e.next;
    } //endwhile

  } //writeTupel

abstract private class Element {
  Element next;
  abstract public void write() throws ParseException;
} //Element

private class Attribut extends Element {
  String attribut;

  public void write() throws ParseException {
    try {
      fr.write(rs.getString(attribut));
    } // endtry
    catch(IOException e) {
      System.out.println("IOExeption in Tupelzeile beim Schreiben
von Attribut:");
      System.out.println(e.getMessage());
    } // endcatch
    catch(SQLException e) {
      //Fehlerbehandlung an Parser weitergeben
      throw new ParseException("SQLExeption in Tupelzeile beim
```

```
Lesen von Attribut:\n" + e.getMessage());
    } // endcatch
  } // write
} //Attribut

private class ElemString extends Element {
  String string;

  public void write() throws ParseException {
    try {
      fr.write(string);
    } // endtry
    catch(IOException e) {
      System.out.println("IOException in Tupelzeile beim
Schreiben von String:");
      System.out.println(e.getMessage());
    } // endcatch

  } //write
} // ElemString

} //Tupelzeile
```

4.1.3 Die Einbindung des Parsers in den Generator

Die Klasse des Generators realisiert die im Abschnitt 3.4 vorgestellte Funktionalität. Der Generator wird hier kommandozeilenorientiert aufgerufen und erhält als Parameter eine Liste bestehend aus einzelnen Dateien bzw. ganzen Verzeichnissen, die zu aktualisieren sind. Da pro Parsevorgang genau eine Datei aktualisiert wird, muß der Generator die einzelnen übergebenen Dateinamen bzw. diese aus den übergebenen Verzeichnissen auslesen und einzeln an den Parser übergeben.

Der Generator verwendet die im Abschnitt 4.1.1 beschriebenen Methoden des Parsers. Zuerst wird ein Objekt des Parser mit übergebenem Datenbanktreiber, JDBC−URL, Benutzernamen und Paßwort angelegt. Danach werden in zwei geschachtelten Schleifen die einzelnen übergebenen Dateinamen ausgelesen und an den Parser übergeben. Die äußere Schleife liest alle übergebenen Parameter einzeln ein. Von jedem wird ein File−Objekt angelegt und überprüft, ob es sich dabei um ein Verzeichnis oder eine Datei handelt. Handelt es sich dabei um ein Verzeichnis, so werden über die `list`−Methode des `File`−Objekts alle Dateien dieses Verzeichnisse eingelesen und in einer inneren Schleife einzeln an die `updateFile`−Methode zur Aktualisierung an den Parser übergeben. Handelt es sich dagegen um eine Datei, so wird diese sofort über die `updateFile`−Methode an den Parser übergeben. Wurden alle Dateien aktualisiert, wird das Parser−Objekt geschlossen.

Die `updateFile`−Methode faßt die Methodenaufrufe zusammen, die notwendig sind, um eine Datei zu aktualisieren. Zuerst wird der Parser mit der übergebenen Datei initialisiert, danach die Startproduktion der HTML−Erweiterung aufgerufen, die den eigentlichen Aktualisierungsvorgang durchführt, und zum Schluß wird die aktualisierte Datei geschlossen. Tritt ein Parsefehler auf, so wird eine `ParseException` ausgelöst und diese an die `handleError`−Methode zur Verarbeitung übergeben. Im folgenden ist die Klassendefinition

des Parser dargestellt:

```
public class Generator {

  public static void main(String args[]) {
    if (args.length >= 1) {
      File file;
      Parser parser = new Parser(DBDriver, JDBC−URL, User,
                                 Password);
      for (int i=0; i<args.length, i++) {
        file = new File(arg[i]);
        if (file.isDirectory()) {
          String files[] = file.list();
          for (int j=0; j<files.length; j++) }
            updateFile(files[j]);
          } //for
        } else { //isFile
          updateFile(args[i]);
      } //for
      parser.close();

    } else {
      System.out.println("Generator:  Usage is one of:");
      System.out.println("java Generator < inputfiles");
      System.out.println("OR");
      System.out.println("java Generator inputfiles");
      return;
    } //if
  } //main

  private void updateFile(String file) {
    try {
      parser.init(file);
      parser.HTMLErweiterung();
      parser.closeFile();
    } catch (ParseException e) {
      parser.handleError(e);
    } catch (FileNotFoundException e) {
      System.err.println("Generator: Datei " + file +
                         " nicht gefunden");
  } //updateFile
}
```

4.2 Der Parser−Entwurf für die HTML−Erweiterung

Bevor in diesem Abschnitt der Parser genauer erläutert wird, soll zunächst auf einige Grund−lagen aus den Bereichen Formale Sprachen und Übersetzerbau (auch Compilerbau ge−nannt) [Aho 88] eingegangen werden. Im engeren Sinne geht es im Übersetzerbau um die Übersetzung einer Programmiersprache in eine andere (meist eine Maschinensprache). In die−ser Arbeit handelt es sich nicht um einen klassischen Übersetzungsvorgang, sondern um das Durchsuchen von WWW−Dokumenten nach HTML−Erweiterungen, dem Parsen von diesen Dokumenten und dem Ausführen von Aktionen. Dabei sind hierfür lediglich die Überset−zungsphasen der lexikalischen Analyse, der syntaktischen Analyse und der semantischen Anayse von Bedeutung.

Der Scanner (lexikalische Analysator) übernimmt die Aufgaben der lexikalischen Analyse. Seine Hauptaufgabe besteht darin, den Quelltext in eine Folge lexikalischer Symbole (= Ter−minalsymbole für die syntaktische Analyse) zu zerlegen und diese dem Parser zu übergeben, bedeutungslose Zeichen wie Leerzeichen zu überlesen und lexikalische Fehler zu finden. Der Parser ist für die syntaktische Analyse zuständig und überprüft die HTML−Erweiterung auf syntaktische Korrektheit.

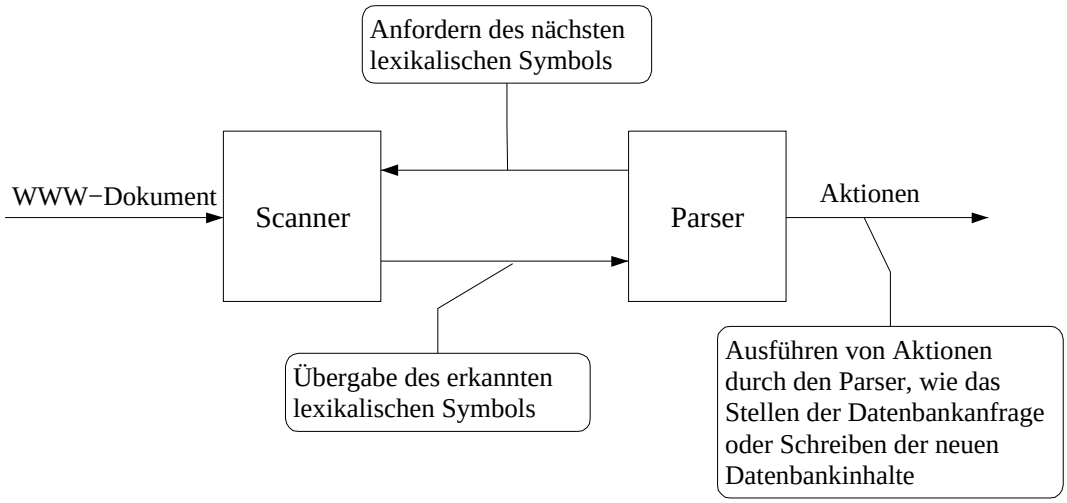

Abbildung 7: Das Zusammenspiel zwischen Scanner und Parser

Abbildung 7 verdeutlicht das Zusammenspiel zwischen Scanner und Parser. Der Scanner wird üblicherweise als Unterprogramm des Parsers betrachtet. Er tritt erst in Aktion, wenn der Par−ser von ihm das nächste lexikalische Symbol anfordert. Dann liest er im Quelltext des WWW−Dokuments solange Zeichen, bis er das nächste lexikalische Symbol erkannt hat. Dieses wird dann dem Parser zur Weiterverarbeitung übergeben und auf syntaktische Kor−rektheit überprüft. Zusätzlich führt der Parser semantische Aktionen aus wie das Stellen einer Datenbankanfrage und das Formatieren der Datenbankinhalte.

Lexikalische Symbole können meist mit regulären Ausdrücken beschrieben und daher mit endlichen Automaten erkannt werden. Der Scanner ist ein endlicher Automat mit einem ge−meinsamen Anfangszustand für alle zu erkennenden Symbole. In jedem Endzustand des Au−tomaten gilt das gelesene Symbol als erkannt.

Die syntaktische Struktur der HTML−Erweiterung wird durch eine kontextfreie Grammatik beschrieben. Diese dient als Grundlage für die Konstruktion des Parsers. In der Theorie der formalen Sprachen unterscheidet man bei Grammatiken Terminalsymbole und Nichtterminal−symbole.

Terminalsymbole (in der lexikalischen Analyse auch lexikalische Symbole oder Token genannt) sind Zeichenfolgen des Quellprogramms, die sich aus syntaktischer Sicht nicht weiter zerlegen lassen.
Nichtterminalsymbole werden zur Gliederung bzw. Strukturbeschreibung der Grammatik verwendet. Für jedes Nichtterminalsymbol gibt es eine Ableitungsregel (oder Produktion), die beschreibt, in welche Folge von Terminalsymbolen und/oder Nichtterminalsymbolen das Nichtterminalsymbol zerlegt werden kann.

Zur Formulierung der Ableitungsregeln wird hier die Wirthsche EBNF [Pom 93] (extended Backus−Naur−Form) verwendet, in der die folgenden Konventionen angewendet werden:

- Eine Ableitungsregel beginnt mit dem Namen des Nichtterminalsymbols, das definiert werden soll, gefolgt von einem Gleichheitszeichen und der eigentlichen Definition. Die gesamte Regel wird durch einen Punkt abgeschlossen.

  = ist definiert als
  . Ende der Produktion

- Zur besseren Unterscheidung beginnen hier die Nichtterminalsymbole mit großen und die Terminalsymbole mit kleinen Anfangsbuchstaben.
- Wenn ein Terminalsymbol durch eine feste Zeichenfolge repräsentiert wird, wird die Zeichenfolge unter Anführungszeichen gesetzt ("...").
- Symbole, die in einer festen Reihenfolge auftreten müssen, werden durch Leerzeichen getrennt hintereinander geschrieben. Die Regel A = x y. bedeutet, daß das Symbol x unmittelbar gefolgt von y steht.
- Ein senkrechter Strich trennt Alternativen. Die Aneinanderreihung von Symbolen bindet stärker als das Alternativzeichen. Die Regel C = x y | z w. bedeutet, daß das Nichtterminalsymbol C entweder aus xy oder aus zw besteht.
- Runde Klammern werden zur Zusammenfassung zusammengehöriger Teile benutzt. Sie können z. B. benutzt werden, um obige Vorrangregel zu durchbrechen. Die Regel D = x (y | z) w. bedeutet, daß das Nichtterminal D immer mit einem x beginnt und mit einem w endet und in der Mitte entweder ein y oder ein z enthält.

  (... | ...) genau eine Alternative aus der Klammer muß stehen

- Eckige Klammern werden zur Kennzeichnung möglicher Auslassungen verwendet.

  [...] Inhalt der Klammer kann stehen oder nicht

- Geschwungene Klammern kennzeichnen Wiederholungen.

  { ... } Inhalt der Klammer kann n−fach stehen, $n \geq 0$

Da die EBNF nur Regeln der Form $A = \beta$ mit $A \in V_N$ enthält, wobei V_N das Alphabet der Nichtterminalsymbole ist, ist jede beschriebene Grammatik in EBNF kontextfrei. Dies geht aus der Chomsky−Hierachie [Rec 97] für Grammatiken hervor.

Insgesamt besteht eine *kontextfreie Grammatik* G = (N, T, P, S) aus vier Komponenten:

1. T: Eine Menge von *Terminalsymbolen*
2. N: Eine Menge von *Nichtterminalsymbolen*
3. P: Eine Menge von *Ableitungsregeln* (Produktionen) für jedes Nichtterminalsymbol
4. S: Ein ausgezeichnetes Nichtterminalsymbol als *Startsymbol*

4.2.1 Grammatik für die HTML−Erweiterung

Dieser Abschnitt beschreibt und erläutert die Grammatik für die in Kapitel 3 entwickelte HTML−Erweiterung. Im folgenden werden die vier Komponenten der kontextfreien Gram−matik, die Liste der Nichtterminalsymbole, die Liste der Terminalsymbole, das Startsymbol und die Ableitungsregeln nacheinander beschrieben.

Beschreibung der kontextfreien Grammatik `G = (N,T,P,S):`

Liste der *Nichtterminalsymbole* `N`:

