Universität Karlsruhe (TH)
Fakultät für Informatik
Zentrum für Multimedia

XGen: Generierung von XML mit Hilfe von Datenbanken

Zur Wissensgenerierung in NICA

Studienarbeit

von

Frederik Thiele

betreut von

Prof. Dr. rer. nat. Peter Deussen
Dipl.-Inform. Sven Claußen

Juni 2002

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Aufgrund hoher Studienanfängerzahlen an der Universität Karlsruhe (TH) ist der Bedarf an Informationen zum Informatikstudium stark gestiegen. Um einer Überlastung der Berater innerhalb und außerhalb der Fakultät entgegenzuwirken, wurde in früheren Arbeiten [HV01] der StudiBot[1] entwickelt, der zur Aufgabe hat, übliche Fragen der aktuellen und zukünftigen Studenten der Fakultät für Informatik zu beantworten. Um dies zu realisieren, wurde ein benutzergesteuertes Dialogsystem entwickelt. Der Benutzer richtet natürlichsprachliche Fragen an das System, welches die Fragen beantwortet. Inzwischen ist daraus die Firma Agtive Systems entstanden, die NICA entwickelt. Mit NICA lassen sich intelligente Beratungssysteme (z. B. der StudiBot) erstellen.

Damit das System in der Lage ist, möglichst alle Anfragen zu verarbeiten, müssen die entsprechenden Informationen in einer Wissensbasis vorliegen. Diese Wissensbasis wird von Wissensingenieuren erstellt und gepflegt. Mit der Zeit wächst das Wissen des Systems, so dass sich die Arbeit der Wissensaktualisierung minimieren sollte. Allerdings kann Wissen veralten (z. B. die Prüfungsordnung), so dass eine ständige Pflege erforderlich ist. Dies stellt einen erheblichen Aufwand dar, da darüberhinaus das Wissen jedes Mal auf Korrektheit zu überprüfen ist.

Auf der anderen Seite existieren schon verschiedene Informationssysteme wie z. B. das [i3v], in dem viele Themen zum Studium in Datenbanken zusammengefasst sind und aktuell gehalten werden. Deshalb besteht die Idee darin, bestimmte Informationsteile der Datenbanken, die besonders häufig

[1] Die Erklärung von Abkürzungen und Fachbegriffen der Studienarbeit befindet sich nochmals im Glossar auf Seite 41

vom Benutzer gefragt werden, in das Wissen des Dialogsystems zu integrieren. Die Idee geht sogar noch weiter: eine Datenbank wird direkt abgefragt, falls eine Frage nicht beantwortet werden kann, sofern eine gewisse Chance besteht, dass sie die richtigen Informationen liefern kann.

Aus diesen Überlegungen heraus ist diese Studienarbeit entstanden: Sie stellt eine Schnittstelle zwischen dem Wissen von NICA und verschiedenen Datenbanken dar.

1.2 Grundidee

Die Wissensbasis von NICA wird auf XML basierend gespeichert. Dieses Wissen ist – vereinfacht dargestellt – wie folgt aufgebaut:

Es existieren Themenkreise (*Topic*), die beliebig ineinander verschachtelt werden können. Weiterhin gibt es ein Themenobjekt (*TopicObject*), welches ein spezielles Objekt dieses Themenkreises darstellt, z. B. könnte einer der Themenkreise „Professoren“ sein. Dieses Topic enthält mehrere TopicObjects, nämlich die einzelnen Professoren mit ihren Daten (Name, Adresse, Telefonnummer, e-Mail etc.). Um nun bei einer bestimmten Frage des Benutzers herauszufinden, welches Objekt angesprochen wurde, existieren sog. *Matches*, die angeben, bei welchen Begriffen sich das Objekt angesprochen fühlen soll. Wenn ein Objekt gefunden wurde, das zu der gerade gestellten Frage passt, wird eine Antwort (*Answer*) ausgegeben, die zu diesem Objekt gehört. Dabei können durchaus mehrere Antworten in dem Objekt vorhanden sein, um den gleichen Sachverhalt auf verschiedene Art und Weise ausdrücken zu können. Ferner kann durch eine Antwort auch auf einen anderen Themenkreis verwiesen werden (*Link2Topic*), falls dieser Themenkreis eventuell auch mit der Frage in Zusammenhang steht oder weiterführende Informationen bietet.

Nach dieser sehr vereinfachten Darstellung der Wissensbasis von NICA betrachten wir nun, wie verschiedene TopicObjects mit Hilfe von Datenbanken erzeugt werden können. Zur Vereinfachung werden nur TopicObjects betrachtet, obwohl es auch denkbar wäre, Topics zu erzeugen. Um neue TopicObjects der Wissensbasis hinzuzufügen, bauen wir in den Baum Generator-Objekte (*XGen*) ein, die eine Datenbank abfragen, beliebige TopicObjects erzeugen und diese in den Baum integrieren. Die Abbildung 1.1 verdeutlicht die Vorgehensweise. Dabei soll ein Präprozessor die XGen ausführen. Bevor der StudiBot gestartet wird, werden die XGens aufgelöst, so dass statt ihnen vollständige TopicObjects im Baum stehen. Die XGens selbst sollen alle Informationen enthalten, die zur Generierung der TopicObjects nötig sind, also die Datenbankinformationen und Informationen darüber, wie die TopicOb-

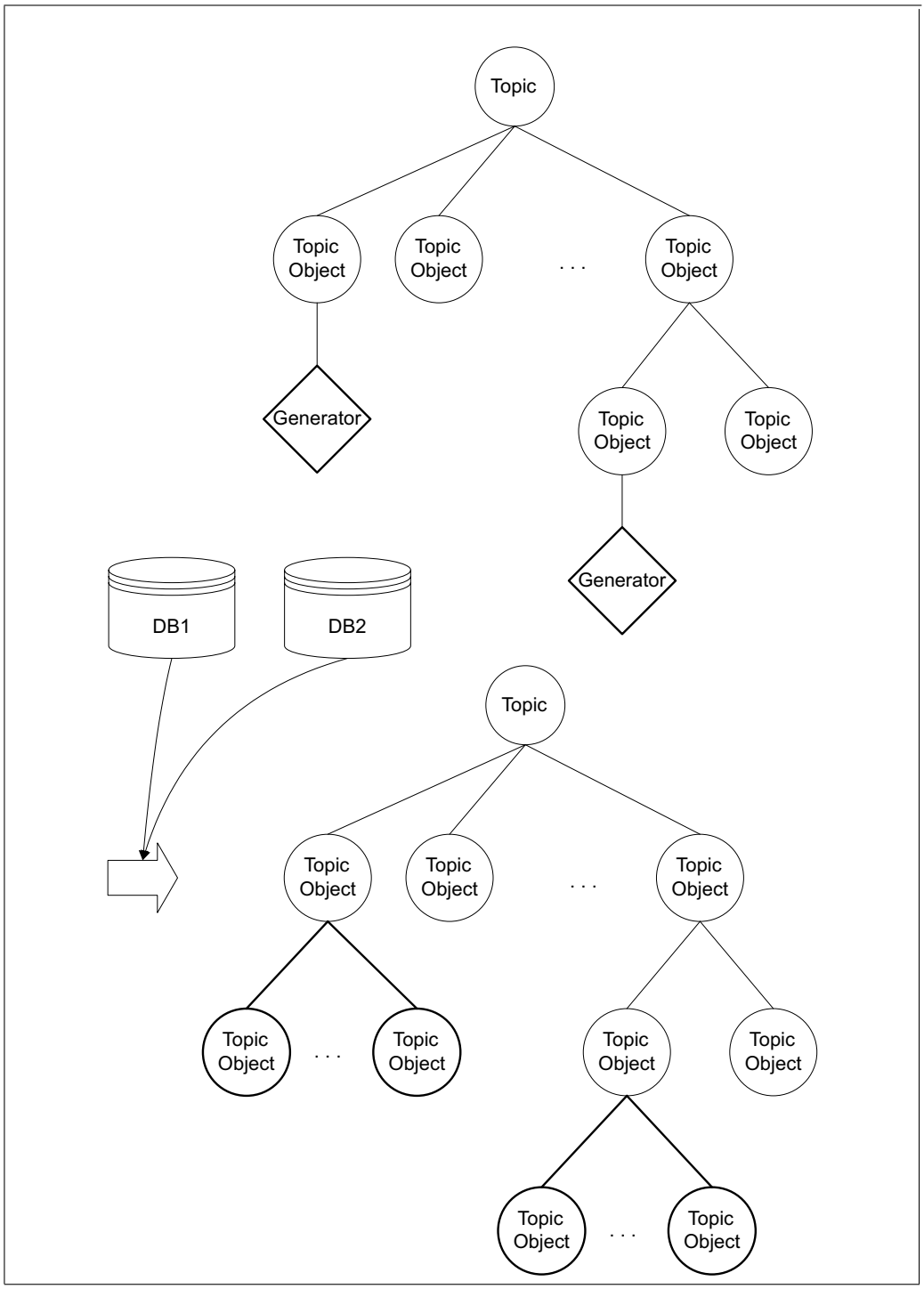

Abbildung 1.1: Modellierung der Generierung von Wissen

jects aussehen sollen. Da sich NICA zur Laufzeit einen eigenen Objektbaum aus den XML-Daten aufbaut, ergibt es zunächst nur Sinn, die XGens vorher auszuführen. Somit ist auch die Laufzeit zur Erzeugung von Teilen der Wissensbasis aus einer Datenbank unkritisch, da der Präprozessor z. B. immer nachts gestartet wird, um das neueste Wissen aus den Datenbanken zu generieren. Natürlich kann durch häufigeres Starten die Aktualität des Wissens erhöht werden, wenn dies gewünscht wird. Beim StudiBot reicht es meist aus, dass die Informationen tagesaktuell zur Verfügung stehen. Die Erneuerung der Daten kann über Nacht erfolgen.

1.3 Aufbau der Arbeit

In dieser Studienarbeit wird ein für Agtive Systems entwickeltes System zur Generierung von XML-Daten mit Hilfe von relationalen Datenbanken vorgestellt. Zunächst wird im zweiten Kapitel gezeigt, wie Daten einer Datenbank in XML eingebettet werden können. Die Struktur der XML-Daten spielt dabei die entscheidende Rolle.

Das Konzept und der Aufbau der XGens wird im nächsten Abschnitt beschrieben. Sie stellen eine Umsetzung der Ausführungen aus dem zweiten Kapitel dar. Ferner werden verschiedene Probleme bei der Einbettung der Daten aus der Datenbank in das XML-basierte Wissen geschildert.

Im vierten Abschnitt wird erläutert, wie die XGens in das NICA-Wissen eingebettet werden und welche Universalität sie aufweisen. Obwohl das Konzept durch einen Präprozessor realisiert wird, achten wir darauf, dass die XGens auch zur Laufzeit ausführbar und somit auch für andere Anwendungssituationen geeignet sind.

Wie die Vorgehensweise zur Erstellung eines XGens ist, wird im Anhang detailliert erklärt. Dies soll allgemein als Anleitung dienen, wenn der Wissensingenieur Teile der Informationen aus Datenbanken in das Wissen eines NICA-Systems integrieren möchte.

Bei der Entwicklung der XGens sind noch weitere Ideen entstanden, die allerdings wegen ihrer Unübersichtlichkeit bzw. wegen technischer Probleme nicht realisiert wurden. Diese Vorschläge werden auch im Anhang dargestellt.

