Aus dem Institut für Informationssysteme
der Universität zu Lübeck
Direktor:
Prof. Dr. rer. nat. Volker Linnemann

Design and Implementation of a Database Programming Language for XML-based Applications

Inauguraldissertation
zur
Erlangung der Doktorwürde
der Universität zu Lübeck
- Aus der Technisch-Naturwissenschaftlichen Fakultät -

Vorgelegt von
Henrike Schuhart
aus Geesthacht

Lübeck, März 2006

Acknowledgements

This work is the result of my research during the last three years. I have been working as a research assistent at the institute of information systems of the university of Lübeck. It started with searching for suitable topics in context of the predecessor project **XOBE** and proceeded to finding solutions, implementing the prototype and finally writing this doctoral thesis. During all this time a lot of people have supported my work and goals. Now I want to express my thanks to all involved persons.

My special thanks goes to my supervisor Prof. Dr. Volker Linnemann, who is also the director of the institute where this work has been developed. I appreciate that he always had time to discuss, criticize and support my research. I thank Prof. Dr. Stefan Fischer for being the second referee, which is connected with a non negligible amount of time. Likewise I would like to thank Prof. Dr. Bernd Fischer to be the chair of the examination board.

Many thanks as well to my colleague Beda C. Hammerschmidt for encouraging discussions in context of my work, in particular reading and improving my publications. Special thanks as well to Dominik Pietzsch with whom I developed the persistency layer for **XOBE**$_{\text{DBPL}}$, which is now going to be an independent project.

I would like to thank Simon Gilbert for proof-reading this work in English.

Finally I want to thank my friend Roberto for supporting me and my work all the time. Thanks especially to my parents Helmut und Heidi Schuhart for providing me with a smooth study and education right from the beginning. Moreover they have always encouraged my plans.

Lübeck, June 2006 Henrike Schuhart

Abstract

XML is the de facto standard for data exchange between arbitrary applications. These applications are written in object-oriented programming languages like Java or C# for example. Consequently the need arises to integrate XML into existing object-oriented programming languages. Moreover, many applications have to keep and deal with persistent data and objects. A lot of frameworks are currently being developed which are trying to solve mismatch problems between the transient object model and the persistent data model. Due to performance reasons relational databases are taken in general. Although existing approaches try to minimize the mapping effort, true transparent persistency is not gained at the moment. In this work transparent persistency means that objects can be stored regardless of type, algorithms remain unchanged whether they operate on persistent or transient data and finally that programmers, if at all, deal with persistency on a very high level. In addition many persistence solutions do not even support the main object-oriented concepts like inheritance and polymorphism. As well object-oriented databases are limited to a single programming language and are rarely available. All that the frameworks or approaches mentioned so far have got in common is that the programmer has to add special code when dealing with persistent objects or classes. The code includes communication with a so-called persistency manager for example. In the past several database programming languages have been developed, but they all have tried to integrate the relational model. Consequently, a holistic and transparent solution is needed that integrates XML and persistency into an existing object-oriented programming language.

In this work **XOBE**$_{\text{DBPL}}$ (**X**ML **OB**j**E**ts **D**ata**B**ase **P**rogramming **L**anguage) is developed as an extension of the object-oriented programming language Java. **XOBE**$_{\text{DBPL}}$ is based on its predecessor project **XOBE**. In **XOBE** XML and XPath are already integrated and checked statically at compile time against an XML schema. XML Schemas and DTDs are supported as XML schema description languages. In **XOBE**$_{\text{DBPL}}$ XML objects can now be manipulated with the help of updates. Moreover the type checking process of **XOBE** is extended to support updates and more complex queries as well. Additionally, transparent type independent persistency supporting all object-oriented concepts is introduced in **XOBE**$_{\text{DBPL}}$ for the first time. The prototypical implementation is based on a persistency layer using web services. The persistency layer keeps persistent objects and data and offers the possibility to exchange data among arbitrary languages and applications. Moreover, it is shown that the introduced concepts can also be realized as a semantical extension of Java. Consequently programmers do not have to learn

a new syntax anymore and existing development environments can be used, which supports efficiency.

Contents

Chapter 1

Introduction

This work focuses on two main aspects, the seamless integration of XML and persistency concepts into the object-oriented programming language Java.

XML is the de facto standard data exchange format between arbitrary applications. There have been many efforts to integrate XML into programming languages reaching from the simple document object model (DOM) to whole XML class generators. These approaches are available in most popular programming languages.