```
Gestaltung, Attributliste, Referenz, Grafikreferenz, Attribut,
Url, Eigenschaften
```

Liste der *Terminalsymbole* `T`:

```
start, zeichenBeliebig
statementSQL, string, textabsatz, textabsatzKompakt, liste,
tabelle, ref, refGrafik, lrKlammer, rrKlammer, semikolon, komma,
htmlCommentEnd, parameter,
ende
```

Mit `start` wird der Startbefehl `<!−−[\Start` der HTML−Erweiterung erkannt, mit `zeichenBeliebig` ein beliebiges Zeichen. Dieses kann auch ein Steuerzeichen sein. Mit `statementSQL` wird die eingebundene SQL−Anfrage erkannt und `string` erkennt eine in Anführungszeichen eingeschlossene Zeichenkette.

Mit `textabsatz`, `textabsatzKompakt`, `liste`, `tabelle`, `ref`, `refGrafik` werden die entsprechenden Schlüsselwörter für die Aufrufe der Darstellungsformen und Referenzen er−kannt.

`lrKlammer` und `rrKlammer` erkennen eine linke runde und eine rechte runde Klammer, `semikolon` ein Semikolon und `komma` ein Komma. `htmlCommentEnd` erkennt den Ende−Befehl `//−−>` eines mehrzeiligen Kommentars. `parameter` erkennt die Parameter in den Aufrufen der Darstellungsformen und Referenzen.

Mit `ende` wird der Endebefehl `<!−−[\Ende−−>` der HTML−Erweiterung erkannt.

Das *Startsymbol* `S`:

```
HTMLErweiterung
```

Die *Ableitungsregeln* (Produktionen) `P`:

```
HTMLErweiterung =
      {zeichenBeliebig}
      {
      start
      Gestaltung
      statementSQL
      htmlCommentEnd
      hier steht im Quelltext das Generatorergebnis: Es wird wie ein
      Kommentar vom Scanner überlesen und wird nicht als Token an
      den Parser zurückgegeben.
      Ende
      {zeichenBeliebig}
      }.
```

Die Ableitung HTML−Erweiterung beginnt mit einem beliebigem Zeichen. Dieses darf belie−big oft oder gar nicht vorkommen. Dies gewährleistet, daß vor dem Auftreten der ersten

HTML−Erweiterung ein beliebiger Inhalt stehen darf, der Zeichen für Zeichen an den Parser übergeben wird und von diesem in das neu generierte WWW−Dokument geschrieben werden kann.

Danach folgt die eigentliche HTML−Erweiterung, eingeschlossen in den Start− und Endebefehl, gefolgt von einem beliebigen Zeichen, das beliebig oft oder gar nicht stehen darf. Die gesamte HTML−Erweiterung mit den folgenden beliebigen Zeichen darf wiederum beliebig oft oder gar nicht vorkommen.

Innerhalb der HTML−Erweiterung werden dann die Gestaltung für die Darstellungsformen gefolgt von der SQL−Anfrage angegeben und mit einem Kommentarende abgeschlossen. Im Quelltext des WWW−Dokuments beginnt hier der generierte Datenbankinhalt. Da dieser lediglich überlesen werden muß und er keine weiteren Aktionen erfordert, reicht es, wenn er vom Scanner überlesen wird und an den Parser nichts zurückgeliefert wird. Den Abschluß des Datenbankinhaltes bildet dann der Endebefehl der HTML−Erweiterung.

Normalerweise erkennt ein Scanner an jeder beliebigen Stelle des Quelltexts die gleiche Menge von Tokens (= Terminalsymbole in der Grammatik). Aufgrund des hier in der Grammatik enthaltenen Terminalsymbols `zeichenBeliebig`, was auf jedes Zeichen im Quelltext zutrifft, würde diese Realisierung des Scanners zu Problemen im Parser führen.

Beispiel:
Der Quelltext vor der ersten HTML−Erweiterung enthält eine öffnende runde Klammer, was dem Terminalsymbol `lrKlammer` entspricht. Würde nun der Scanner überall im Quelltext nach allen möglichen Tokens (Terminalsymbolen) suchen, so würde er die öffnende Klammer nicht wie gewünscht als beliebiges Zeichen zurückliefern, sondern als Token linke runde Klammer. Der Parser hat bisher beliebige Zeichen erkannt und erwartet entweder wieder ein beliebiges Zeichen oder den Startbefehl der HTML−Erweiterung (Terminalsymbol `start`). Ein zurückgeliefertes Token `lrKlammer` veranlaßt den Parser zur Ausgabe der Fehlermeldung, daß er entweder `zeichenBeliebig` oder `start` erwartet, aber keine `lrKlammer`.

In einem WWW−Dokument mit HTML−Erweiterung gibt es drei Bereiche, in denen jeweils eine andere Menge von Tokens erkannt werden muß. Der erste Bereich geht vom Anfang des Quelltextes oder vom Ende der vorher enthaltenen HTML−Erweiterung bis einschließlich des Startbefehls der nächsten HTML−Erweiterung. In diesem Bereich werden die Token `zeichenBeliebig` und `start` erkannt. Der nächste Bereich erstreckt sich von `gestaltung` bis einschließlich `htmlCommentEnd`. In ihm werden die Tokens `statementSQL`, `string`, `textabsatz`, `textabsatzKompakt`, `liste`, `tabelle`, `ref`, `refGrafik`, `lrKlammer`, `rrKlammer`, `semikolon`, `komma`, `htmlCommentEnd` und `parameter` erkannt. Der letzte Bereich schließt sich daran bis einschließlich des Tokens `ende` an. Die nachfolgenden beliebigen Zeichen zählen dann wieder zum ersten Bereich.

Damit die Schlüsselwörter `<!−−[\Start`, `<!−−[\Ende−−>`, `//−−>`, `Textabsatz`, `TextabsatzKompakt`, `Liste`, `Tabelle`, `Ref`, `RefGrafik` und die Zeichen "(", ")", ";" und "," im Scanner den drei unterschiedlichen Erkennungsmodi genau zugeordnet werden können, sind sie nicht als feste Zeichenfolgen in der Grammatik enthalten.