Kapitel 2

Einbettung von Informationen aus Datenbanken in XML

Obwohl die Sicht der Daten von relationalen Datenbanken und von XML zunächst nichts miteinander zu tun haben, ist es trotzdem sehr interessant, eine Verbindung zwischen ihnen herzustellen, da durch das Internet XML immer mehr an Bedeutung gewinnt. Deswegen sind die Datenbankhersteller sehr daran interessiert, XML mit Datenbanken in Einklang zu bekommen. Vor allem spielt der Aspekt der Einbindung von Informationen aus Datenbanken eine Rolle. Um dies zu bewerkstelligen, gibt es eine Reihe von Mitteln und Wegen, auf die im Folgenden eingegangen wird.

2.1 Stand der Technik

In der Vergangenheit wurden viele Ideen entwickelt, mit denen man XML und Datenbanken miteinander verknüpfen kann. Allerdings ist das Problem nach wie vor, dass es kaum einheitliche Standards gibt, wie ein Austausch von Informationen zwischen Datenbanken und XML geschehen soll. Hier werden meist sehr individuelle Lösungen entwickelt, die mehr oder weniger einfach auf die eigenen Bedürfnisse angepasst werden können. Ein häufig gebrauchter Weg ist z. B., dass aus einer Datenbank XML-Daten generiert werden und die Struktur der Datenbank auf XML abgebildet wird.

Prinzipiell existieren drei verschiedene Wege, um XML-Dokumente aus Datenbanken zu generieren:

- Abbilden des vollständigen Datenbankinhalts
- Abbilden von Anfrageergebnissen
- Einsatz individueller Transformationsvorschriften

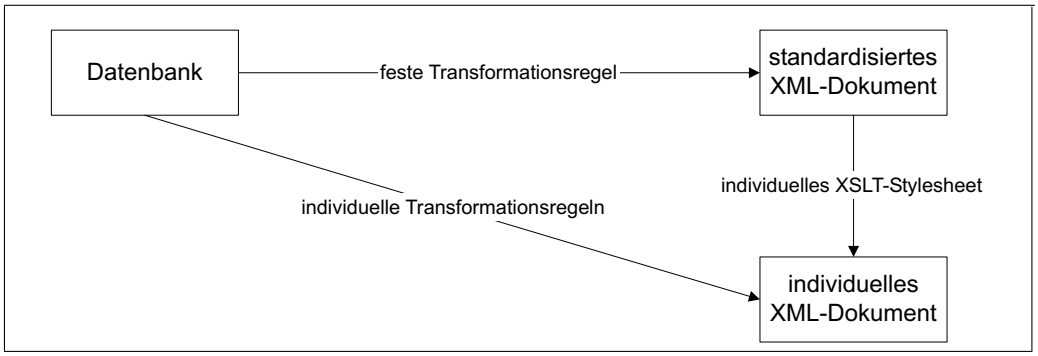

Abbildung 2.1: Transformationsmöglichkeiten

Die ersten beiden Punkte bilden die gesamte Datenbank oder Teile von ihr (einzelne Relationen) auf eine XML-Struktur ab, die dem Datenbankschema entspricht. Dadurch ist nur eine Umformatierung der Daten nötig. In vielen aktuellen Datenbanken gibt es bereits Schnittstellen, die aus der Datenbasis XML-Dokumente erzeugen. Allerdings besteht keinerlei Flexibilität bei den Ausgabeformaten.

Bei der dritten Variante kann der Benutzer selbst festlegen, wie das Ausgabeformat aussehen soll. Hier wird ein zweistufiger Prozess durchlaufen, in dem zunächst die Daten aus der Datenbank abgefragt und danach in ein spezielles DTD-konformes Schema transformiert werden (siehe Abbildung 2.1). Meist wird hier zunächst aus der Datenbank ein standardisiertes XML-Dokument erzeugt und danach mittels einer geeigneten XSLT-Transformation in das gewünschte Ausgabeformat umgeformt. Eine andere Möglichkeit besteht darin, beide Schritte ohne Zwischenergebnis durchzuführen. Vor allem ist die Variante sehr interessant, die Daten in bestehende XML-Dokumente einzubetten anstatt ein neues Dokument zu erzeugen. In den nächsten Abschnitten stellen wir zwei Arten der individuellen Transformation vor.

2.2 XQuery

Eine funktionale XML-Anfragesprache ist XQuery, das vom [W3C] spezifiziert wird und sich von Quilt ableitet. Wie in der Entstehungsgeschichte zu sehen ist (Abbildung 2.2), vereint Quilt wieder mehrere Konstrukte aus verschiedenen Anfragesprachen (wie XQL, XML-QL, Lorel und Yatl).

XQuery verwendet das verbreitete XPath, um die Navigation in hierarchisch strukturierten Dokumenten durchzuführen, und ist nach dem FLWR-Ausdruck (ausgesprochen: „flower“) aufgebaut. Dies erlaubt die Auswahl

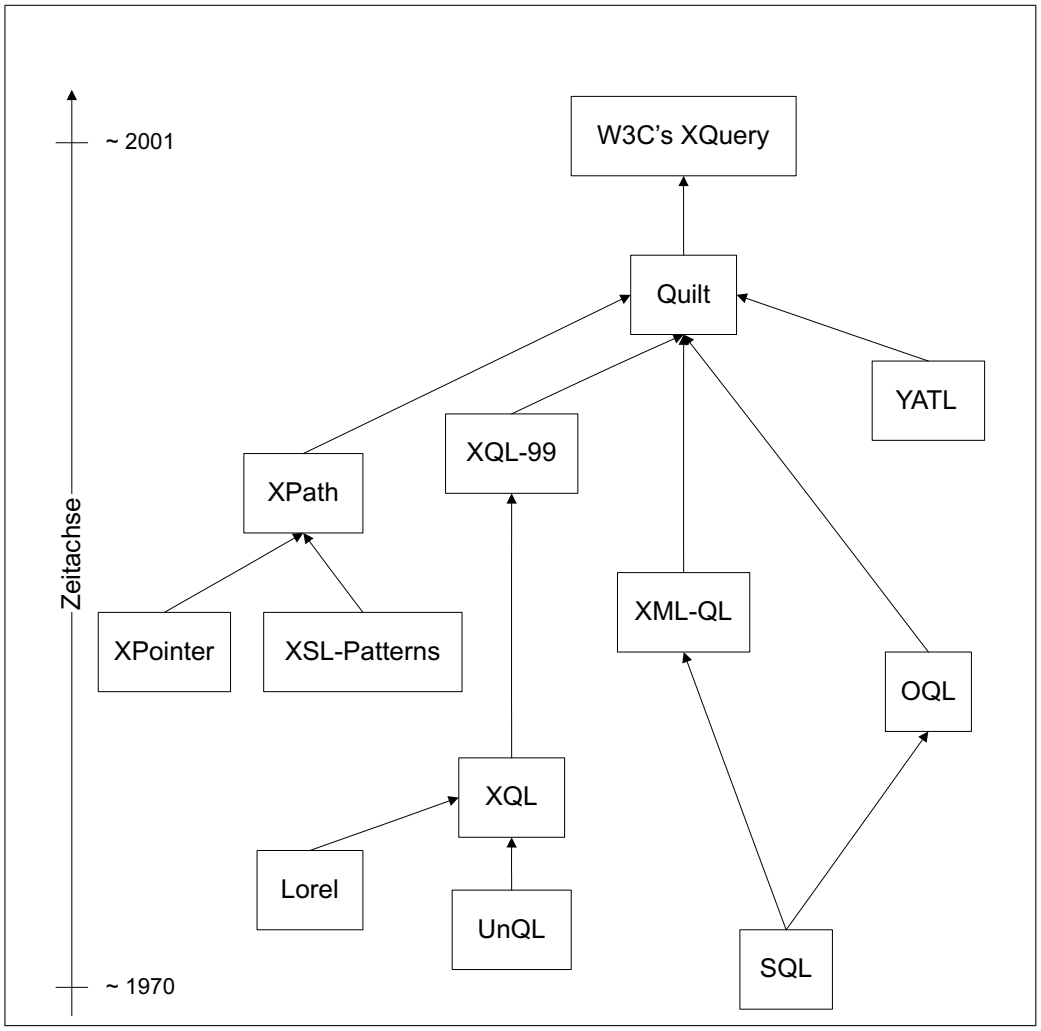

Abbildung 2.2: Geschichte von XQuery

(`FOR`), Bindung (`LET`) und Ausgabe (`RETURN`) der Daten, die zusätzlich bestimmten Bedingungen (`WHERE`) folgen. Allerdings wird zur Zeit noch intensiv an XQuery gearbeitet, so dass viele – auch wichtige – Entscheidungen noch offen stehen. Daher ist die Durchsetzungsfähigkeit noch fraglich, obwohl von der Idee her XQuery durchaus gute Chancen hat, an Bedeutung zu gewinnen [XQ].

2.3 Java als Schnittstelle

Java bietet sowohl für Datenbanken als auch für XML standardisierte Schnittstellen an, mit deren Hilfe man die Einbettung der Daten durchführen kann.

Dies entspricht dem direkten Weg von der Datenbank zum individuellen XML-Dokument (siehe Abbildung 2.1).

Für die Datenbankanbindung bietet sich JDBC an, mit dem ein SQL-Ausdruck an eine relationale Datenbank gesendet und die Ergebnisrelation in Java ausgewertet werden kann. Daraufhin lassen sich die Tupel in XML-Dokumente einbetten. Dafür gibt es zwei Möglichkeiten. Die erste ist die Manipulation eines DOM-Baumes (Document Object Model). Um in Java einen DOM-Baum aus einem bestehenden XML-Dokument aufzubauen, kann JAXB oder auch JDOM verwendet werden, wobei zur Zeit weder die eine noch die andere Schnittstelle endgültig fertig gestellt sind [JXB, JDO].

Die zweite Möglichkeit besteht darin, dass ein SAX-kompatibler Parser ein XML-Dokument einliest. Um die Informationen der Datenbank in das Dokument einzubauen, werden die *Handler* des Parsers derart manipuliert, dass sie das XML-Dokument in veränderter Form hinaus schreiben [SAX].

Die Vorgehensweise mit Java als Schnittstelle zwischen Datenbanken und XML wird auch für die XGens verwendet. Für den prinzipiellen Ablauf siehe Abbildung 3.1 in Kapitel 3.

Kapitel 3

Der XGen

Wie im vorigen Kapitel bereits gesehen, ist die Funktionsweise der Generierung von XML im Prinzip sehr einfach: Man holt die geforderten Informationen aus der Datenbank, bereitet sie auf und schreibt XML wieder zurück. Dabei treten zunächst zwei Probleme auf:

- Wie gelangt man an die Daten einer beliebigen Datenbank, d. h. welches Vorwissen benötigt man über die Datenbank.
- Wie sollen die Daten in XML modelliert werden, d. h. welche Struktur legt man zugrunde.

Abstrakt betrachtet bedeutet das, dass wir ein Datenbankschema und ein XML-Schema (z. B. in Form einer DTD) bekommen und Daten des einen in die des anderen überführen. Allerdings sollte als Ergebnis dieser Studienarbeit ein lauffähiges System (nämlich die XGens) zur Verfügung stehen, so dass sich diese abstrakte Problematik als zu komplex herausstellt. Daher haben wir uns auf den Fall beschränkt, dass die Informationen, die aus einer Datenbank abgefragt werden sollen, vorher definiert und durch einen geeigneten SQL-Ausdruck abgerufen werden. Dadurch braucht man hauptsächlich nur das XML-Schema zu betrachten.