The integration of persistency into programming languages has been done by database programming languages as well as by certain new popular frameworks like Hibernate or approaches like EJB. Nevertheless, these approaches suffer from certain limitations concerning in particular transparency and object-orientation. While existing database programming languages integrate the relational model, Hibernate and EJB 3.x does not support polymorphism in general. EJB 2.x does not even support inheritance. In addition, although they try to, the approaches except by some database programming languages are not transparent. In this work transparency means that arbitrary types may become persistent. Moreover, algorithms remain unchanged whether they are executed on transient or persistent objects. Finally, users can work with persistency on a very high level.

Since there are so many currently developed frameworks trying to solve the integration problem of XML and persistency into object-oriented programming languages, the need for a holistic and transparent object-oriented database programming language seems to be there. The starting point of **XOBE**$_{\text{DBPL}}$, which stands for **X**ML **OB**j**E**cts **D**atabase **P**rogramming **L**anguage, is the predecessor project **XOBE** [43]. **XOBE** concentrates on the integration of XML objects and XPath as the query language for these objects. The most important feature of **XOBE** is that each XML operation is statically type checked against the declared XML schema. In **XOBE**$_{\text{DBPL}}$ the XML integration is extended regarding the manipulation of XML objects. Before, XML objects could only be queried but not updated. The static type checking idea is kept and enhanced to include updates. While **XOBE**'s intentions lie on the development of web applications, all objects can remain transient. **XOBE**$_{\text{DBPL}}$ is supposed to deal with persistent objects as well.

1.1 Motivation

After introducing the context a short motivation for the main topics of this work is given. These topics are first of all the integration of XML and suitable operations into existing programming languages. Second, the advantage of static type checking in contrast to dynamic type checking is presented and finally, the transparent integration of persistency into existing programming languages is shown. Java is chosen as the source programming language. This is because Java is popular and many programming tools like development environments are available as open sources. Moreover, in many ways Java can be seen as deputy for object-oriented programming languages. Hence concepts and their realization introduced in **XOBE**$_{\text{DBPL}}$ are not limited to extending Java.

XML in Programming Languages

Since XML is the de facto standard for data exchange, many data is available in XML only. Moreover, most databases regardless of storing data (object-)relationally or natively in XML, offer applications an XML view of their data. In contrast to XML as data exchange format, object-oriented programming languages like Java and C# are today's de facto standard for application development. Consequently the processing of XML data is important for these applications. There are many approaches that try to integrate XML, either by mapping it to general models like the DOM or by generating specific XML classes. In fact, XML data cannot be mapped to object-oriented classes without losing information in general. In **XOBE**$_{\text{DBPL}}$ XML is integrated as XML into the existing object-oriented programming language Java.

Static Type Checking

Approaches that deal with XML reach from programming languages to XML databases. If at all, most of them check XML at runtime against the required schema. A schema defines certain structural and value-based constraints on XML documents. But checking XML documents at runtime means that this checking process has to be repeated each time the program is executed. Moreover checking in particular an XML document at runtime often means parsing the whole document over and over again. More generally spoken, dynamic in contrast to the static type checking process depends on the data's size. If the manipulated data is larger, the dynamic type checking process will take longer. This is an important problem in the context of XML and updates, since XML databases often consist of a single document.

In contrast, static type checking is done only once at compile time. If a program is checked statically, it treats any possible document, maybe restricted to a certain XML schema, correctly. Please notice that Java and most other object-oriented programming languages are statically type checked for general objects and method invocations anyway. If such a program is checked it can be run arbitrarily often without the need for checking it anymore. It is guaranteed to run correctly. Although static type checking is not decidable in general, it can detect most errors in advance and saves a lot of time in program development. In **XOBE**$_{\text{DBPL}}$ the static type checking concept of its source programming language Java is enhanced to the integrated XML. In addition, XML data and its corresponding operations are statically type checked in **XOBE**$_{\text{DBPL}}$, including XPath for querying and update expressions for updates.