```
Gestaltung =
       (textabsatz
      | textabsatzKompakt
      | liste) lrKlammer Attributliste rrKlammer
      | tabelle lrKlammer Attributliste
        { semikolon Attributliste } rrKlammer.
```

Die Ableitungsregel `Gestaltung` beschreibt alle möglichen Darstellungsformen der HTML−Erweiterung. Der Aufruf der Darstellungsformen Textabsatz, TextabsatzKompakt und Liste enthält genau eine in Klammern eingeschlossene Attributliste. Der Aufruf einer Tabelle darf dagegen beliebig viele durch Semikolons getrennte Attributlisten enthalten.

```
Attributliste =
      ( [string] (Attribut | Referenz | Grafireferenz) [string] )
      { komma
      [string] (Attribut | Referenz | Grafikreferenz) [string] } .
```

Die Elemente einer Attributliste sind Attribute aus der SQL−Anfrage, Referenzaufrufe oder Grafikreferenzaufrufe. Jedes Element darf durch Strings eingeschlossen werden. Eine Attributliste darf beliebig viele durch Kommas getrennte Elemente enthalten.

```
Referenz =
      ref lrKlammer [string] Url [string] semikolon
      [string] (Attribut | Grafikreferenz) [string]
      { komma [string] (Attribut | Grafikreferenz) [string] }
      rrKlammer .
```

Der Aufruf einer Referenz enthält in Klammern eingeschlossen zuerst das Verweisziel durch Angabe der URL. Danach folgt durch ein Semikolon getrennt der Verweistext. Dieser kann entweder ein Attribut aus der SQL−Anfrage oder eine Grafikreferenz sein. Davon können beliebig viele durch Kommas getrennt aneinandergereiht werden. Jedes Element einer Referenz darf wiederum durch Strings eingeschlossen sein.

```
Grafikreferenz =
      refGrafik lrKlammer [string] Url [string] semikolon
      Eigenschaften rrKlammer .
```

Eine Grafikreferenz besteht aus der URL für die Grafik und den Eigenschaften dieser Grafik, die durch ein Semikolon getrennt und in Klammern eingeschlossen sind.

```
Attribut = parameter .
Url = parameter .
Eigenschaften = string .
```

Attribute und URLs werden auf `parameter` abgebildet. Die Eigenschaften einer Grafik bestehen aus einer Zeichenkette.

4.2.2 Lexikalischer Analysator (Scanner)

Wie im vorherigen Abschnitt dargestellt, gibt es in einem WWW−Dokument mit HTML−Erweiterung drei unterschiedliche Bereiche, in denen jeweils eine unterschiedliche Menge von Tokens erkannt werden muß. Um dies zu realisieren, besitzt der Scanner für jeden dieser Bereiche einen Modus, in dem die dafür vorgesehenen Tokens erkannt werden können. Der Scanner befindet sich zu jedem Zeitpunkt in genau einem dieser Modi.

Die Beschreibung der Tokens findet mit regulären Ausdrücken statt. Dabei soll keine formale Einführung in diese gegeben werden, sondern lediglich die Bedeutung der Konstrukte, die hier verwendet werden, angegeben werden. Es findet eine Anlehnung an JavaCC [Jav] statt, wo folgende Konventionen verwendet werden:

- "..." Zeichenfolgen werden in Anführungszeichen gesetzt

- ... | ... genau eine Alternative muß stehen.
- (... | ...) genau eine Alternative aus der Klammer muß stehen
- (...)* der Inhalt der Klammer darf null mal oder beliebig oft stehen
- (...)+ der Inhalt der Klammer darf einmal oder öfter stehen

Die einstelligen Operatoren * und + haben die höchste Priorität, die Konkatenation (Aneinanderreihung) hat die zweithöchste Priorität und das Alternativzeichen | hat die niedrigste Priorität.

Das Konstrukt »[...]« beschreibt die enthaltenen Zeichen. Ein »~« davor beschreibt die nicht enthaltenen Zeichen.

Beispiele zu [...]:
["a"−"z","A"−"Z"] beschreibt die Zeichen vom kleinen a bis zum kleinen z und vom großen A bis zum großen Z
~["a","b"] beschreibt alle Zeichen außer dem kleinen a und dem kleinen b
~[] beschreibt alle Zeichen

Im folgenden werden die regulären Ausdrücke für die drei Modi angegeben, die der Scanner erkennen und als Token zurückliefern soll.

Reguläre Ausdrücke für Modus 0:

```
start: "<!−−[\Start"
zeichenBeliebig: ~[]
```

reguläre Ausdrücke für Modus 1:

```
statementSQL: "[" (~["]"])* "]"
string: """ (~["""])* """ | "'" (~["'"])* "'"
textabsatz: "Textabsatz"
textabsatzKompakt: "TextabsatzKompakt"
liste: "Liste"
tabelle: "Tabelle"
ref: "Ref"
refGrafik: "RefGrafik"
lrKlammer: "("
rrKlammer: ")"
Semikolon: ";"
komma: ","
htmlCommentEnd: "//−−>"
parameter: (alphanumerisches Zeichen und Sonderzeichen)+
```

regulärer Ausdruck für Modus 2:

```
ende: "<!−−[\Ende−−>"
```

Zu diesen regulären Ausdrücken wird für jeden Modus ein *Übergangsdiagramm* [Aho 88] angegeben. Übergangsdiagramme beschreiben den Ablauf der Symbolerkennung, wie er beim Aufruf des Scanners durch den Parser erfolgt.

Die Positionen in einem Übergangsdiagramm werden als Kreise dargestellt und heißen *Zustände*. Die Zustände sind durch Pfeile verbunden, die *Kanten* genannt werden. Kanten, die zwei Zustände verbinden, stellen einen Zustandsübergang dar. Ein Zustandsübergang wird

durchgeführt, wenn die Beschriftung der Kanten mit den Eingabezeichen übereinstimmt. In [Aho 88] wird jede Kante mit genau einem Eingabezeichen beschriftet. In dieser Arbeit wird der Vereinfachung und Übersichtlichkeit wegen davon abgewichen und es werden Kantenbeschriftungen mit Zeichenketten zugelassen.

Ein Zustand jedes Übergangsdiagramms ist als Startzustand gekennzeichnet. In ihm befindet sich die Kontrolle zu Beginn des Erkennungsprozesses. Zustände mit doppelten Kreisen stellen Endzustände dar, in denen ein Token erkannt wurde.

Im folgenden wird für alle drei Modi je ein Übergangsdiagramm angegeben. In allen Diagrammen wird das Token mit der längsten möglichen Übereinstimmung erkannt. Gibt es davon mehrere mit derselben Länge, so wird das in der Anordnung zuerst auftretende Token akzeptiert.

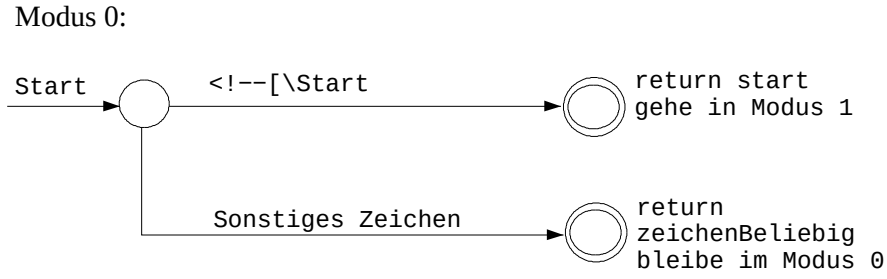

Abbildung 8: Das Übergangsdiagramm für Modus 0

Abbildung 8 zeigt das Übergangsdiagramm für Modus 0. Dieser ist der Startmodus, mit dem der Scanner zu Beginn initialisiert wird. Hier wird der Inhalt vor der HTML−Erweiterung Zeichen für Zeichen (auch Steuerzeichen) gelesen und als Token an den Parser übergeben, damit dieser den Inhalt in das aktualisierte Dokument übernehmen kann. Wird die Zeichenkette `<!--[\Start` gelesen, so wurde der Beginn einer HTML−Erweiterung erkannt. Es wird dann das Token `start` zurückgegeben und in den Modus 2 gewechselt, um die eigentliche HTML−Erweiterung erkennen zu können.

Abbildung 9 zeigt das Übergangsdiagramm für Modus 1. Es ist für die Darstellungsformen samt SQL−Anfrage zuständig und liest die HTML−Erweiterung nach dem Startbefehl bis einschließlich des Kommentarendes (`//-->`). Danach wird in den Modus 3 gewechselt, der für das Überlesen des alten Datenbankinhaltes zuständig ist.

Das Übergangsdiagramm für Modus 2 ist in Abbildung 10 dargestellt und überliest solange die Zeichen des alten formatierten Datenbankinhaltes, bis die Zeichenkette `<!--[\Ende-->` gefunden und damit das Ende HTML−Erweiterung erkannt wurde. Um nun im restlichen WWW−Dokument nach einer weiteren HTML−Erweiterung suchen zu können, wird wieder in den Startmodus, den Modus 0, gewechselt.

Modus 1:

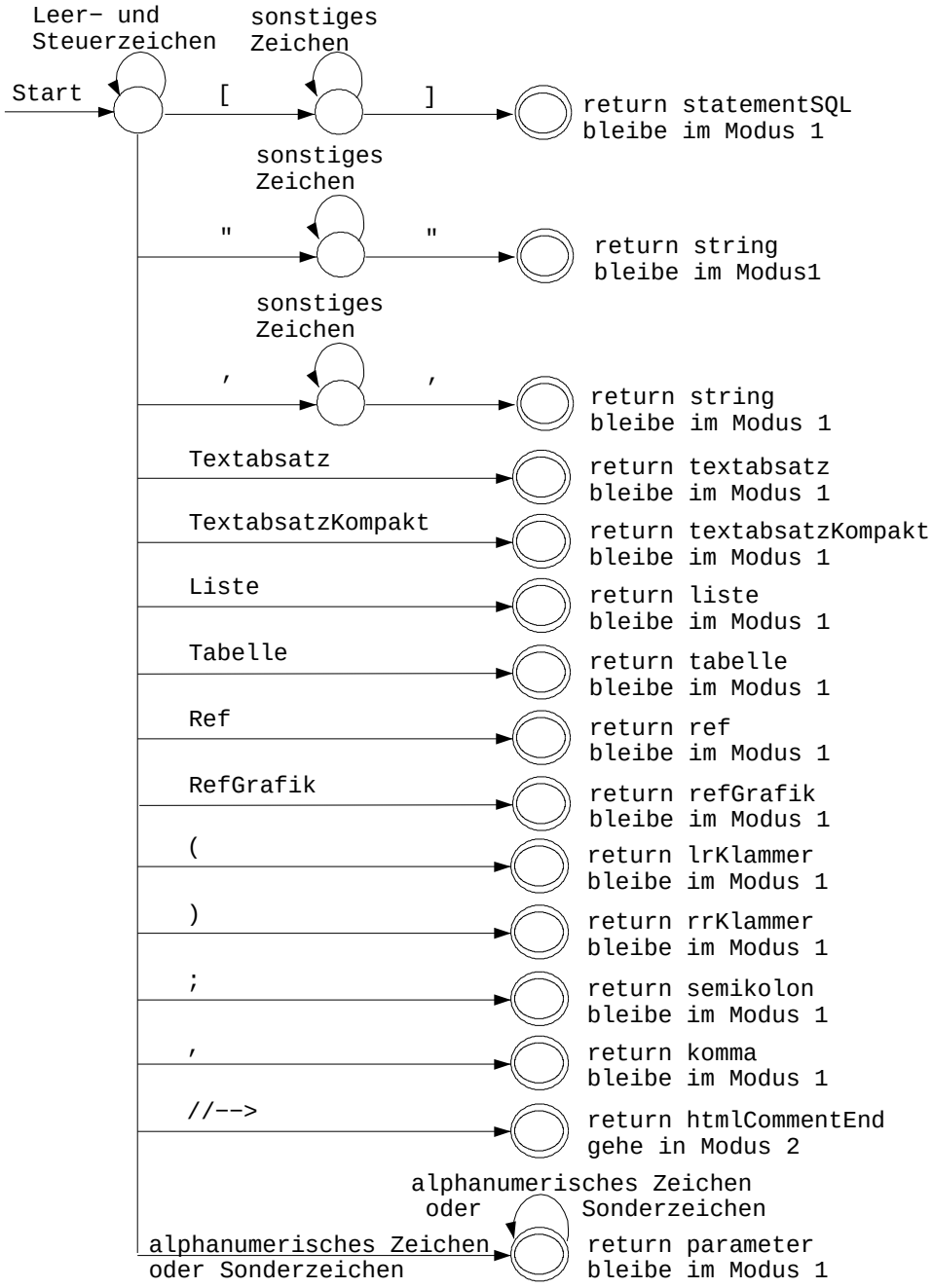

Abbildung 9: Das Übergangsdiagramm für Modus 1

Modus 2:

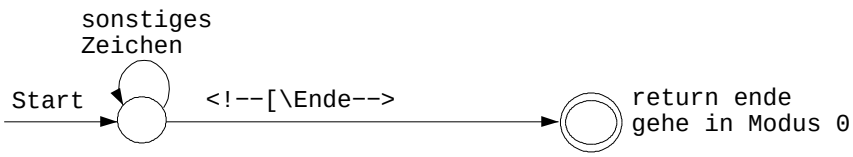

Abbildung 10: Das Übergangsdiagramm für Modus 2

4.2.3 Die attributierte Grammatik für die HTML−Erweiterung

Eine kontextfreie Grammatik beschreibt lediglich die Syntax einer Sprache. Der Parser ist da−her nur in der Lage zu prüfen, ob der Quelltext syntaktisch korrekt ist. Bei Syntaxtfehlern kann er entsprechende Fehlermeldungen ausgeben. Um bei einem Parsevorgang zusätzlich Aktionen ausführen zu können, kann die kontextfreie Grammatik zu einer *attributierten Grammatik* [Plö 94] erweitert werden, indem man sie um

Attribute,

Attributregeln und

Kontextbedingungen

ergänzt.

Dabei beschreiben Attribute Eigenschaften von Symbolen der Grammatik. Attributregeln ge−ben an, wie die Attribute innerhalb einer Ableitungsregel berechnet werden sollen. Die Kon−textbedingungen spezifizieren, welche Bedingungen die Attribute erfüllen müssen, damit das Programm semantisch korrekt ist. Attributgrammatiken sind zwar ein mächtiges Beschrei−bungsmittel, doch verbergen sie viele Implementierungsdetails, da bei ihnen nicht direkt die Stellen in der Grammatik angegeben werden, an denen die Attributregeln (Aktionen) ausge−führt werden sollen. Dies ermöglicht ein Übersetzungsschemata.

Ein *Übersetzungsschemata* [Aho 88] besteht aus

- einer kontextfreien Grammatik,
- Attributen für Grammatiksymbole und
- auf der rechten Seite der Ableitungsregeln zwischen Klammern { } eingeschlossene se−mantische Aktionen. Im Gegensatz zu allgemeinen attributierten Grammatiken wird hier die Auswertungsreihenfolge der Regeln explizit angegeben.

Für die einfachere spätere Umsetzung wird deshalb in dieser Arbeit das Übersetzungsschema−ta verwendet. Ein Übersetzungsschemata ist zwar auch eine Art attributierte Grammatik, doch ist dies lediglich für einfache Grammatiken geeignet, bei denen die Attributauswertung in ei−nem einzigen Durchlauf erfolgen kann. Solche Grammatiken heißen L−attributierte Gramma−tiken [Aho 88]. Die Grammatik für die HTML−Erweiterung erfüllt diese Bedingung.

Im folgenden werden die Attribute und das dazugehörige Übersetzungsschemata der HTML−Erweiterung beschrieben. Die Syntax der semantischen Aktionen wird dabei an die Program−miersprache Java angelehnt. Auf eine Fehlerbehandlung wurde verzichtet.

Attribute der *Terminalsymbole*:

`parameter.name`
: erkennt die Parameter, die in den Darstellungsformen, den Referenzen und Grafikreferen−zen angegeben werden.

`statementSQL.anfrage`
: enthält die erkannte SQL−Anfrage in eckige Klammern eingeschlossen.

`string.wert`
: enthält den String in Anführungszeichen (".." oder ’..’) eingeschlossen.

`zeichenBeliebig.wert`
: enthält ein beliebiges Zeichen.

Attribute der *Nichtterminalsymbole*:

`Attribut.name`
enthält den Attributnamen einer Relation zur Anordnung in der Darstellungsform.

`Eigenschaften.wert`
enthält die Eigenschaften zur Formatierung einer Grafik als Zeichenkette.

`Gestaltung.type`
enthält den Typ der Darstellungsform.

`Url.name`
enthält den Namen einer URL.

Zur Beschreibung der semantischen Aktionen in dem Übersetzungsschemata werden folgende Objekte benötigt:

`Tupelzeile tz;`
ist eine Datenstruktur zur Abspeicherung der Zeilenstruktur für die Datenbankinhalte, wie sie vom Benutzer in einer Darstellungsform angegeben wurde.

`FileWriter fr;`
ermöglicht die Ausgabe des aktualisierten WWW−Dokuments in eine Datei.

`Statement stmt;`
ist für das Senden der SQL−Anweisungen an das DBMS zuständig.

`ResultSet rs;`
repräsentiert die Tupelmenge, die sich als Ergebnis einer SQL−Anfrage qualifiziert hat und verwaltet einen Cursor, der mit der Methode `next` zur nächsten Tupelzeile bewegt werden kann.

Die Konstante `NEWLINE` steht für einen Zeilenumbruch, die Escape−Sequenzen `\t` für einen horizontalen Tabulator und `\"` für ein doppeltes Anführungszeichen.