Für die XGens beschränken wir uns auf relationale Datenbanken, da die meisten bestehenden Datenbanken so aufgebaut sind. Außerdem werden Standards für objektorientierte Datenbankanfragen kaum unterstützt, obwohl eine Umsetzung in Java denkbar wäre.

Weiterhin ist der in diesem Kapitel erwähnte Benutzer der Wissensingenieur, der das XML-Wissen erstellt und pflegt. Der Wissensingenieur ist auch derjenige, der XGens erstellt und ändert.

Um eine flüssige Einarbeitung in das Thema zu gewährleisten, sind in diesem Kapitel die Unterschiede zwischen den bereits vorhandenen Anwen-

dungen und den durch diese Studienarbeit entstandenen Werkzeugen nicht klar getrennt. Daher werden hier kurz die Voraussetzungen dargestellt:

Die Definition der Struktur von XML ist frei. Sie wird mit Hilfe von DTDs spezifiziert. Dazu gibt es vom W3C verschiedene Empfehlungen, wie z. B. für verschlüsselte Elemente. Um mit Java XML verarbeiten zu können, bedarf es einiger zusätzlicher Werkzeuge. Dazu existieren Parser, die nach dem SAX-Standard entworfen werden. Ein weiteres Werkzeug ist JAXB, welches z. Z. noch in der Entwicklung ist. Weiterhin wird eine Unterstützung von JDBC vorausgesetzt, welches eine Schnittstelle zwischen Java und einer Datenbank zur Verfügung stellt. Schließlich benötigt man die Java-Pakete zur Verschlüsselung (JCE) und einen geeigneten Provider für JCE.

Sämtliche Überlegungen werden anhand des XML-basierten Wissens des StudiBots durchgeführt (vor allem für den Prototyp), lassen sich aber auch auf andere XML-Strukturen übertragen.

3.1 Aufbau

Ein XGen ist in XML spezifiziert und wie folgt aufgebaut: Er besteht aus Variablen, die Informationen über die Datenbank liefern, und einem Prototyp. Die Informationen über die Datenbank werden benötigt, um eine JDBC-Verbindung zu einer Datenbank aufzubauen. Dies sind Treiber, URL, Benutzername und Passwort. Weiterhin gehört auch ein SQL-Ausdruck dazu, der angibt, welche Daten abgefragt werden. Dies reicht bereits, um die Datenbank abzufragen und das geforderte Ergebnis zu erhalten. Nun definieren wir noch eine Abbildung zwischen lokalen Variablen (*LocalName*) und Datenbankvariablen (*LabelInSQL*). Hier soll der Benutzer angeben, welche Variablen aus der Datenbank betrachtet werden können und unter welchem Namen sie im Prototyp zu benutzen sind. Dies ist aus Gründen der Transparenz sehr wichtig, da im SQL-Ausdruck stellvertretend auch der *-Operator im SELECT-Ausdruck benutzt werden darf. Zukünftig weiss aber außer dem Ersteller des XGens meist niemand mehr, welche Spaltennamen sich hinter dem *-Operator verbergen. Da ein XGen immer in XML spezifiziert wird, gibt es keine einfache Möglichkeit, nachträglich die Spaltennamen zu erschließen. Somit ist der Ersteller gezwungen, sie anzugeben, so dass später andere Benutzer in der Lage sind, Änderungen vorzunehmen.

Nach den Datenbankinformationen folgt der Prototyp. Bei der Spezifikation eines Prototypen gibt es keine Einschränkungen, solange man keine Erweiterungen hinzufügen möchte. Zum Beispiel lässt sich durch Beachtung bestimmter Regeln eine Fehlerbehandlung für die Informationen aus der Datenbank durchführen. Dies ist allerdings nicht zwingend nötig. Dies wird in

Abschnitt 3.1.2 und 3.4.6 beschrieben. Die DTD eines XGens ist wie folgt definiert:

```
<!--******* XGen ********************************** -->
<!ELEMENT XGen(GenVariables, Prototype)>
<!ELEMENT GenVariables (DBConfig, SQLQuery, VariableDef+)>
<!--Database-->
<!ELEMENT DBConfig (DBDriver, DBURL, DBLogin, DBPassword)>
<!ELEMENT DBDriver (#PCDATA)>
<!ELEMENT DBURL (#PCDATA)>
<!ELEMENT DBLogin (#PCDATA)>
<!ELEMENT DBPassword (EncryptedData)>
<!ENTITY % encrypt.dtd SYSTEM "encrypt.dtd">
%encrypt.dtd;
<!--Data and Variables-->
<!ELEMENT SQLQuery (#PCDATA)>
<!ELEMENT VariableDef (LabelInSQL, LocalName)>
<!ELEMENT LabelInSQL (#PCDATA)>
<!ELEMENT LocalName (#PCDATA)>
<!--Convert data to Objects-->
<!ELEMENT Prototype (prot:Topic | prot:TopicObject)+>
<!ATTLIST Prototype
    xmlns:prot CDATA #REQUIRED
>
```

3.1.1 Datenbankanbindung

Da wir nach Voraussetzung eine relationale Datenbank ansprechen, können wir direkt die vorhandenen Schnittstellen nutzen. Für Java bietet sich JDBC an, sofern für die Datenbank ein entsprechender Treiber vorhanden ist. Anderenfalls besteht meist eine ODBC-Anbindung der Datenbank, so dass man auf die von Sun Microsystems zur Verfügung gestellte JDBC-ODBC-Brücke zurückgreifen kann.

Wenn man eine Verbindung zur Datenbank herstellen möchte, müssen zunächst folgende Fragen geklärt werden:

- Welcher JDBC-Treiber muss benutzt werden?
- Wie heißt die Datenbank, d. h. wie lautet die URL?
- Ist ein Benutzer-Account notwendig?

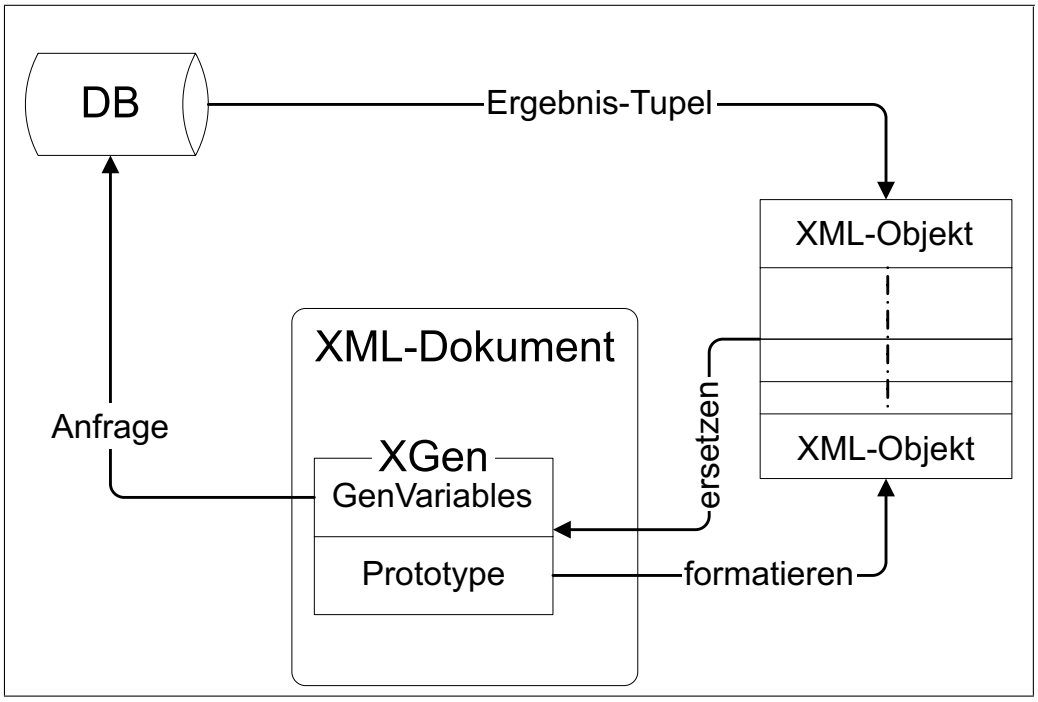

Abbildung 3.1: Aufbau eines XGens und Ablauf beim Generieren

Diese Fragen werden mit dem Datenbankadministrator abgeklärt. Falls eine eigene Datenbank angesprochen werden soll, muss man sich die nötigen Treiber vom Datenbankhersteller besorgen. Über die URL und den Benutzer-Account kann selbst entschieden werden. Bei einigen Datenbanken (z. B. Microsoft Access) lassen sich die Daten auch ohne Benutzernamen und Passwort ansprechen. Dann braucht man nichts in die Tags DBLogin und DBPassword einzutragen. Die Informationen werden im DBConfig-Abschnitt des XGen eingetragen.

Nachdem eine Verbindung zur Datenbank besteht, kann man sich der Hauptaufgabe zuwenden: den Daten. Um an die gewünschten Daten in der Datenbank zu gelangen, wird eine SQL-Anfrage gestartet. Falls eine fremde Datenbank angesprochen werden soll, muss wieder der Datenbankadministrator weiterhelfen, indem er eine Sicht (View) erstellt, auf die man durch den einfachen Ausdruck

```
SELECT * FROM view_name;
```

zugreifen kann. Aber es ist durchaus auch erlaubt, beliebig komplexe SQL-Ausdrücke in XML zu hinterlegen, so dass der Datenbankadministrator nicht unbedingt eine Sicht zu erstellen braucht. Dies erscheint allerdings häufig

sinnvoll zu sein, da er durch Sichten wesentlich einfacher Zugriffsbeschränkungen auszusprechen vermag. Denn es genügt, wenn die aktuellen Informationen aus der Datenbank gelesen werden können.

Nun sind alle Informationen über die Datenbank vorhanden, so dass sich der XGen das *ResultSet* in Java holen kann. Dies wird direkt weiterverarbeitet, indem die zurückgegebenen Attribute eines Tupels in diesem ResultSet auf die lokalen Variablen des XGens abgebildet werden (vgl. DTD in Abschnitt 3.1) und für jedes Tupel ein XML-Objekt erzeugt wird. Durch weitere Kommentare und vollständige Variablendefinition ist es möglich, Änderungen am Prototyp (s. u.) vorzunehmen, ohne sich wieder vollständig mit den Datenbankinformationen beschäftigen zu müssen, falls ein Wissensingenieur den XGen modifizieren möchte.