Transparent Persistency

Besides the integration of XML into programming languages another topic for popular programming languages seems to be gaining more and more importance. The topic is called persistency. When speaking about programs dealing with objects, these objects will normally disappear, if the program stops. In object-oriented programming languages objects have got a transient life time. Nevertheless many applications need to keep these objects persistently, meaning that they exist independently of a program's execution time. Hence, most approaches and frameworks provide interfaces to databases. Due to performance reasons, these databases are most often relational databases. Since relational databases don't store objects, the programmer is responsible for mapping these objects to the underlying relational model. There are many frameworks available which try to offer high-level mapping tools. Nevertheless, persistency is not integrated transparently as defined previously. In particular, communication with a so-called persistence manager is needed or object-oriented concepts like inheritance are not supported. Object-oriented databases are out of date, limited to a single programming language and also effort intransparent communication. In the past some database programming languages were developed but most of them tried to integrate the relational model into existing programming languages. **XOBE**$_{\text{DBPL}}$ focuses on true transparent persistency. Persistency is nearly hidden from the programmer, every type, in particular XML, is accepted and all object-oriented concepts including inheritance are supported.

1.2 Goals and Organization of this Work

The main goal of this work is to develop a database programming language providing high-level and transparent integration of XML and persistency. In contrast to many other approaches this work focuses on a holistic and consistent solution. Moreover static type checking aspects in the context of XML are offered. This work is organized as follows: In chapter 2 an introduction to basics regarding the further chapters and an overview of related approaches are given. Chapter 3 introduces the integration of XML in **XOBE**$_{\text{DBPL}}$. The focus in this chapter lies on XML update syntax and semantics and the extended static type checking process. In chapter 4 persistency as a transparent and type independent concept is explained. Closely connected to persistency are transactions, which are introduced in chapter 5. Transactions guarantee in particular consistent access to persistent objects. In **XOBE**$_{\text{DBPL}}$ persistency is achieved by an underlying persistency layer, which is described in chapter 6. The persistency layer is based on web services and also provides programming language independent exchange. Chapter 7 demonstrates the architecture and implementation of the previously introduced concepts. While **XOBE**$_{\text{DBPL}}$ originally extends the Java programming language syntactically, an alternative approach is introduced in chapter 8. Once again Java is taken as the source, but this time only its semantics are extended to realize **XOBE**$_{\text{DBPL}}$'s concepts. In chapter 9 some first experimental results regarding the integration of XML updates and persistency are performed and evaluated. Finally, conclusions and an outlook on future work are given in chapter 10.

Chapter 2

Basics and Related Work

This chapter introduces important and needed topics in the context of the **XOBE**$_{\text{DBPL}}$ concepts as well as their realization. Besides, related approaches are also introduced. Most of these approaches can only be seen as related regarding certain aspects of **XOBE**$_{\text{DBPL}}$, concerning the integration of XML or object persistency in particular. Moreover, the advantages and disadvantages of the introduced related approaches are discussed.

2.1 Java

Java is an object-oriented programming language. Its specification can be found in [84] and [85] for versions 1.4.2 and 1.5.0 respectively. In Java 1.5 some important features were added, which are also available in C# and the .Net framework languages [54]. Among them are for example generic types, varargs, enhanced for-loops and annotations. The latter ones are especially interesting in the context of this work. **XOBE**$_{\text{DBPL}}$ is originally defined as syntactic and semantic enhancement of Java. These extensions are explained through the chapters 3 to 7. In contrast chapter 8 introduces another approach. An approach that tries to keep all **XOBE**$_{\text{DBPL}}$ concepts but without any syntax extensions to the Java programming language. The following constituents of Java play an important role for its realization.

2.1.1 Annotations

Annotations are metadata and a new feature in Java 1.5. Annotations are used to mark Java declarations, e.g. classes, methods or fields. An annotation indicates that the declared declaration should be processed in some special way e.g. by a (post)compiler on byte-code-level. Older versions of Java have rough annotation functionality. Built-in annotations like `@deprecated` have to be placed within a JavaDoc tag and cause the Java compiler to produce warnings. With Java 1.5, finally, annotations are a typed part of the language. Annotation types are blueprints for annotations equivalent to classes and objects and may contain arguments as well. The detection and further processing of user-defined annotations have to be written and added, e.g. to a postcompiler, by the programmer. Otherwise user-defined annotations are ignored by the Java compiler.

2.1.2 The Java Virtual Machine

The Java Virtual Machine (JVM) is the cornerstone of the Java and Java 2 platforms. It is the component of the technology responsible for its hardware - and operating system - independence, the small size of its compiled code, and its ability to protect users from malicious programs.

The JVM is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at runtime. The JVM knows nothing of the Java programming language, only of a particular binary format, the `class` file format. A `class` file contains JVM instructions (or `bytecodes`) and a symbol table, as well as other information.