```
HTMLErweiterung =
    {zeichenBeliebig         {fr.write(zeichenBeliebig.wert)}                    1.
    }
    {
    start                    {fr.write("<!−−[\Start"); tz.initListe()}           2.
    Gestaltung
    statementSQL             {fr.write ("    " + statementSQL.anfrage + NEWLINE);
                                                                                 3.
                             rs=stmt.executeQuery(statementSQL.anfrage.substring 4.
                             (1,statementSQL.anfrage.lenght()−1))
                             //substring entfernt Klammern [..] aus statementSQL
                             tz.initResultSet(rs)                                5.
                             //Uebergabe des SQL−Ergebnisses an tz}
    htmlCommentEnd           {fr.write("//−−>" + NEWLINE)}                       6.
    Ende                     {
                             if (Gestaltung.type == TEXTABSATZKOMPAKT) {         7.
                               fr.write("<P>" + NEWLINE)
                             } //endif
                             while (rs.next()) {                                 8.
                               tz.writeTupel(); fr.write(NEWLINE);
                             } //endwhile
                             if (Gestaltung.type == TEXTABSATZKOMPAKT) {         9.
                               fr.write("</P>" + NEWLINE);
                             } //endif
                             fr.write("<!−−[\Ende−−>" + NEWLINE);               10.
                             }
```

```
    {zeichenBeliebig          {fr.write(zeichenBeliebig.wert)}                  11.
    }
    }.
```

1. Jedes erkannte Zeichen vor der HTML−Erweiterung wird in die aktualisierte Datei übernommen.
2. Schreiben des Startbefehls in die Datei und Initialisieren der Tupelzeile, um die Zeilenstruktur einer eventuell schon vorher erkannten HTML−Erweiterung zu löschen.
3. Schreiben der SQL−Anfrage in die Datei.
4. Senden der SQL−Anfrage an das DBMS und speichern der Ergebnismenge in einem `ResultSet`−Objekt.
5. Übergabe der SQL−Ergebnismenge an das `Tupelzeile`−Objekt für die spätere Ausgabe der Ergebnismenge.
6. Schreiben des Kommentarendes in die Datei
7. Falls es sich bei der Darstellungsform um einen `TextabsatzKompakt` handelt, wird vor dem eigentlichen Datenbankinhalt der `<P>`−Befehl geschrieben.
8. Die Ergebnismenge wird mit der `next`−Methode tupelweise durchlaufen und die Methode `writeTupel` gibt das aktuelle Tupel in HTML formatiert aus.
9. Bei der Darstellungsform `TextabsatzKompakt` wird der Datenbankinhalt durch den `</P>`−Befehl abgeschlossen.
10. Schreiben des Endebefehls der HTML−Erweiterung
11. Jedes nach der HTML−Erweiterung erkannte Zeichen wird in die aktualisierte Datei übernommen bis die Datei vollständig gelesen ist oder eine weitere HTML−Erweiterung erkannt wird.

```
Gestaltung =
     (textabsatz             {Gestaltung.type = TEXTABSATZ;                      1.
                             fr.write(" Textabsatz "); tz.addString("<P>")}
    | textabsatzKompakt      {Gestaltung.type = TEXTABSATZKOMPAKT;
                             fr.write(" TextabsatzKompakt ")}
    | liste                  {Gestaltung.type = LISTE;
                             fr.write(" Liste "); tz.addString("<LI>")}
    ) lrKlammer              {fr.write("(")}
     Attributliste
    rrKlammer                {fr.write(")" + NEWLINE); //Newline für SQL−Anfrage
                             switch (Gestaltung.type) {                          2.
                               case TEXTABSATZ: tz.addString("</P>"); break
                               case TEXTABSATZKOMPAKT: tz.addString("<BR>");break
                               case LISTE: tz.addString("</LI>");break
                             default: break;
                             } //endswitch }
    | tabelle                {Gestaltung.type = TABELLE;                         3.
                             fr.write(" Tabelle ");
                             tz.addString("<TR>" + NEWLINE + "\t<TD>")}
    lrKlammer                {fr.write("(")}
    Attributliste
    { semikolon              {fr.write("; "); tz.addString("</TD>" +
                             NEWLINE + "\t<TD>")}
    Attributliste }
    rrKlammer.               {fr.write(")" + NEWLINE); //Newline für SQL−Anfrage
                             tz.addString("</TD>" + NEWLINE + "</TR>")}
```

1. Zuweisung des erkannten Gestaltungstyps für die späterere Abfrage in der Ableitungsregel

`HTMLErweiterung`. Der Aufruf der erkannten Darstellungsformen wird in die neue Datei und der entsprechende HTML−Befehl zur Formatierung in das `Tupelzeile`−Objekt übernommen.

2. Nach dem Erkennen der schließenden Klammer wird für jede Darstellungsform der entsprechende HTML−Befehl zur Formatierung in das `Tupelzeile`−Objekt übernommen.

3. Setzen des Gestaltungstyps auf `Tabelle`, übernehmen des Tabellenaufrufs in die neue Datei und schreiben der benötigten HTML−Befehle in das `Tupelzeile`−Objekt.

```
Attributliste =
    ( [string               {fr.write(string.wert + " ");                        1.
                            tz.addString(string.wert.substring(1,string.wert.    2.
                            length()−1))
                            //substring entfernt die Anführungszeichen ".."}
    ] (Attribut             {fr.write(Attribut.name);                            3.
                            tz.addAttribut(Attribut.name)}                       4.
    | Referenz
    | Grafireferenz
    )
    [string                 {fr.write(" " + string.wert);
                            tz.addString(string.wert.substring(1,string.wert.
                            length()−1))
                            //substring entfernt die Anführungszeichen ".."}
    ] )
    { komma                 {fr.write(", ")}
    [string                 {fr.write(string.wert + " ");
                            tz.addString(string.wert.substring(1,string.wert.
                            length()−1))
                            //substring entfernt die Anführungszeichen ".."}
    ]
    (Attribut               {fr.write(Attribut.name);
                            tz.addAttribut(Attribut.name)}
    | Referenz
    | Grafikreferenz
    )
    [string                 {fr.write(" " + string.wert);
                            tz.addString(string.wert.substring(1,string.wert.
                            length()−1))
                            //substring entfernt die Anführungszeichen ".."}
    ] } .
```

1. Übernahme der erkannten Zeichenkette in die neue Datei.

2. Schreiben des Strings ohne Anführungszeichen in das `Tupelzeile`−Objekt.

3. Der von der Ableitungsregel `Attribut` übergebene Attributwert `Attribut.name` wird in die neue Datei übernommen.

4. Schreiben des Attributnamens in das `Tupelzeile`−Objekt. Dort werden die Attributwerte von der Ergebnismenge ausgelesen.

```
Referenz =
    ref                     {fr.write("Ref"); tz.addString("<A HREF=\"")}        1.
    lrKlammer               {fr.write("(")}
    [string                 {fr.write(string.wert + " ");
                            tz.addString(string.wert.substring(1,string.wert.
                            length()−1))
                            //substring entfernt die Anführungszeichen ".."}
    ] Url                   {fr.write(Url.name); tz.addString(Url.name)}         2.
    [string                 {fr.write(" " + string.wert);
                            tz.addString(string.wert.substring(1,string.wert.
                            length()−1))
```

```
                          //substring entfernt die Anführungszeichen ".."}
    ] semikolon           {fr.write("; "); tz.addString("\">")}
    [string               {fr.write(string.wert + " ");
                          tz.addString(string.wert.substring(1,string.wert.
                          length()-1))
                          //substring entfernt die Anführungszeichen ".."}
    ]
    (Attribut             {fr.write(Attribut.name);
                          tz.addAttribut(Attribut.name)}
    | Grafikreferenz
    )
    [string               {fr.write(" " + string.wert);
                          tz.addString(string.wert.substring(1,string.wert.
                          length()-1))
                          //substring entfernt die Anführungszeichen ".."}
    ]
    { komma               {fr.write(", ")}
    [string               {fr.write(string.wert);
                          tz.addString(string.wert.substring(1,string.wert.
                          length()-1))
                          //substring entfernt die Anführungszeichen ".."}
    ]
    (Attribut             {fr.write(Attribut.name);
                          tz.addAttribut(Attribut.name)}
    | Grafikreferenz
    )
    [string               {fr.write(" " + string.wert);
                          tz.addString(string.wert.substring(1,string.wert.
                          length()-1))
                          //substring entfernt die Anführungszeichen ".."}
    ] }
    rrKlammer .           {fr.write(")"); tz.addString("</A>")}
```

1. Der Referenzaufruf `Ref` wird in die neue Datei und der Anfang des Referenzbefehls in das `Tupelzeile`−Objekt hinzugefügt.
2. Der von der Ableitungsregel `Url` übergebene Attributwerte `Url.name` wird hier in die neue Datei und in das `Tupelzeile`−Objekt geschrieben.