Hier ein Beispiel ohne verschlüsseltes Passwort-Element und der Variablenabbildung, die hier vollständig dargestellt wird, da der SQL-Ausdruck auf zwei Attribute beschränkt ist:

```
<GenVariables>
    <DBConfig>
        <DBDriver>sun.jdbc.odbc.JdbcOdbcDriver</DBDriver>
        <DBURL>JDBC:ODBC:Prof</DBURL>
        <DBLogin>nica</DBLogin>
        <DBPassword>nica</DBPassword>
    </DBConfig>
    <SQLQuery>SELECT ID, Name FROM Professoren;</SQLQuery>
    <VariableDef>
        <LabelInSQL>ID</LabelInSQL>
        <LocalName>id</LocalName>
    </VariableDef>
    <VariableDef>
        <LabelInSQL>Name</LabelInSQL>
        <LocalName>name</LocalName>
    </VariableDef>
</GenVariables>
```

3.1.2 XML-Prototyp

Neben den Daten für die JDBC-Anfrage werden Informationen über den zu generierenden XML-Text benötigt. Daher wird im XML-Abschnitt *Prototype* des XGens festgelegt, wie die zu erzeugenden XML-Objekte aussehen sollen. Diese Schablone wird in einer DTD spezifiziert. Es sollten dabei zwei Dinge beachtet werden:

1. Der gesamte Prototyp soll im Namensraum *prot* sein.
2. *VarCalls* sollen an den richtigen Stellen zulässig sein.

Der erste Punkt ist durch einen Kniff relativ einfach zu erfüllen, obwohl Namensräume durch DTDs nicht unterstützt werden; darauf gehen wir näher im Abschnitt 4.2 ein.

Um die in Abschnitt 3.1 erwähnten lokalen Variablen benutzen zu können, werden so genannte *VarCalls* im Prototypen eingeführt. Sie werden als Element definiert, wobei der Elementinhalt stets einen Variablennamen bezeichnet. Daher ist beim zweiten Punkt zu beachten, dass Elemente nur in andere Elemente verschachteln werden können, d. h. es ist nicht möglich, Informationen aus der Datenbank in Attribute einzulesen. Um die VarCalls einzubauen, müssen wir uns überlegen, an welchen Stellen wir sie zulassen möchten. Meistens möchten wir auch noch zusätzlichen Text bei dem entsprechenden Element zulassen (z. B. bei Antworten: „Professor `<VarCall>`Name `</VarCall>` hat die Telefonnumer `<VarCall>`Telefon`</VarCall>`“). Deswegen setzen wir in der DTD an den Stellen, an denen `#PCDATA` steht, stattdessen `(#PCDATA | prot:VarCall)*` ein. Hierbei forden wir ein wohlgeformtes XML-Dokument, da ein validierender Parser – wie er für SAX oder JAXB verwendet wird – sonst keine korrekte Arbeitsweise garantiert. Dadurch sind keine Variablenaufrufe in Attributen möglich. Wenn dies tatsächlich benötigt wird, so sollte man sich überlegen, ob es nicht sinnvoller wäre, dieses Attribut zu einem Unterelement zu machen. Um trotzdem mit Attributen weiterarbeiten zu können, wäre es möglich, statt den VarCall-Elementen VarCall-Entitäten einzuführen. Allerdings wirft das eine Reihe von weiteren Problemen auf, die im Anhang B.3 näher behandelt werden.

Dieses Modell mit Elementen gemischten Inhalts wird in der vorliegenden Implementierung nicht angewendet. Das hängt mit den Alternative-Elementen zusammen, da die Behandlung der Text-Elemente sonst sehr aufwändig wäre (siehe Abschnitt 3.4.6). Stattdessen werden *VariableText*-Elemente eingesetzt, die *String*- und *VarCall*-Elemente enthalten dürfen. Dies macht die Arbeit des Wissensingenieurs zunächst etwas komplizierter, trägt aber auch zur Übersichtlichkeit der XGens bei, da schneller erkannt werden kann, wo ein VarCall auftritt.

3.2 SAX vs. DOM

Um XML in Java einzulesen und zu bearbeiten, gibt es prinzipiell zwei Möglichkeiten:

1. XML mit Hilfe eines SAX-Parsers lesen.

2. XML als Objektbaum (DOM) in Java verarbeiten.

SAX definiert einige Schnittstellen zum Parsen von XML-Dokumenten. Dabei wird bei bestimmten Ereignissen (z. B. Element, Attribut oder Entität) eine Methode aufgerufen. Jeder SAX-kompatible Parser muss diese Schnittstellen implementieren. Einige der Parser bieten darüberhinaus noch weitere Funktionen an (z. B. Auslesen des InfoSets). Allgemein sind Parser nur zum Lesen eines Dokumentes gedacht, und nicht dazu, ein XML-Dokument wieder zurückzuschreiben. Daher sind z. B. Kommentare und Processing Instructions nur optional erreichbar, weil sie in der Regel bei einer Ereignissteuerung nicht interessant sind. Das InfoSet bei XML [IS] ist in den heutigen SAX-Parsern (SAX, Version 2.1) überhaupt nicht erreichbar. Es gibt zwar schon Entwürfe, wie so etwas gehandhabt werden könnte, aber sie sind noch nicht standardisiert (siehe Anhang B.2).

Die zweite Möglichkeit besteht darin, dass das komplette XML-Dokument in einen Objektbaum eingelesen wird. Dazu bietet sich JAXB an, welches aus einer DTD Java-Klassen erstellt, um einen Baum aufzubauen. Außerdem stehen schon Funktionen zur Ausgabe bereit, so dass dies im Gegensatz zu SAX-Parsern keine Schwierigkeiten bereitet. Weiterhin kann man sämtliche Navigationsmöglichkeiten in einem Baum ausnutzen. Dafür sind auch schon eine Reihe von Methoden in JAXB eingebaut. Es ist damit möglich, alle Elemente eines Namens in dem Baum als Liste zu erhalten. Das bedeutet, dass es sehr einfach möglich wäre, sämtliche XGens eines XML-Dokumentes zu bekommen, so dass sie nacheinander bearbeitet werden können. Allerdings befindet sich JAXB (oder auch JDOM) zur Zeit noch im Beta-Stadium, so dass dessen Benutzung noch etliche Probleme bereitet [JXB, JDO].

Der wesentliche Unterschied zwischen diesen beiden Möglichkeiten besteht darin, dass auf SAX basierende Parser das XML-Dokument sequentiell durchlaufen, d. h. es muss zur Bearbeitungszeit immer nur das nächste Element verfügbar sein. Danach wird es nicht mehr benötigt. Bei einem XML-Baum dagegen muss der gesamte Inhalt des XML-Dokumentes im Speicher stehen. Falls die Dokumente sehr groß werden, kann es dabei schnell zu Performanzproblemen kommen. Ein weiteres Problem bei einem XML-Baum besteht darin, dass von einer XML-Datei wieder in eine XML-Datei zurückgeschrieben werden soll, so dass man zunächst aus einem XML-Stream einen Baum erzeugen und in ihm die Operationen durchführen müsste, um ihn dann wieder in einen Stream zu verwandeln. Da SAX bereits auf Streams basiert, ersparen wir uns hierbei das Umwandeln. Der einzige Nachteil besteht darin, dass die Ausgabe selbst gesteuert werden muss. Aber da wir bei einer Wissensstruktur, wie sie hier vorliegt, durchaus mit großen XML-Dateien rechnen können, beruht die vorliegende Implementierung auf SAX-Parsern.

SAX	DOM
• ereignisorientiert • einfacher Zugriff • einfach oder gleichartig strukturierte Dokumente • auch geeignet für sehr große XML-Dokumente	• Objektstruktur • Navigation durch Dokumentstruktur • für sehr große XML-Dokumente problematisch • Speicherungsstruktur

Tabelle 3.1: Vergleich zwischen SAX und DOM

3.3 Benutzung von SAX

Um einen SAX-Parser zu verwenden, muss man sich im wesentlichen nur um den entsprechenden Handler kümmern. Der Handler muss von einer Handler-Schnittstelle erben (DefaultHandler, LexicalHandler etc.) und gewisse Methoden implementieren. Die Methoden werden bei bestimmten Ereignissen aufgerufen, so dass daraufhin die gewünschten (implementierten) Aktionen erfolgen. Beispielsweise wird

```
startElement(String uri, String localName,
             String qName, Attributes attr)
```

beim Anfangs-Tag eines Elementes mit den entsprechenden Parametern aufgerufen. Hierbei kann abgefragt werden, um welches Element es sich handelt, welche Attribute dieses Element hat etc. Je nach Handler-Schnittstelle, von der geerbt wird, werden verschiedene Ereignisse bearbeitet. Zum Beispiel ist der LexicalHandler für das Auslesen der Kommentare und Entitäten zuständig. Ausführliche Anleitungen befinden sich unter [SAX]. In den folgenden Abschnitten werden die einzelnen Handler erläutert, die durch diese Studienarbeit entstanden sind.

3.3.1 KnowledgeHandler

Der eigens entworfene KnowledgeHandler liest ein XML-Dokument ein und schreibt es zunächst ohne Veränderung wieder hinaus. Es soll alles erhalten bleiben – sowohl die Kommentare, Processing Instructions und das InfoSet als auch die Entitäten. Da bei dem SAX-Parser keine Funktionen zur Ausgabe existieren, müssen bei jedem Ereignis auch wieder die entsprechenden

XML-Daten in einen OutputStream geschrieben werden. Dies ist auch deswegen notwendig, weil es keine Möglichkeit gibt, durch die XML-Struktur zu navigieren, wie das bei einem DOM-Baum möglich wäre, der auch gleich Methoden zur Ausgabe bereit hält (siehe hierzu auch [TUT], insbesondere dort „Part II: Serial Access with the Simple API for XML (SAX)“).

Weiterhin wird der KnowledgeHandler dazu benutzt, sämtliche XGens aus einem XML-Dokument herauszusuchen. Wenn das Start-Tag eines XGens (`<XGen>`) gefunden wird, schreibt der Handler die folgenden Daten in einen neuen Stream. Nach Ende des XGens (`</XGen>`) lässt sich der Stream, der den kompletten XGen enthält, zur Verarbeitung an den GenHandler weiterreichen. Danach bekommt der KnowledgeHandler einen Stream zurück, der die erzeugten XML-Objekte enthält. Dieser Stream kann nun an den Dokument-Stream angehängt werden. Damit ist der XGen aus dem ursprünglichen XML-Dokument entfernt. Wenn der KnowledgeHandler die Daten einmal sequentiell durchlaufen hat, steht der Ergebnis-Stream zur Verfügung, in dem alle XGens durch ihre generierten XML-Daten ersetzt wurden. Damit ist die Umwandlung fertig. Der gesamte Ablauf ist in Abbildung 3.2 dargestellt.

3.3.2 GenHandler

Der GenHandler ist dafür zuständig, einen XGen-Stream zu parsen. Dabei sollen ein GenKnowledge-Objekt erzeugt, alle Informationen aus dem XGen ausgelesen und in diesem Objekt gespeichert werden. Ein GenKnowledge-Objekt ist ein Java-Objekt zur Verarbeitung der XGens. Bei den Informationen, die ein GenKnowledge-Objekt speichert, handelt es sich um sämtliche Datenbankinformationen, die Abbildung der SQL-Variablen auf die lokalen Variablen und letztendlich den Prototyp selbst, der als Stream behandelt wird. Danach steht ein startbereites GenKnowledge-Objekt zur Verfügung, welches dann den zu generierenden Stream mit Hilfe des ProtHandlers erzeugen kann. Insbesondere ist der GenHandler für die Entschlüsselung des Passwortes verantwortlich (siehe Abschnitt 3.4.4). Natürlich lässt sich der GenHandler auch unabhängig vom KnowledgeHandler aufrufen. Das Aufrufen des GenHandlers übernimmt der Konstruktor von GenKnowledge, so dass danach bereits ein fertiges GenKnowledge-Objekt zur Verfügung steht.