For the sake of security, the JVM imposes strong format and structural constraints on the code in a `class` file. However, any language with functionality that can be expressed in terms of a valid `class` file can be hosted by the JVM. Consequently, this generally available and machine-independent platform can be used by other languages as a delivery vehicle too.

More details about the JVM can be found in [83].

2.1.3 The Byte Code Engineering Library

The Byte Code Engineering Library (BCEL) is intended to give users a convenient possibility to analyze, create and even manipulate (binary) Java class files. These classes are represented by objects which contain all the symbolic information of the given class, for example methods, fields and byte code instructions.

Such objects can be read from an existing file, be transformed by a program and dumped to another file again. An even more interesting application is the creation of classes or methods from scratch. This can even happen at runtime.

BCEL is suitable for several projects including compilers, optimizers, code generators and analysis tools. In this work it is used primarily as a code generator and analysis tool.

More details about BCEL are available in [3].

2.2 XML

The Extensible Markup Language (XML) is derived from the Standard Generalized Markup Language (SGML), which is already standardized by ISO 8879 and a W3C recommendation [98]. SGML is originally designed to meet the challenges of large-scale electronic publishing. The main idea is to strictly separate a document's presentation from its logical structure. Due to the propagation of the World Wide Web (WWW) and therefore the Hypertext Markup Language (HTML), another application of SGML, a second application task gains more and more importance, namely the task to exchange data instead of whole documents. This development finally leads to the XML specification. Today XML is the de facto standard format for data exchange in the context of the internet. The XML specification gained W3C recommendation status in 1998 for the first time.

Basically XML consists of a syntax to support data exchange between different applications. But only the fact that the syntax is standardized and is used in a lot of different areas and by many programs, makes XML an ongoing standard. Since one demand is human readability, XML documents and data are represented in text form. XML documents are structured with the help of elements. The beginning of an element is always marked with a start tag, e.g. `<site>` and its end respectively with an end tag, e.g. `</site>`. These tags are also called text markups. Within a start and an end tag the element can either contain simple text, other elements or a mixture of them. Elements that are part of another element's content are called subelements. Elements with an empty content can be abbreviated by `<site/>`. In this case the start and corresponding end tag are summarized by one tag. The name given in the start and regarding end tag together with an element's content structure define its type. The type of elements used throughout XML documents is defined by and depends solely on the user. An additional part of XML documents are attributes. Attributes belong to a certain XML element and are defined within a start tag. The example `<person id=''123''>` sets the value of the person's attribute `id` to `123`. Again, attribute names, their value types as well as their affiliation to a certain element are user defined and make up the type of an attribute. Finally XML documents can contain comments, e.g. `<!-- closed auctions -->`.

Certain conditions have to hold for XML documents. An XML document starts with a correct prolog followed by a single `document element`. Every element may contain sub or child elements as well as text data. Therefore the document element is the root of an arbitrarily deep nested hierarchical structure. Each element has to be correctly nested, meaning that opening and closing tags are enclosing it. Tags of different elements are not allowed to overlap each other. An element can contain any number of attributes or none. Besides, attribute names within one tag have to be unambiguous. In contrast to attributes, elements can have several subelements of the same type and name. Another important aspect is that attribute values are flat, meaning that their content cannot consist of elements. If all conditions hold, the XML document is called `well-formed`. Well-formedness is a rather weak condition, since it only guarantees that these documents can be represented by a tree structure after parsing. Definition 2.2.1 presents a simplified grammar for XML documents. The prolog is omitted.

Definition 2.2.1 An XML document is built up according to the following grammar:

document	$\rightarrow$	*element*
element	$\rightarrow$	*empty_element_tag* |
		start_tag content end_tag
start_tag	$\rightarrow$	**’<’** *name* (*attribute*)$*$ **’>’**
attribute	$\rightarrow$	*name* **’=’** *attribute_value*
end_tag	$\rightarrow$	**’</’** *name* **’>’**
content	$\rightarrow$	(*element* | *char_data* | *comment*)$*$
empty_element_tag	$\rightarrow$	’<’ *name* (*attribute*)$*$ **’/>’**
comment	$\rightarrow$	**’<!–’** *char_data* **’–>’**

The nonterminal symbol *attribute_value* stands for a string within single or double quotes, *name* represents an element name and *char_data* consists of alpha numerical symbols. □

Throughout this work an internet auction scenario is used for examples. A more detailed introduction is given in section 2.6. The most important constituents of an auction site are the auctions, either still open or already closed, and the items being auctioned as well as the participating persons. In 2.2.1 an example auction site XML document is given .