```
Grafikreferenz =
    refGrafik             {fr.write("RefGrafik");                          1.
                          tz.addString("<IMG SRC=\"")}
    lrKlammer             {fr.write("(")}
    [string               {fr.write(string.wert + " ");
                          tz.addString(string.wert.substring(1,string.wert.
                          length()-1))
                          //substring entfernt die Anführungszeichen ".."}
    ] Url                 {fr.write(Url.name);
                          tz.addAttribut(Url.name)}
    [string               {fr.write(" " + string.wert);
                          tz.addString(string.wert.substring(1,string.wert.
                          length()-1))
                          //substring entfernt die Anführungszeichen ".."}
    ] semikolon           {fr.write("; "); tz.addString("\" ")}
    Eigenschaften         {fr.write(Eigenschaften.wert);                   2.
                          tz.addString(" " + Eigenschaften.wert)}
    rrKlammer .           {fr.write(")"); tz.addString(">")}
```

1. Der Referenzaufruf `RefGrafik` wird in die neue Datei und der Anfang des Grafikrefe−renzbefehls in das `Tupelzeile`−Objekts hinzugefügt.
2. Der von der Ableitungsregel `Eigenschaften` übergebene Attributwert `Eigenschaf−ten.wert` wird hier in die neue Datei und in das `Tupelzeile`−Objekt geschrieben.

```
Attribut = parameter .          {Attribut.name = parameter.name}
Url = parameter .               {Url.name = parameter.name}
Eigenschaften = string .        {Eigenschaften.wert = string.wert}
```

In den Ableitungsregeln Attribut, Url und Eigenschaften werden die auf der rechten Seite vom Scanner erkannten Werte der Terminalsymbole an die Attribute der Nichtterminalsymbole zur späteren Verarbeitung zugewiesen.

Die Umsetzung einer attributierten Grammatik in einen Parser ist eine Routine−Tätigkeit, die von einem Programm übernommen werden kann. Im nächsten Kapitel wird dazu der Parser−generator JavaCC vorgestellt.

5 Die Programmierumgebung für die Implementierung

Die Implementierung erfolgte in der Programmiersprache Java, der Datenbankzugriff mit JDBC. Für die Umsetzung der in Kapitel 4 vorgestellten attributierten Grammatik samt Scanner wurde der Parsergenerator JavaCC verwendet. Deshalb sollen hier die für diese Arbeit wichtigen Grundlagen von JDBC und JavaCC dargestellt werden. Auf die Darstellung von Implementierungsdetails wurde verzichtet, da die vorgestellte Grammatik die Hauptfunktionalität des Parsers darstellt und mit ihren semantischen Aktionen unmittelbar in die Syntax von JavaCC übertragen werden kann.

5.1 Die Datenbankschnittstelle JDBC

JDBC (Java Database Connectivity) [Ham 98] ist eine Java−Programmierschnittstelle (API), mit der auf relationale Datenbanken mittels SQL von Java aus zugegriffen werden kann. Es besteht aus einer Menge von Java−Klassen und Schnittstellen und stellt somit dem Entwickler eine objektorientierte Sichtweise auf relationale Datenbanken dar.

Durch die Verwendung von JDBC ist es möglich, SQL−Anweisungen an nahezu jede relationale Datenbank zu senden, wobei auch gleichzeitig mehrere verschiedene Datenbanken angesprochen werden können. In Kombination mit Java garantiert JDBC eine hohe Portabilität und Plattformunabhängigkeit.

JDBC dehnt die Möglichkeiten von Java auf Datenbanken aus. So ist es z. B. mit in WWW−Dokumenten eingebundenen Servlets (vgl. Abschnitt 2.2.2) und dem JDBC−API möglich, Datenbankinhalte in das World Wide Web zu stellen oder WWW−Dokumente mit einem Applet zu veröffentlichen, das einen Zugriff auf eine entfernte Datenbank erlaubt.

JDBC ist eine Schnittstelle auf »niedriger Ebene«. So müssen die SQL−Anweisungen an die Java−Methoden als Zeichenketten übergeben werden und können nicht direkt in die Programmiersprache des Anwendungsprogramms eingebettet werden. Eine Schnittstelle »höherer Ebene« wäre Embedded SQL, in der SQL−Anweisungen direkt mit Java−Code und Java−Variablen gemischt verwendet werden können. In JDBC kann man zwei Arten von Schnittstellen unterscheiden:

- Eine Programmierschnittstelle (API), über die mit SQL herstellerunabhängig auf Datenbanksysteme zugegriffen werden kann.
- Eine Treiberschnittstelle, die bestimmte Anforderungen festlegt, die Hersteller von Treibern für ihre Datenbanksysteme erfüllen müssen.

In dieser Arbeit wird auf die Treiberschnittstelle nicht weiter eingegangen.

Im folgenden werden die wichtigsten Schnittstellen und Klassen vorgestellt, die für das Stellen von SQL−Anfragen und deren Ergebnisverarbeitung notwendig sind.

Das Laden der Datenbanktreiber erfolgt in der Regel durch folgenden Aufruf:

```
Class.forName("my.sql.Driver");
```

Dabei wird die Treiberklasse `my.sql.Driver` geladen und automatisch eine Instanz der Treiberklasse erzeugt. Diese wird in der Treiberliste der Klasse `DriverManager` nach dem Laden automatisch registriert. Die Klasse `DriverManager` ist die Verwaltungsebene von JDBC, die die registrierten Treiber verwaltet und sich um den Verbindungsaufbau zwischen Datenbank und zugehörigem Treiber kümmert. Dafür verwendet sie die Methoden der Klasse `Driver`. Zusätzlich besteht die Möglichkeit mehrere Datenbanktreiber zu unterschiedlichen

Datenbanken gleichzeitig zu laden.

Sobald `Driver`−Klassen registriert und geladen sind, kann eine Verbindung zu einer Datenbank hergestellt werden. Dazu wird die Methode `DriverManager.getConnection` mit einem URL eventuell zusammen mit Authentifizierungsinformationen wie Login−Name und Paßwort aufgerufen. Die Klasse DriverManager versucht einen Treiber zu lokalisieren, der mit der durch den URL repräsentierten Datenbank eine Verbindung herstellen kann und liefert nach erfolgreichem Verbindungsaufbau ein `Connection`−Objekt zurück.

Jedes Objekt der Klasse `Connection` repräsentiert eine Verbindung der Anwendung zu einer bestimmten Datenbank. Dabei können aus einer einzelnen Anwendung heraus gleichzeitig mehrere Verbindungen zu einer einzelnen Datenbank oder zu verschiedenen Datenbanken bestehen. Aufgebaute Verbindungen werden dazu benutzt, SQL−Anweisungen an eine bestimmte Datenbank zu senden. JDBC beschränkt dabei in keiner Weise die Arten von SQL−Anweisungen, die an die Datenbank geschickt werden können, was es ermöglicht, sowohl datenbankspezifische als auch Nicht−SQL−Anweisungen zu senden. Bei dieser hohen Flexibilität muß der Benutzer allerdings sicherstellen, daß die gesendeten Anweisungen auch von der Datenbank verarbeitet werden können.

Um SQL−Anweisungen an eine Datenbank senden zu können, ist die Erzeugung eines `Statement`−Objekts notwendig. Dieses wird über ein aktives `Connection`−Objekt mit der zugehörigen Methode `createStatement` erzeugt und dient der Ausführung von einfachen SQL−Anweisungen ohne Parameter. Mit der `executeQuery`−Methode werden an dieses die SQL−Anfragen als String übergeben. Zusätzlich gibt es zwei weitere Arten von `Statement`−Objekten. Das `PreparedStatement`, Unterklasse von `Statement`, dient zur Ausführung von vorübersetzten SQL−Anweisungen und wird verwendet, wenn Anweisungen mehrmals hintereinander ausgeführt werden sollen. Dabei können vor jeder Ausführung Parameterwerte neu festgelegt werden, die Struktur der Anweisung bleibt dabei erhalten. Mit `CallableStatement`, Unterklasse von `PreparedStatement`, können in einer Datenbank gespeicherte Prozeduren aufgerufen werden. Diese werden nach Aufruf von einem Java−Clienten lokal innerhalb des DBMS ausgeführt. Nach Beendigung erhält der Client die Ergebnisdaten zurück. Gespeicherte Prozeduren bilden eine Sequenz von SQL−Anweisungen, die eine logische Einheit bilden und bestimmte, in der Regel komplexe Aufgaben durchführen.

Ein `ResultSet`−Objekt repräsentiert die Tupelmenge, die sich als Ergebnis einer SQL−Anfrage qualifiziert hat und verwaltet einen Cursor, der auf das aktuelle Tupel der Ergebnisrelation zeigt. Dieser wird mit jedem Aufruf der Methode `next` zum nächsten Tupel bewegt. Mit den `getXXX`−Methoden können einzelne Attributwerte des aktuellen Tupels ausgelesen werden. Dabei werden die SQL−Datentypen in den für die `getXXX`−Methode entsprechenden Java−Datentyp umgewandelt. So erhält man z. B. mit `getString` jeden SQL−Datentyp als String zurückgeliefert.

Den gesamten Ablauf der Verarbeitung einer SQL−Anfrage verdeutlicht der Aufrufgraph in Abbildung 11 [Ran 97].

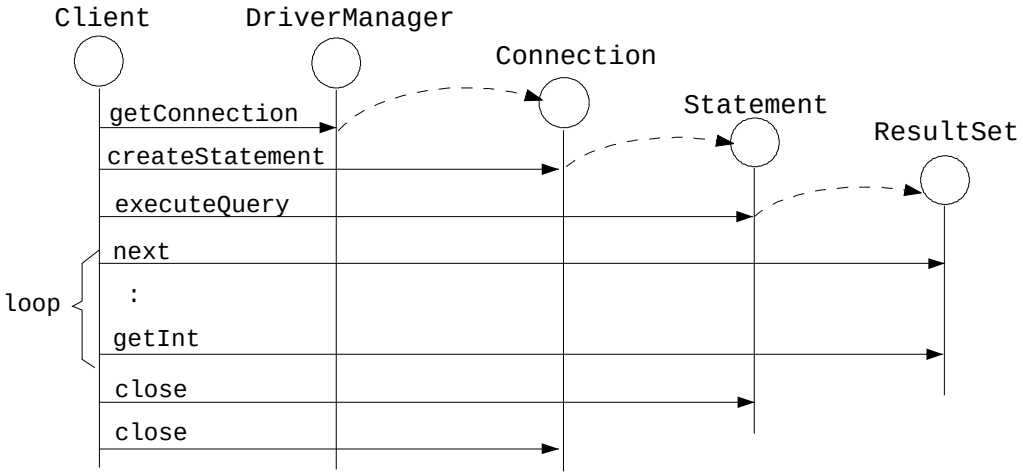

Abbildung 11: Aufrufgraph bei der Verarbeitung einer SQL−Anfrage

Damit eine Java−Anwendung eine Datenbank eindeutig identifizieren kann, ist beim Verbindungsaufbau die Angabe eines JDBC−URL notwendig. Der Aufbau ähnelt dem eines Uniform Resource Locator (URL) und sieht wie folgt aus:

```
jdbc:<subprotocol>:<domain name>
```

Soll z. B. eine Verbindung zu einer DB2−Datenbank der Firma IBM aufgebaut werden, so heißt das Subprotokoll `db2`. Befindet sich diese auf dem Rechner `pandora.informatik.uni−stuttgart.de` mit der Portnummer `6666` und heißt `MyDatabase`, so lautet das vollständige JDBC−URL:

```
jdbc:db2://pandora.informatik.uni−stuttgart.de:6666/MyDatabase
```

5.2 Der Parsergenerator JavaCC

Die Umsetzungen attributierter Grammatiken in einen Parser stellen eine Routine−Tätigkeit dar, die automatisiert, d. h. einem Programm überlassen werden können. Solche Programme werden *Parsergeneratoren* oder *Compiler−Compiler* genannt. Im Hinblick auf Erweiterbarkeit und Vermeidung von Programmierfehlern wurde in dieser Arbeit darauf verzichtet, einen Parser für die in Kapitel 4 dargestellte Grammatik von Grund auf neu zu implementieren. So wurde ein in der Java−Welt bereits existierendes Programmierwerkzeug dafür verwendet.

Die Firma Sun Microsystems stellt einen top−down Parsergenerator mit dem Namen *JavaCC* (Java Compiler Compiler) [Jav] zur Verfügung, der aus einer attributierten Grammatik einen Parser als Java−Programm erzeugt. Die an JavaCC übergebenen Grammatiken müssen LL(k) [Aho 88] sein. Dabei bedeutet das erste »L« in LL(k), daß der Eingabestrom von links nach rechts gelesen wird; das zweite »L«, daß eine Linksableitung erzeugt wird und das k, daß in jedem Schritt des Parse−Prozesses k Tokens vorausgeschaut wird (Lookahead), um entscheiden zu können, welche Aktion durchzuführen ist.

Ein Parser kann unter Verwendung von JavaCC auf folgende Weise konstruiert werden. Zuerst wird eine Datei, nennen wir sie Beispiel.jj, vorbereitet, die eine JavaCC−Spezifikation des Parsers enthält. Diese wird durch den Aufruf

```
javacc Beispiel.jj
```

in die Datei Beispiel.java und weitere Dateien, die den Parser und Scanner implementieren, übersetzt. Die Übersetzung der so erzeugten Java−Programmdateien erfolgt durch den Aufruf

des Java−Compilers:

```
javac *.java
```

Der Aufruf des nun fertigen Parsers erfolgt wie ein normaler Java−Programmaufruf:

```
java Beispiel
```

Die Spezifikation der Grammatik in einer JavaCC−Datei besteht aus drei Teilen. Im ersten Abschnitt kann man verschiedene Optionen festlegen, die globale Eigenschaften des Parsers beschreiben. Mit der `static`−Option läßt sich z. B. festlegen, ob alle Variablen und Methoden des Parsers mit dem Attribut `static` versehen werden sollen. Standardmäßig ist `static=true` gesetzt. Dies hat zur Folge, daß alle Methoden des Parsers nicht an die Existenz eines konkreten Objekts gebunden sind, sondern Klassenmethoden sind. Der Parser läßt sich in diesem Fall lediglich einmal verwenden. Soll es möglich sein, vom Parser verschiedene Objekte in unterschiedlichen Klassen anzulegen, so muß man `static=false` setzen.

Der zweite Abschnitt bildet die Übersetzungseinheit, eingeschlossen in `PARSER_BEGIN(name)` und `PARSER_END(name)`. Die einzige Bedingung in dieser Übersetzungseinheit ist, daß diese eine Klasse mit Namen `name` definiert. Sonst kann sie beliebigen Java−Code enthalten. Soll der Parser als Hauptprogramm aufgerufen werden, so muß die Übersetzungseinheit die Main−Methode mit angelegtem Parserobjekt und Aufruf des Startsymbols der Grammatik enthalten. Soll die Instanzierung des Parsers in einer anderen Klasse stattfinden, so enthält die Übersetzungseinheit z. B. die Konstruktoren des Parsers.

Der dritte Abschnitt untergliedert sich in zwei Bereiche, der lexikalischen Spezifikation gefolgt von der Liste der Ableitungsregeln.

In der lexikalischen Spezifikation werden die Tokens definiert. Dabei können unterschiedliche Arten von Tokens definiert werden. Die zwei wichtigsten sind SKIP und TOKEN.

Beispiel:

```
SKIP:
{
  " "
| "\t"
| "\n"
| "\r"
}
```

Überliest alle Leerzeichen, Tabulatoren und Zeilenumbrüche.

```
TOKEN:
{
  <ID: ["a"−"z","A"−"Z","_"] (["a"−"z","A"−"Z","_","0"−"9"])*>
| <NUM: (["0"−"9"])+>
}
```

Erkennt die lexikalischen Symbole, Bezeichner und Zahlen, und liefert die entsprechenden Tokenwerte `ID` und `NUM` an den Parser zurück.

Zusätzlich gibt es die Möglichkeit, Tokens für bestimmte Zustände zu definieren und anzugeben, in welchen Folgezustand nach deren Erkennung gewechselt werden soll. Dabei befindet sich der Scanner in genau einem dieser Zustände und erkennt genau die für diesen Zustand definierten Token.

In Anschluß an die definierten Tokens werden die Ableitungsregeln definiert. Jede Regel besteht aus einer Ableitungsregel und den semantischen Aktionen. Eine Ableitungsregel

```
<linke Seite> := <alt1> | <alt2> | ... | <altn>.
```

wird in JavaCC folgendermaßen