3.3.3 ProtHandler

Bei der dritten Stufe – dem ProtHandler – handelt es sich wieder um einen Handler, der zunächst nur die Daten des zu parsenden Streams zurückschreiben soll. Dabei achtet er zusätzlich darauf, wann ein VarCall-Element beginnt (`<VarCall>`). Ist dies der Fall, wird der Inhalt (= der Name der Variablen)

dieses Elementes ausgelesen und nachgeschaut, welchen Wert die entsprechende Variable hat. Dies wird in dem GenKnowledge-Objekt gespeichert, da vor dem Parsen des Streams mit dem ProtHandler der Datenbankaufruf stattfindet. Dabei werden die benötigten Werte in einer Hashtabelle abgespeichert, damit der Zugriff über die lokalen Variablennamen erfolgen kann. Die Abbildung der SQL-Variablen auf die lokalen Variablen erfolgt bereits beim Auslesen des ResultSets. Der Inhalt wird nun statt des kompletten VarCalls eingesetzt, d. h. aus „`<VarCall>`Vorname`</VarCall>`“ wird z. B. „Vanessa“. Hierbei werden die Objekte (z. B. die TopicObjects) über die Anzahl der Ergebnistupel iteriert, es werden also genauso viele Objekte erzeugt wie es Tupel in der Ergebnisrelation gibt.

Der ProtHandler wird über die Methode *generateStream* aus der Klasse GenKnowledge aufgerufen. Es ist daher auch möglich, eine Instanz der Klasse GenKnowledge zu erzeugen, in der die Datenbankinformationen manuell gesetzt werden. So wird über die Methode

```
OutputStream generateStream(InputStream prot, GenKnowledge gen)
```

aus einem Prototyp direkt ein OutputStream erzeugt. Dies kann vor allem zur Laufzeit eingesetzt werden (siehe Abschnitt 4.3.3).

3.4 Umsetzung

Da die technischen Lösungen bei der Verwendung verschiedener Hilfsmittel in dieser Studienarbeit nicht gleich einsichtig sind, gehen wir hier auf einige Probleme ein und erläutern sie etwas näher.

3.4.1 Handhabung von Streams

Der prinzipielle Umgang mit Streams in Java sollte bekannt sein. Das Problem besteht darin, dass zwischen Input- und OutputStreams hin und her gewechselt wird. Dafür bieten sich Byte-Array-Streams an, da sich aus ihnen ein Byte-Array erzeugen lässt. Die oben beschriebenen Handler benötigen immer einen InputStream und schreiben einen OutputStream heraus. Da aber der XGen, den der KnowledgeHandler liefert, wieder vom GenHandler geparst werden soll, muss der OutputStream in ein InputStream umgewandelt werden. Dazu braucht der Benutzer lediglich das Byte-Array des Streams auszugeben und dem Konstruktor des jeweils anderen Stream-Typs zu übergeben. In anderen Stream-Implementierungen ist dies meist nur über Umwege möglich (z. B. über Dateien).

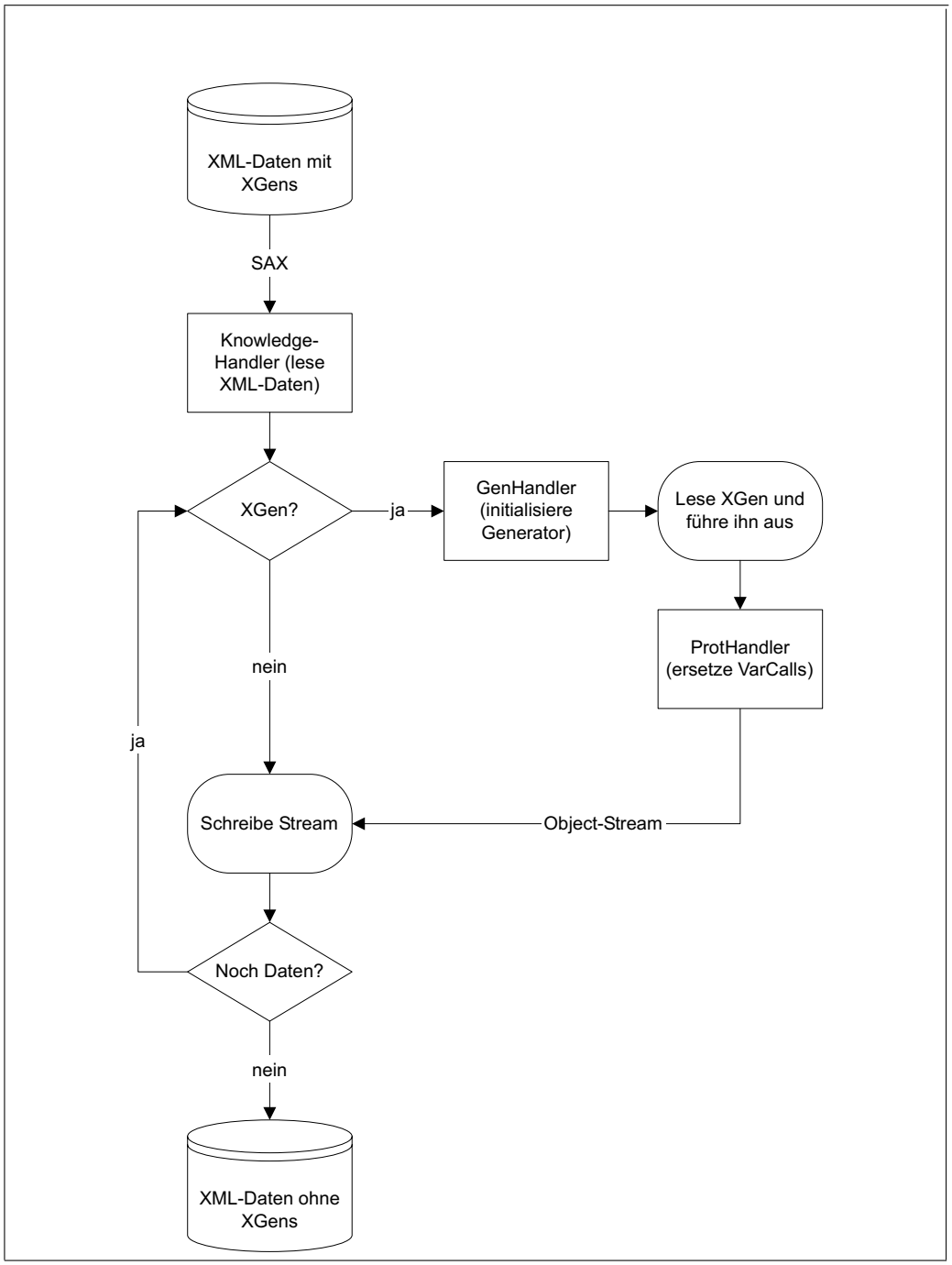

Abbildung 3.2: Durchlauf der Daten durch die Handler beim Generieren

Enitität	Ersetzung	Zeichen
&amp;	&	&
&apos;	&#39;	’
&gt;	&#62;	<
&lt;	<	>
&quot;	&#34;	\

Tabelle 3.2: Definitionen der standardmäßig definierten Entitäten

3.4.2 Kodierung

Die Kodierung eines XML-Dokumentes wird im InfoSet eingetragen. Als Standardkodierung für XML wird UTF-8 verwendet, allerdings ist es möglich, auch andere Kodierungen einzusetzen. Die Parser wandeln unabhängig von der Kodierung den Text korrekt in Java-Strings um. In den Implementierungen dieser Studienarbeit wird immer UTF-8 benutzt. Wenn im InfoSet eine andere Kodierung verwendet wird, lässt sich diese nicht übernehmen, da sich das InfoSet nicht über die SAX-Parser auslesen lässt. Die Handler verwenden stattdessen automatisch UTF-8 und kodieren das XML-Dokument unabhängig von seiner ursprünglichen Kodierung um (siehe auch Anhang B.2).

3.4.3 Entitäten und Sonderzeichen in XML

In XML werden standardmäßig für fünf Zeichen Entitäten definiert, da sie als Sonderzeichen in XML anzusehen sind. In Tabelle 3.2 werden sie mit den zugehörigen Entitäten dargestellt. Wann immer sie im XML-Dokument in einem Text (`PCDATA`) auftreten, müssen sie als Entität zurückgeschrieben werden. Allerdings löst der SAX-Parser automatisch die Entitäten auf, d. h. schon beim Parsen werden sie durch ihre Zeichen ersetzt. Da aber bei jeder Entität ein entsprechendes Ereignis eintritt, kann dies abgefangen werden. Dazu wird das schon ersetzte Zeichen unterdrückt und die Entität eingefügt. Weiterhin werden die aus der Datenbank eingefügten Werte überprüft, ob sie eines dieser fünf Zeichen enthalten. Falls dies der Fall ist, wird auch hier die entsprechende Entität eingesetzt.

3.4.4 Verschlüsselung des Passwortes

Da das XML-Dokument sehr einfach zu lesen ist, lässt sich auch das Passwort für die Datenbankanbindung sofort herausbekommen. Dies lässt sich dadurch verhindern, dass der Inhalt des Elementes DBPassword verschlüsselt darge-

stellt wird. Dabei gehen wir nach den Vorschlägen des W3C zur Verschlüsselung von XML-Daten vor. Es wird ein *EncryptedElement* eingefügt, welches alle nötigen Informationen enthält, um wieder aus dem verschlüsselten Inhalt zum Klartext zu gelangen [Enc]. Dabei sind in dieser Implementierung einige Freiheiten gelassen worden. Der Benutzer muss sich zwischen folgenden Möglichkeiten entscheiden:

- Asymmetrische Verschlüsselung
- Symmetrische Verschlüsselung
- Symmetrische Verschlüsselung unter Mitgabe des asymmetrisch verschlüsselten Schlüssels

Bei allen Möglichkeiten muss ein Schlüssel hinterlegt werden. Dies wird über die Schlüsselverwaltung von Java realisiert [KEY]. Außerdem kann der Benutzer noch das Verfahren der Verschlüsselung wählen, wobei er darauf achten muss, dass der entsprechende JCE-Provider (Java Cryptography Extension [Pro]) diesen Algorithmus auch unterstützt. Hier wird der BouncyCastle-Provider verwendet, der neben RSA als asymmetrischem Verfahren alle gängigen symmetrischen Verfahren (AES, DES, Triple-DES, Rijndael, Blowfish etc.) unterstützt [BCP].

Die DTD für das verschlüsselte Element DBPassword ist wie folgt spezifiziert:

```
<!ELEMENT EncryptedData (EncryptionMethod, ds:KeyInfo,
                         CipherData, EncryptionProperties?)>
<!ATTLIST EncryptedData
    Id CDATA #IMPLIED
    Type CDATA #FIXED "http://www.w3.org
                      /2001/04/xmlenc#Content"
    xmlns CDATA #FIXED "http://www.w3.org/2001/04/xmlenc#"
>
<!ATTLIST EncryptionMethod
    Algorithm CDATA #REQUIRED
>
<!ELEMENT EncryptionMethod (IV?)>
<!ATTLIST EncryptionMethod
    Algorithm CDATA #REQUIRED
>
<!ELEMENT IV (#PCDATA)>
<!ELEMENT ds:KeyInfo (EncryptedKey?, ds:KeyName)>
<!ATTLIST ds:KeyInfo
```

```
    xmlns:ds CDATA #FIXED "http://www.w3.org/2000/09/xmldsig#"
>
<!ELEMENT EncryptedKey (EncryptionMethod, CipherData)>
<!ATTLIST EncryptedKey
    Id CDATA #IMPLIED
    xmlns CDATA #FIXED "http://www.w3.org/2001/04/xmlenc#"
>
<!ELEMENT ds:KeyName (#PCDATA)>
<!ELEMENT CipherData (CipherValue?, CipherReferenceURI?)>
<!ELEMENT CipherValue (#PCDATA)>
<!ELEMENT CipherReferenceURI (#PCDATA)>
<!ELEMENT EncryptionProperties EMPTY>
<!ATTLIST EncryptionProperties
    Type CDATA #IMPLIED
>
```

Es werden längst nicht alle Möglichkeiten ausgeschöpft, die das W3C spezifiziert hat. Auf der einen Seite ist die weitere Funktionalität hier nicht nötig, während auf der anderen Seite dies mit Hilfe von DTDs auch kaum realisierbar ist, da das W3C die EncryptedElements bereits in *XML-Schema* spezifiziert. XML-Schema bietet vor allem mächtigere Konsistenzbedingungen an, so dass sich nicht alles in DTDs übertragen lässt. Aber für die Ansprüche des DBPassword reicht es aus.

Da diese Studienarbeit sich nur nebensächlich mit der kryptographischen Komponente beschäftigt, möchten wir hier auf die umfangreiche Literatur verweisen, die zu diesem Thema verfügbar ist [Bar00, JCE].

3.4.5 Eindeutigkeit der erzeugten Objekte

Falls die erzeugten Objekte eindeutig sein sollen (wie z. B. die TopicObjects), d. h. für jedes Objekt existiert ein Schlüsselelement (z. B. *Name* bei den TopicObjects), dessen Inhalt in dem gesamten Dokument nur einmal auftauchen darf. Wenn durch den XGen mehrere Objekte erzeugt werden und in dem Schlüsselelement keine VarCalls auftreten, so erhalten wir viele nicht eindeutige Objekte. Um dies zu vermeiden, prüft der ProtHandler, ob im Prototyp beim Schlüsselelement ein VarCall auftritt. Ist dies der Fall, wird davon ausgegangen, dass die erzeugten Objekte eindeutig sind. Ansonsten wird automatisch eine laufende Nummer an den Inhalt des Schlüsselelements gehängt, so dass alle Objekte eindeutig sind. Ein möglicherweise auftretendes Problem ist, dass die VarCalls keine eindeutigen Ergebnisse liefern. Um dieses Problem sicher zu umgehen, braucht man entweder gar kein VarCall

anzufügen, so dass eine laufende Nummer hinzugefügt wird, oder es wird der Schlüssel der Relation aus der Datenbank eingefügt. Dies garantiert die Eindeutigkeit der erzeugten Objekte, sofern sich nicht ein bereits existierendes Objekt mit gleichem Namen in der Wissensbasis befindet.

3.4.6 Behandlung von NULL-Werten aus der Datenbank

Da es passieren kann, dass die Datenbank für ein Attribut keinen gültigen Wert enthält, wird nach JDBC-Definition ein NULL-Wert übergeben. Sofern das der Fall ist, wird dieser Wert auf einen Leerstring ("") abgebildet. Das bedeutet, dass bei einem Aufruf der entsprechenden Variablen einfach das VarCall-Element entfernt und nichts dafür eingesetzt wird. Dies kann allerdings zu unerwünschten Effekten führen. So könnte es beispielsweise sein, dass zu einer Person, zu der aus einer Datenbank sämtliche Informationen eingelesen werden sollen, keine Homepage existiert. Dann würde der XGen diese Person trotzdem so behandeln, als hätte sie eine. Als Folge davon würde im Antworttext eines TopicObjects erscheinen: „Klar kenne ich die Homepage von dieser Person! Sie lautet: “. Der Benutzer würde sich wundern, warum keine Internetadresse angegeben wird.

Um diesem Problem zu begegnen, wird jedem Element ein optionales Attribut *skipIfNull* hinzugefügt. In diesem Attribut kann eine Liste von VarCalls angegeben werden. Falls eines dieser VarCalls einen NULL-Wert enthält, wird das gesamte Element mit seinen Unterelementen übersprungen. Damit kann sich der Wissensingenieur dafür entscheiden, einen Teil des Prototyps wegzulassen, falls keine sinnvollen Variablenwerte vorhanden sind.

Der Wissensingenieur hat allerdings noch eine zweite Möglichkeit: Er kann einem Element als erstes Unterelement Alternativen hinzufügen. Diese Alternative-Elemente sind wie folgt spezifiziert:

```
<!ELEMENT prot:Alternative (prot:Alternative?,
                    (prot:VariableText | prot:Text))>
<!ATTLIST prot:Alternative
    useIfNull NMTOKENS #REQUIRED
>
<!-- VariableText ist einfacher Text mit VarCalls -->
<!ELEMENT prot:VariableText (prot:String | prot:VarCall)*>
<!ELEMENT prot:String (#PCDATA)>
<!--    Text
        mit HTML/XML angereichert, wird dem Client gesendet -->
<!ELEMENT prot:Text (prot:Alternative?, prot:VariableText)>
```

```
<!-- VarCall zum Ersetzen der aus der Datenbank
eingelesenen Daten -->
<!ELEMENT prot:VarCall (#PCDATA)>
```

In einem Alternative-Element lässt sich über das Attribut *useIfNull* angeben, welche Variablen auf NULL-Werte zu überprüfen sind. Falls eine dieser Variablen auf NULL gesetzt ist, soll bei der Generierung statt des üblichen Inhaltes diese Alternative verwendet werden. Da der Inhalt einer Alternativen u. U. wieder einen VarCall enthalten kann, sind sie rekursiv anwendbar.

Nach Einführung der *Alternative*-Elemente wurden in unserem Fall auch Elemente mit gemischtem Inhalt (also mit Text *und* Elementen) entfernt. Statt dessen existiert ein Element *VariableText*, das aus verschiedenen Elementen (*String* und *VarCall*) zusammengesetzt ist. Dies erleichtert die Arbeit mit den Alternative-Elementen bei der Implementierung, da zum Zusammensetzen des richtigen Inhaltes verschiedene String-Operationen in Java ausgeführt werden müssen.

Mit diesen beiden Hilfen hat der Wissensingenieur sehr viele Freiheiten. Er kann damit individuell entscheiden, welche Möglichkeit für seine Situation am günstigsten erscheint, wobei zuerst immer die SkipIfNull-Attribute ausgewertet werden, bevor das Alternative-Element behandelt wird. Wenn keines von beiden angegeben wurde, wird der Leerstring eingesetzt.

Kapitel 4

Einbindung der XGens in XML-Dokumente

Nachdem die XGens konzeptuell entworfen und implementiert sind, sollen sie in ein bestehendes System eingebunden werden. Dabei gilt es, einige Dinge zu beachten. Es kommt hierbei etwas auf die spezielle Anwendung an, in die der XGen integriert wird, allerdings treten gewisse Probleme häufiger auf. Diese sollen hier noch einmal zusammengefasst werden, obwohl sie schon in Kapitel 3 genannt wurden.

Hier wird die Einbettung anhand von NICA erläutert. Natürlich lassen sich die XGen auch in andere Systeme einbetten. Dann müssen entsprechende Änderungen an der DTD vorgenommen werden. Dabei können die Mechanismen für die Behandlung von NULL-Werten im Prototyp übernommen werden.

Der Benutzer ist in diesem Kapitel der Entwickler einer NICA-Anwendung, z. B. des StudiBots.

4.1 Einbettung in das XML-basierte Wissen von NICA

Es braucht nur die DTD erweitert werden, um die XGens in NICA zu integrieren, so dass sie zulässig sind. Wenn ein XML-Dokument erstellt wurde, in dem es XGens gibt, wird ein Präprozessor ausgeführt, der sie in die gewünschte Form übersetzt, so dass schließlich ein XML-Dokument zur Verfügung steht, welches der ursprünglichen Version der DTD ohne XGens folgt. Weiterhin muss sich der Benutzer überlegen, wie die Prototypen definiert sein sollen. Im Fall des XML-basierten Wissens von NICA sollen die Prototypen Topics und TopicObjects enthalten. Dazu gehören auch sämtliche Unterele-

mente. Ferner muss der Benutzer entscheiden, an welchen Stellen ein VarCall-Element zulässig ist und dies in der DTD umsetzen.

Nachdem diese Schritte durchgeführt wurden, besteht das Problem, dass durch die Definition des Prototyps mehrere Definitionen eines TopicObjects in der DTD vorhanden sind. Darauf gehen wir im folgenden Abschnitt ein.

4.2 Namensräume

Namensräume sind dafür geeignet, mehrere Definitionen gleicher Elemente mit unterschiedlichen Bedeutungen auseinander zu halten. Häufig kommt dies vor, wenn mehrere XML-Dokumente bereits vorhanden sind und diese zu einem Dokument zusammengefasst werden sollen. Dabei kann es zu Namenskonflikten kommen. Um diese aufzulösen, wird jedem Dokument ein eigener Namensraum zugewiesen, so dass die Konflikte über die Zuordnung zu den Namensräumen aufgelöst werden [NSE, NSG].

Leider arbeiten Namensräume und DTDs nicht miteinander zusammen, da die Definitionssprache DTD zeitlich vor den Namensräumen entwickelt wurde. Erst in *XML-Schema* werden auch Namensräume unterstützt. Aber es gibt Wege, Namensräume in DTDs einzubinden. Der erste besteht darin, dass die Namensräume fest in die DTD hinein geschrieben werden. Eine zweite Möglichkeit ist, Entitäten für die Namensräume zu definieren. So kann die Bezeichnung der Namensräume einfach geändert werden.

Im Falle der Prototypen wird der Namensraum *prot* definiert. Alle Elemente, die zu dem Prototypen gehören, befinden sich per Definition der DTD in diesem Namensraum. Somit kann zwischen den TopicObjects des Wissens und den TopicObjects im Prototyp unterschieden werden.

4.3 Verwendungsmöglichkeiten

Um die XML-Daten, die mit Datenbankinformationen angereichert werden, zu generieren, wird ein Präprozessor benutzt, der schließlich ein XML-Dokument liefert, welches die aktuellen Informationen der Datenbank enthält. Allerdings ist es durchaus möglich, den XGen auch zur Laufzeit zu benutzen, d. h. man kann ihn nicht nur mit Hilfe eines XML-Dokumentes benutzen, sondern in Java direkt XML-Objekte erzeugen. Ferner besteht eine hohe Flexibilität bei der Verwendung von Datenbanken und Prototypen, wie im Folgenden gezeigt wird.

4.3.1 Datenbanken

Solange es sich um eine Datenbank handelt, die JDBC bzw. ODBC unterstützt, kann sie für die Generierung von Daten verwendet werden. Also können wir praktisch jede relationale Datenbank einsetzen, da diese die Bedingung im Allgemeinen erfüllen. Dazu wird der entsprechende JDBC-Treiber installiert und der SQL-Ausdruck geschrieben. Wenn eine ODBC-Schnittstelle verwendet wird, muss die Datenbankverbindung noch konfiguriert und die entsprechende ODBC-JDBC-Brücke verwendet werden. Wie eine ODBC-Quelle eingerichtet wird, ist u. a. in [ODB] nachzulesen.