```
<Rückgabewert> <linke Seite>() :
{Deklarationsteil für Variablen}
{
  <alt1>
   {semantische Aktion 1}
| <alt2>
   {semantische Aktion 2}
| ...
| <altn>
   {semantische Aktion n}
}
```

geschrieben. Zusätzlich zu der hier vorgestellten Auswahl existieren in JavaCC noch weitere der EBNF−Syntax ähnliche Strukturen, um die gegebenen Ableitungsregeln umzusetzen. Jedes Nichtterminal wird hier als eine Java−Methode implementiert. Eine semantische Aktion in JavaCC ist eine Folge von Java−Anweisungen. Attributwerte werden vom Scanner zurückgeliefert und müssen zur Weiterverarbeitung in Variablen zwischengespeichert werden. Die Übergabe von Attributwerten an andere Methoden erfolgt über Rückgabewerte.

Folgendes Beispiel soll die Umsetzung einer Grammatik in die oben dargestellte Spezifikation einer JavaCC−Datei verdeutlichen. Die Beispielgrammatik beschreibt einen Ausdruck mit dem Startsymbol `Expression`, den Terminalsymbolen `id` und `num` und den folgenden Ableitungsregeln:

```
Expression = Term {"+" Term} .
Term       = Faktor {"*" Faktor} .
Faktor     = id | num | "(" Expression ")" .
```

Die beiden regulären Ausdrücke, die der Scanner erkennen und an den Parser zurückliefern soll, lauten:

```
id: <ID: ["a"−"z","A"−"Z","_"] (["a"−"z","A"−"Z","_","0"−"9"])*>
num: <NUM: (["0"−"9"])+>
```

Die Grammatik erkennt Ausdrücke aus Bezeichnern und Zahlen, die addiert, multipliziert und eingeklammert werden können. Durch die Verteilung der Operatoren Addition, Multiplikation und Klammerung auf jeweils nur eine Ableitungsregel, werden die aus der Mathematik bekannten Prioritäten realisiert. Dabei hat eine Klammerung die höchste, die Multiplikation die zweithöchste und die Addition die niedrigste Priorität. Die Umsetzung in JavaCC sieht wie folgt aus.

```
/*ErsterAbschnitt: Festlegen von Optionen*/
Options {
  STATIC=true
}
/*Zweiter Abschnitt: Übersetzungseinheit*/
PARSER_BEGIN(Expression)
public class Expression {
  public static void main(String args[]) throws ParseException {
    //Anlegen eines Parserobjekts
    Expression parser = new Expression(System.in);
    parser.Expression(); //Aufruf der Startprodukiton
  }
}
PARSER_END(Expression)

/*Dritter Abschitt: Bereich der Lexikalischen Spezifikation*/
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
}

TOKEN :
{
  <ID: ["a"-"z","A"-"Z","_"] (["a"-"z","A"-"Z","_","0"-"9"])*>
|
  <NUM: (["0"-"9"])+>
}

/*Dritter Abschnitt: Liste der Ableitungsregeln*/
void Expression() :
{}
{
  Term()
  ( "+" Term()
  )*
} //End Expression

void Term() :
{}
{
  Factor()
  ( "*" Factor()
  )*
} //End Term
```

```
void Factor() :
{}
{
  <ID>
|
  <NUM>
|
  "(" Expression() ")"
} //End Faktor
```

6 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es, einen Generator zu entwickeln und zu implementieren, der eine automatische Aktualisierung von in WWW−Dokumenten integrierten Datenbankinhalten ermöglicht. Um Datenbankinhalte in WWW−Dokumente einbinden zu können, war der Entwurf einer HTML−Erweiterung notwendig. Sie ermöglicht es dem Benutzer, an beliebiger Stelle im Dokument die Datenbankinformationen mittels einer SQL−Anfrage aus der Datenbank abzufragen und in der gewünschter Darstellungsform zu formatieren. Die HTML−Erweiterung enthält die für den Generator notwendigen Angaben für den Aktualisierungsvorgang.

Zusätzlich wurden zwei bereits verbreitete Lösungsmöglichkeiten, dynamische WWW−Dokumente mit eingebundenen Datenbankinhalten zu erzeugen, vorgestellt und deren Vor− und Nachteile erläutert und verglichen. Die erste Möglichkeit ist das Common Gateway Interface (CGI), die zweite sind Servlets. Der entscheidende Nachteil von CGI war für diese Arbeit die schlechte Performance der CGI−Programme, da diese einen erhöhten Speicher− und Verwaltungsbedarf an das Server−Betriebssystem stellen. Servlets verbessern zwar diesen Nachteil, doch ist bei jedem Aufruf eines WWW−Dokuments eine Kommunikation zwischen WWW−Server und Datenbank notwendig. Das Ziel des Generators war unter anderem, diese Nachteile zu beheben.

Der Vergleich der beiden bestehenden Lösungsmöglichkeiten mit dem Konzept des Generators ergab, daß der Generator den Kommunikationsaufwand zwischen WWW−Server und Datenbank deutlich senkt. Zusätzlich entfällt der Aufwand für den Server, bei jedem Aufruf die Programme (CGI oder Servlet) zum Generieren der Datenbankinhalte auszuführen und deren Ergebnisse zu verarbeiten.

Ein WWW−Dokument mit der entwickelten HTML−Erweiterung ist ein statisches WWW−Dokument und hat somit keine Performancenachteile im Vergleich zu dynamischen Ansätzen. Zusätzlich stellt die HTML−Erweiterung eine benutzer− und änderungsfreudliche Alternative zu CGI− und Servlet−Programmen dar. Allerdings ist die Aktualität der WWW−Dokumentinhalte davon abhängig, wann ein Generatorlauf durchgeführt wird. Diesen Nachteil gibt es bei CGI− und Servlet−Programmen nicht, da die Datenbankinformationen bei jedem Aufruf neu generiert werden.

Bei der Entwicklung des Generators wurde auf Wiederverwendbarkeit Wert gelegt. So wurde die eigentliche Funktionalität des Generators in den Parser ausgelagert. Der Parser wurde im Rahmen dieser Arbeit zunächst in einen kommandozeilenorientierten Generator integriert, der die zu aktualisierenden Dateinamen der WWW−Dokumente an das integrierte Parser−Objekt übergibt. Für das Erkennen der HTML−Erweiterung und die Aktualisierung der darin enthaltenen Datenbankinhalte ist dann der Parser zuständig.

Die Schnittstelle und die Methoden der Parser−Klasse wurden so entwickelt, daß sie nicht nur wie hier in einem Kommandozeilenprogramm integriert werden können, sondern auch in beliebige Programme wie Editoren und Entwurfswerkzeuge für WWW−Dokumente und Datenbanken. So wäre ein Editor für das Editieren der Informationen in einer Datenbank denkbar, der nach Datenbankänderungen den Parser zur Aktualisierung von WWW−Dokumenten aufruft.

Weitere gewünschte Darstellungsformen bzw. solche, die sich aus einer neueren HTML−Version ergeben, lassen sich in die HTML−Erweiterung integrieren, indem die Grammatik für die HTML−Erweiterung im Parser angepaßt wird.

Das Konzept eines Generators ist darauf ausgerichtet, schon vorher feststehende Datenbankinhalte in WWW−Dokumente zu integrieren. Werden interaktive Dokumente benötigt, wie dies bei der Eingabe von Suchbegriffen in Formularen der Fall ist, oder sollen individuelle Informationen für ein persönliches Benutzerprofil erstellt werden, so kann der Generator nicht verwendet werden.

Literaturverzeichnis

[Aho 88] Alfred V. Aho, R. Sethi, J. D. Ullmann: *Compilerbau*. Teil 1 und 2 Addision−Wesley 1988

[Cli 98] Paul Clip: *Servlets: CGI the Java Way*. BYTE, Edition May 1998

[Gre 98] Bernd Greiner: *Kommunikations− und Präsentationssystem für universitäre Umgebungen*: Diplomarbeit Nr. 1645, 1998

[Ham 98] Graham Hamilton, Rick Cattell, Maydene Fisher: *JDBC™ : Datenbankzugriff mit Java™* . Bonn: Addison−Wesley−Longman, 1998

[Jav] JavaCC Documentation: Java CompilerCompiler Version 0.8pre1
`http://www.suntest.com/JavaCC/`

[Mün 98] Stefan Münz: *HTML−Dateien selbst erstellen.* SELFHTML: Version 7.0 vom 27.04.1998,
`http://www.teamone.de/selfaktuell/`

[Pom 93] Gustav Pomberger: *Software Engeneering*: München; Wien: Hanser, 1993

[Ran 97] Ralf Rantzau: *Realisierung eines Link−Managers für das World Wide Web.* Studienarbeit Nr. 1605, 1997

[Rec 97] Peter Rechenberg, Gustav Pomberger: *Informatik−Handbuch,* Wien: Hanser, 1997

[RFC #] Requests for Comments (RFC), wobei # für die RFC−Nummer steht,
`http://www.internic.net/rfc/rfc#.txt`

[Thi 98] Uwe Thiemann: *Das Netzwerk ist der Computer.* BYTE Deutschland, Ausgabe Mai 1998

[W3] W3−Konsortium:
`http://www.w3.org/Markup/`

Erklärung:

Ich versichere, daß ich diese Arbeit selbständig verfaßt habe und nur die angegeben Quellen und Hilfsmittel verwendet habe.

....

Datum (Martin Notz)