4.3.2 Prototypen

Die Prototypen werden selbst definiert und entsprechend der eigenen Ansprüche erstellt. Dabei soll es sich später um wohlgeformte und gültige XML-Daten handeln. Dies ist auch der Grund, warum in Attributen keine Var-Calls zulässig sind (siehe Anhang B.3). Sofern auch die Behandlung von NULL-Werten aus der Datenbank erwünscht ist, kann der DTD-Abschnitt der Alternative-Elemente übernommen werden. Ansonsten gibt es keine Einschränkungen für die Definition der Prototypen.

4.3.3 Laufzeit

Neben dem Benutzen des XGens als Präprozessor kann er auch während der Laufzeit eingesetzt werden. Dazu wird ein GenKnowledge-Objekt erzeugt, in dem alle Daten, die normalerweise im XGen angegeben werden, per Hand zu übergeben sind. Somit bleibt noch der Prototyp, der spezifiziert werden muss. Danach ist die angegebene Datenbank abzufragen und anhand des Prototyps der XML-Stream zu erzeugen. So wird es möglich, unabhängig von den definierten XGens (abgesehen vom Prototyp) Wissen zur Laufzeit zu generieren. Das bedeutet, dass ganz aktuell die Informationen aus der Datenbank in NICA übertragen werden könnten.

Kapitel 5

Zusammenfassung

Die Integration von Informationen aus Datenbanken in XML ist heute noch ein Forschungsthema, zu dem es keine allgemeinen Standards bzw. Werkzeuge – allenfalls Vorschläge – gibt. Es gibt verschiedene Wege, Datenbanken und XML zusammenzuführen, wobei diese immer stark von den Erfordernissen der Anwendung abhängen. Die Herausforderung der Umsetzung liegt in der wahlfreien Übernahme von Daten aus Datenbanken in eine XML-Struktur, wobei die Strukturen der Datenbank und von XML beliebig sind. Weiterhin sollen die erzeugten XML-Objekte in ein XML-Dokument eingebettet werden.

In dieser Studienarbeit wird ein Modell vorgestellt, mit dem genau dieses Problem gelöst wird. Dabei werden viele praktische Hindernisse behandelt, ohne dass große Einschränkungen vorzunehmen wären.

Das Modell wurde zunächst für das XML-Wissen des StudiBots entwickelt. Dabei hat sich letztendlich ein flexibles System ergeben, welches bei der Auswahl der Datenbank und beim Integrieren des Ergebnisses in XML-Dokumente sehr viele Freiheiten lässt. Einzige Voraussetzung für die Datenbank ist, dass sie die Schnittstelle ODBC oder JDBC unterstützt. Für den XML-Text kann jeder seine eigenen Definitionen verwenden, wobei freigestellt ist, ob die Behandlung von fehlerhaften Daten aus der Datenbank berücksichtigt werden soll. Dabei können die Ergebnisse der Datenbankanfrage beliebig eingebaut werden, sofern Elemente an diesen Stellen im Dokument zulässig sind. Das schließt aus, dass Attribute mit Inhalten aus der Datenbank gefüllt werden, allerdings ist es in vielen Fällen meist günstiger, solche Attribute durch Elemente zu realisieren.

Durch die Flexibilität der XGen bezüglich der Datenbankanbindung und der Struktur des zu erzeugenden XML-Textes kann die spezielle Entwicklung für den StudiBot direkt für NICA übernommen und gegebenenfalls auch für andere Systeme angepasst werden.

Anhang A

Anleitung zur Erstellung eines XGens

Um in XML einen neuen XGen zu spezifizieren, muss wie folgt vorgegangen werden:

Voraussetzung: Datenbank mit JDBC- oder ODBC-Anbindung und den nötigen Informationen (Login, Passwort, SQL-Ausdruck, vgl. Abschnitt 3.1.1).

1. Ein neues XGen-Element in einem XML-Editor anlegen, welches nach einer entsprechenden DTD spezifiziert ist.

2. Die nötigen Datenbankinformationen ausfüllen, also

   - Datenbanktreiber (z. B. `<DBDriver>`sun.jdbc.odbc.JdbcOdbcDriver`</DBDriver>`)
   - URL der Datenbank (z. B. `<DBURL>`JDBC:ODBC:Test`</DBURL>` für eine Microsoft Access Datenbank mit dem Namen Test.mdb, ODBC muss dafür eingerichtet sein)
   - Login
   - Passwort, dies muss verschlüsselt eingegeben werden. Dafür werden folgende Informationen benötigt:
     - Algorithmus zum Verschlüsseln des Passwortes
     - Evtl. Algorithmus zum Verschlüsseln des Schlüssels, der im XGen mitgegeben wird

     Alle weiteren Informationen sind über das GUI-Tool „Password-GUI“ erhältlich. Dies sind der verschlüsselte Passwort-String, evtl. der verschlüsselte Schlüssel und evtl. der Initialisierungsvektor.

Diese Informationen werden entsprechend der DTD eingetragen. In dem GUI-Tool kann man auch die Algorithmen auswählen und übernehmen. Falls man andere benutzen möchte, schaue man die möglichen Algorithmen unter [BCP] nach.

Beispiel (symmetrisches Verschlüsselungsverfahren im CBC-Modus, Schlüssel wird asymmetrisch verschlüsselt und mitgegeben):

```
<DBPassword>
 <EncryptedData
    Type="http://www.w3.org/2001/04/xmlenc#Content"
    xmlns="http://www.w3.org/2001/04/xmlenc#">
  <EncryptionMethod
    Algorithm="http://www.w3.org/2001/04/xmlenc
              #AES/CBC/PKCS5Padding">
   <IV>gd/cjtcprfpGlFkHU4vXGQ==</IV>
  </EncryptionMethod>
  <ds:KeyInfo
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
   <EncryptedKey>
    <EncryptionMethod
        Algorithm="http://www.w3.org/2001/04/xmlenc
                   #RSA/NONE/OAEPPadding"/>
    <CipherData>
     <CipherValue>+kIM4eUT/PBqUhVC2v/8=</CipherValue>
    </CipherData>
   </EncryptedKey>
   <ds:KeyName>AgtiveKey</ds:KeyName>
  </ds:KeyInfo>
  <CipherData>
   <CipherValue>RmOnyz6ZpeY8dcQ9TtNbRw==</CipherValue>
  </CipherData>
 </EncryptedData>
</DBPassword>
```

- SQL-Ausdruck (z. B. `<SQLQuery>SELECT * FROM Professoren; </SQLQuery>`)
- Variablendefinitionen angeben; hier alle Attribute eines Tupels eintragen, also für jede Spalte, die der SQL-Ausdruck liefert, einen Eintrag machen. Beispiel für einen Eintrag:

```
<VariableDef>
```

```
        <LabelInSQL>SQL-Name</LabelInSQL>
        <LocalName>neuerName</LocalName>
    </VariableDef>
```

3. Den Prototypen nach Bedarf spezifizieren. Variablen können mittels

   ```
   <VarCall>neuerName</VarCall>
   ```

   aufgerufen werden.

   Speziell für NICA:
   Wie gewohnt die TopicObjects erstellen. Dabei möglichst das Element Name der TopicObjects sinnvoll mit einem VarCall belegen, so dass diese eindeutig werden. Das VariableText-Konzept für Text verwenden und an den gewünschten Stellen Variablen einfügen. Evtl. die TopicObjects mit Alternative-Elementen oder SkipIfNull-Attributen versehen.

Anhang B

Ideensammlung

Hier wird dargestellt, welche Ideen zu den XGens weiterhin existieren und warum sie nicht umgesetzt wurden.

B.1 Rekursive XGens

Rekursive XGens sind Generatoren, die innerhalb ihres Prototyps weitere Generatoren enthalten. Dabei müssen zunächst die XGens innerhalb des Prototyps ausgeführt werden, bevor der übergeordnete XGen den XML-Text erzeugt.

Solange man nur *eine* Datenbank abfragt, lohnt sich die Umsetzung von rekursiven XGens nicht, da dies über SQL-Ausdrücke realisiert werden kann. Erst wenn man mehr als eine Datenbank innerhalb *eines* XGens verwenden möchte, lohnt sich die Rekursivität. Allerdings ist die Umsetzung kompliziert, da zunächst rekursive XGens behandelt werden müssten und weiterhin Probleme bei der Integration von Informationen zweier (oder mehr) Datenbanken auftreten würden. Zuerst würde über die Anzahl der Ergebnistupel der ersten Datenbank iteriert werden, und danach über die der zweiten. Somit hätte wir das Kreuzprodukt realisiert. Meistens wird aber eine Verbindungsrelation (Join) benötigt, die wir hier nicht definieren können. Weil außerdem keine aktuelle Anwendungssituation denkbar ist, wird auf rekursive XGens verzichtet.

B.2 Auslesen der XML Deklaration (InfoSet)

Das InfoSet steht in XML-Dokumenten in der ersten Zeile, die die Kodierung und die Version des XML-Dokumentes angibt. Um mit Hilfe eines Parsers dieses InfoSet auszulesen, sind zwar die notwendigen Voraussetzungen bereits

gegeben, allerdings werden sie von den meisten heutigen Parsern noch nicht unterstützt. SAX bietet den sog. *Locator2* an, der die nötigen Funktionen zum Auslesen des InfoSets enthält. Allerdings ist es noch nicht vorgeschrieben, dass ein SAX-konformer Parser diesen Locator2 unterstützt. Es wird lediglich der Locator vorausgesetzt. Somit existiert z. Z. nur ein Parser (AElfred2), der den Locator2 implementiert [OTC].

B.3 Elemente vs. Entitäten für Variablenaufrufe

Elemente dürfen nicht Inhalt eines Attributes sein. Falls aber ein Variablenaufruf in einem Attribut gewünscht wird und das Attribut nicht als Element geschrieben werden soll, kann man statt Elementen auch Entitäten für die Variablenaufrufe im Prototypen verwenden. Dazu führt man zwei neue Entitäten ein (z. B. `VarCall-start` und `VarCall-end`), die den Namen einer Variable umschließen.

Dabei treten folgende Probleme auf: Zunächst kann man dem Wissensingenieur nicht mehr verbieten, an bestimmten Stellen VarCalls einzusetzen, da Entitäten überall in einem wohlgeformten XML-Dokument erlaubt sind. Weiterhin gibt es – bedingt durch die sofortige Auflösung der Entitäten durch die Parser – bei der Implementierung Schwierigkeiten, die nicht ohne weiteres zu lösen sind. Da die einzelnen Handler in mehreren Stufen durchlaufen werden, sind ihnen nicht alle Informationen zugänglich, d. h. nur der erste Handler kennt die Entitätsdefinitionen. Alle anderen Handler betrachten die Entitäten als normalen Text. Dies lässt sich nur schwer umgehen.

Da es in vielen Fällen sinnvoll ist, ein Attribut als Element umzudefinieren, wurde die komfortablere Lösung mit den Elementen vorgezogen.

B.4 XML-Schema

Bislang werden nur DTDs für die Schemadefinition verwendet, obwohl auch XML-Schema existiert und sogar an manchen Stellen erwähnt wird. Das liegt daran, dass XML-Schema z. Z. noch nicht von den XML-Werkzeugen für Java (SAX, JAXB etc.) unterstützt wird. Daher lässt es sich trotz einiger im Text genannter Vorteile gegenüber der DTD (z. B. bessere Konsistenzbedingungen) nicht verwenden.

Anhang C

Glossar

Da im XML-Bereich sehr viel mit Abkürzungen gearbeitet wird, hier nochmals eine Zusammenfassung der wichtigsten Begriffe, die in dieser Studienarbeit verwendet werden.

Base64 Eine Codierung für Binärdaten zur Übertragung von E-Mail-Textkörpern gemäß der MIME-RFC von IETF. Base64 besteht lediglich aus 64 Codierungszeichen (A-Z, a-z, 0-9, +, /), die eine Teilmenge von US-ASCII darstellen.

DOM Das Document Object Model ist eine plattform- und sprachneutrale Schnittstelle, die ein Standardmodell dafür bietet, wie die Objekte in einem XML-Objekt zusammengesetzt werden. Weiterhin gewährt dies Modell eine Standardschnittstelle für den Zugriff auf diese Objekte sowie das Manipulieren ihrer Beziehungen zueinander.

DTD Eine DTD (Document Type Definition) beschreibt die Struktur einer Klasse von SGML- oder XML-Dokumenten, also einer SGML- oder XML-Applikation, mit Hilfe eines Text-Files, das alle Syntax-Regeln in einem von SGML vorgeschriebenen Format enthält. Beispielsweise ist jede HTML-Version durch eine DTD definiert. Eine Alternative dazu ist die Definition mit Hilfe eines Schemas.

Entität Eine Entität ist ein Verweis auf Daten, der ihre Wiederverwendung erleichtert und die Dokumentgröße minimiert. Entwickler können mit Entitäten die Handhabung wiederholt vorkommender Informationen vereinfachen.

InfoSet Ein XML-Dokument beginnt immer mit dem InfoSet (Information Set). Es beinhaltet die Kodierung und die XML-Version.

JAXB Java Architecture for XML Binding ist eine Java-API zur Unterstützung bei der Entwicklung von XML Data Binding-Anwendungen in Java.

JAXP Java API for XML Processing ist eine Java-API zur Unterstützung bei der Entwicklung von XML-verarbeitenden Anwendungen mit SAX bzw. DOM.

JCE Java Cryptography Extension liefert als eine Extension ein Framework für Kryptographie und Verschlüsselung (Austausch von Schlüsseln) als Drop-In Package für Java.

JCE-Provider Um das JCE-Package verwenden zu können, muss ein sog. Provider die Algorithmen zur Verfügung stellen, da das JCE nur die Schnittstellen liefert.

JDBC Java Database Connectivity, eine standardisierte Datenbankschnittstelle für Java. Diese Technologie erlaubt das Verwenden einer einmal erstellten Anwendung mit jeder SQL-Datenbank, die über einen JDBC-Treiber verfügt.

Namespace W3C-Aktivität im Zusammenhang mit XML. Namespaces sollen Dokumente in die Lage versetzen, in fremden DTDs angegebene Namen zu verwenden. Eine Namespace-Deklaration innerhalb eines XML-Dokuments verweist über einen URI auf einen Namespace „ns“. Daher sind die Namen in diesem Namespace in der Form *ns:name* in einem bestimmten Teil des Dokumentenbaums verfügbar.

NICA NICA bedeutet Natural language Intelligent Consulting Agent und ist ein von der Firma Agtive Systems entwickeltes System zur Erstellung intelligenter Beratungssysteme. Eine Anwendung davon ist der StudiBot, der speziell für die Universität Karlsruhe entwickelt wurde.

ODBC Unter ODBC (Open DataBase Connectivity) versteht man eine 1992 von der Firma Microsoft entwickelte Software-Schnittstelle, die den Zugriff aus einem Anwendungsprogramm auf unterschiedliche Datenbanken gewährleisten soll.

Processing Instruction Anweisung an den XML Parser oder XSL Prozessor innerhalb eines XML-Dokumentes.

Prototyp Der Prototyp ist ein XML-Abschnitt, der aussagt, wie die Daten aus der Datenbank in XML konvertiert werden sollen. An ausgewählten Stellen wird dies durch ein VarCall angezeigt.

SAX SAX (Simple API for XML) ist eine Programm-Schnittstelle (Application Programmers Interface = API) für die Verarbeitung einer Klasse von XML-Dokumenten, also einer XML-Applikation, mit Hilfe einer objektorientierten Programmiersprache wie z. B. Java. SAX liefert ein XML-Element nach dem anderen in einem Eingabestrom und eignet sich daher auch für sehr große XML-Files.

SGML SGML (Standard Generalized Markup Language) ist eine Meta-Sprache, in der Markup-Sprachen wie z. B. HTML definiert werden können. Die Struktur und Syntax solcher Markup-Sprachen („SGML-Anwendungen“) wird mit einer DTD festgelegt.

SQL SQL (Structured Query Language) wurde von IBM zur Abfrage von relationalen Datenbanken entwickelt.

StudiBot Der StudiBot stellt ein interaktives System dar, welchem man Fragen bezüglich des Informatik-Studiums an der Universität Karlsruhe stellen kann. Der StudiBot ist eine Anwendung von NICA.

Topic Ein Topic ist ein Themenkreis, der wieder weitere Themenkreise und TopicObjects enthalten kann.

TopicObject Ein TopicObject gehört zu einem speziellen Thema und repräsentiert dabei ein Objekt – wie z. B. ein einzelner Professor im Themenkreis der Professoren.

URI Universal Resource Identifier, entweder ein URL oder ein URN. Ein URI ist ein Mittel zur Angabe eines Inhalts im Web. Dabei kann es sich um eine Textseite, ein Video, ein Tonelement, ein Bild oder ein Programm handeln.

URL Universal Resource Locator, eine eindeutige Adresse eines Dokuments oder einer Ressource im Internet im Format protokoll://serverdomänenname/pfadname. Beispiele für Protokolle sind HTTP und FTP.

URN Uniform Resource Name, eine Ressource im Internet mit einem Namen, der eine persistente Bedeutung hat. Der Benutzer einer URN kann somit davon ausgehen, dass eine andere Person oder ein Programm in der Lage ist, diese Ressource zu finden. Ein URN gleicht einer Adresse einer Webseite bzw. einem URL (Uniform Resource Locator). Bei einer URN braucht der Benutzer nur den Namen der Ressource zu kennen, d. h. er muss nicht wissen, wo sich die Ressourcen befinden.

UTF-8 Unicode-Format, das eine variable Anzahl von Bit pro Zeichen verwendet (im Gegensatz zu UTF-16, bei dem stets 16 Bit pro Zeichen verwendet werden).

VarCall Ein Element, welches einen Variablennamen enthält, der einem Wert aus der Datenbank zugewiesen werden kann.

W3C World Wide Web Consortium, ein Zusammenschluss von Firmen der IT-Branche, der durch Spezifikationen und Referenzsoftware die Anwendung von Standards für die Weiterentwicklung des Web und für das Zusammenwirken von WWW-Produkten fördert. Obwohl das W3C von Mitgliederfirmen finanziell unterstützt wird, ist es herstellerneutral. Die Produkte des W3C sind kostenlos erhältlich.

XML Extensible Markup Language, Weiterentwicklung des SGML-Standards. Im Gegensatz zu SGML-Dokumenten benötigen XML-Dokumente keine Schemabeschreibung in Form einer DTD. Sie bestehen hauptsächlich aus Text und Tags; die Tags erzeugen eine Baumstruktur im Dokument. Wenn das XML-Dokument ordnungsgemäß strukturiert ist, d.h. wenn eine ordnungsgemäße Verschachtelung der Tags vorliegt, wird es als „well-formed“ (korrekt strukturiert) bezeichnet. Wenn zusätzlich eine DTD für das Dokument existiert, wird es als „valid“ (gültig) bezeichnet.

XML Data Binding Konzept zur Verarbeitung von XML-Daten mit Hilfe objekt-orientierter Programme. Es umfasst Unmarshalling von XML-Daten (Umwandlung in Objekte) und Marshalling von Objekten (Umwandlung in XML-Daten) und befasst sich mit der Idee, XML-verarbeitende Anwendungen durch die Übersetzung eines XML-Schemas zu entwickeln.

XML-Schema Ein Schema beschreibt die Struktur einer Klasse von XML-Dokumenten, also einer XML-Applikation, ähnlich wie eine DTD, jedoch nicht in der DTD-Syntax sondern in einer eigenen XML-Syntax.

XQuery Entwurf für die Implementierung einer XML-Abfragesprache, die durch das W3C veröffentlicht wurde. Sie nutzt die Struktur von XML zur Abfrage von Daten und benutzt dabei eine SQL-ähnliche Syntax.

XSL Mit XSL (Extensible Style Language) wird ein Style-Sheet definiert, das angibt, wie der in einem XML-Dokument definierte Inhalt vom Web-Browser oder von anderen Programmen dargestellt werden soll. XSL ist mächtiger als CSS und DHTML:

- Mit XSLT (Transformation) kann man aus einem XML-File ein anderes XML-File machen, also z.B. bestimmte Elemente weglassen, die Elemente in anderen Reihenfolgen anordnen und zusätzliche Elemente hinzufügen.
- Mit XSL-FO (Formatierung) kann man das Layout der Darstellung für die Elemente festlegen.

Literatur

[Bar00] R. Bartel: *Kryptographische Operationen in Java. Analyse und Darstellung der Möglichkeiten der JCA, JCE und Entrust-Implementierung.* Studienarbeit, Arbeitsbereich Softwaresysteme, Technische Universität Hamburg-Harburg, Januar 2000.

[BCP] *Legion of the Bouncy Castle.* http://www.bouncycastle.org/.

[Enc] *XML Encryption Syntax and Processing, W3C.* http://www.w3.org/TR/xmlenc-core/.

[HV01] H. Holzapfel und M. Völkel: *Entwurf und Implementierung eines webbasierten Dialogsystems für die Studienberatung.* Studienarbeit, Zentrum für Multimedia, Universität Karlsruhe (TH), Juni 2001.

[i3v] *Homepage der Entwicklungsfirma des Informationssystems i3v.* http://www.ginit.de/.

[IS] *XML Information Set, W3C.* http://www.w3.org/TR/xml-infoset/.

[JCE] *Java Cryptography Extension (JCE).* http://java.sun.com/products/jce/.

[JDO] *JDOM.* http://www.jdom.org/.

[JXB] *Java Architecture for XML Binding (JAXB).* http://java.sun.com/xml/jaxb/index.html.

[KEY] *keytool - Key and Certificate Management Tool.* http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/keytool.html.

[NSE] *Namespaces in XML, W3C.* http://www.w3.org/TR/REC-xml-names/.

[NSG] *Namespaces in XML, Übersetzung.* http://www.schumacher-netz.de /TR/1999/REC-xml-names-19990114-de.html.

[ODB] *Einrichten einer ODBC-Datenquelle.* http://www.techbase.de/main/cnsoft/ti1037.htm.

[OTC] *OpenText Corporation.* http://www.opentext.com/microstar/.

[Pro] *Overview of JCA/JCE Implementations.* http://www.nue.et-inf.uni-siegen.de/SignStreams/csp/overview_provider.html.

[SAX] *Offizielle Homepage für SAX, der einfachen API für XML (Simple API for XML).* http://www.saxproject.org/.

[TUT] *The Java API for Xml Processing Tutorial.* http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/index.html.

[W3C] *W3C, World Wide Web Consortium.* http://www.w3.org/.

[XQ] *XQuery, W3C.* http://www.w3.org/TR/xquery/.