Example 2.2.1

```
<site>
        <regions>
         <africa/>
         <asia/>
         <australia/>
         <europe/>
         <namerica/>
         <samerica/>
        </regions>
        <categories>
                <category id="c00001">
                        <name>Software </name>
                        <description>Business , education and games</description>
                </category>
        </categories>
        <catgraph/>
        <people>
                <person id="p00001">
                        <name>Mary Fernandez </name>
                        <emailaddress>fernandez@xquery .com</emailaddress>
                        <creditcard >123−2345−235</creditcard>
                </person>
        </people>
        <open_auctions/>
        <closed_auctions/>
</site>
```

The example auctions site document consists of a root element called `site`. Within this root element is the `regions` element listing several possible regions e.g. `africa` or `namerica` in form of empty elements. The root child element `categories` contains itself a child el-

ement named `category`. The `category` element contains a name element, which has got simple text content e.g. *Software*, and a description element also containing simple text content, e.g. *Business, education and games*. Another root child element `people` contains a `person` child element. The `person` and `category` element given in the example include an attribute named `id`, which is used to identify a certain person or respectively a certain category. Finally, the empty elements `open_auctions` and `closed_auctions` are part of the site's children. The given document describes a rather small auction site without running or finished auctions and only one registered person and category. Nevertheless it gives an impression of XML documents.
As mentioned before, well-formedness is a weak condition upon XML documents. In most cases processes extracting or changing data of XML documents need more information about the element and attribute structure. For this purpose schema languages are available. Schema languages enable to define structural and value conditions for a certain class of XML documents. XML documents, which are well formed and obey all rules defined by a certain schema, are `valid` according to this schema . `Validity` is a much more expressive condition upon an XML document. In this work it is differentiated between `structural validity` and `value-based validity`. Structural validity refers to conditions limiting the number, occurences and structural aspects of the content of certain elements. Contrastly value-based validity regards to conditions restricting for example attribute values. In the following the most common XML schema languages DTD and XML Schema are introduced. There are in fact many XML schema languages e.g. RelaxNG [92], which are not going to be discussed in this work. DTD and XML Schema are W3C recommendations and are supported by **XOBE**$_{\text{DBPL}}$.

2.2.1 DTD

Theoretically each XML application program can check the document structure itself, but this procedure is error-prone and exhausting. SGML had already noticed this and introduced the Document Type Definition (DTD) . DTDs can only define conditions regarding document structure. A reduced and simplified version of these original DTDs is taken over by the W3C [98]. Now, with the help of a DTD, a parser reads the document and is able to check structural conditions. Each DTD can be seen as a description for a class of XML documents, comparable to classes and instances in the object-oriented paradigm. A document type definition consists of element types, attribute lists, entities, notations, comments and processing instructions. Element types describe element content with the help of regular expressions. Table 2.1 illustrates these constituents.

Processing instructions give hints to the parser, while comments are supposed for human readers of the DTD. Entities define physical building blocks. Entity references can then be used to build up documents or DTDs. The entity concept enables reusability of components. In XML several kinds of entities are mentioned, e.g. general, parameter, external, internal, parsed and unparsed. Since further details are not important, the most interesting DTD parts, namely ele-

<!ELEMENT person ...>	element type declaration
<!ATTLIST person ...>	attribute list declaration
<!ENTITY copyright "...">	entity declaration
<!NOTATION jpg ... >	notation declaration
<!– ... –>	comments
<? ... ?>	processing instructions

Table 2.1: Constituent parts of a document type definition (DTD)

ment and attribute list declarations are explained.
An element type declaration is used to define structural conditions upon an element and its content. The DTD offers five different content models. The `empty` content model defines elements without any content, e.g. `<!ELEMENT interest EMPTY>`. The `any` model allows element to have arbitrary content, e.g. `<!ELEMENT whatever-you-want ANY>`. Finally there are content models restricting an element's content exclusively to strings e.g. `<!ELEMENT shipping #PCDATA>`, exclusively to elements, e.g. `<!ELEMENT people (person*)>`, and a mixture of strings and subelements, e.g. `<!ELEMENT text (#PCDATA | bold)* >`. Element declarations with element as well as mixed content are based on additional element type declarations. Final element declarations are only allowed to have simple string content. String content is abbreviated with `#PCDATA`, which stands for parsed character data. In case of complex models the content can be defined further with the help of regular expressions. Regular expressions contain the sequence operator (`,`), the choice operator (|) and the following cardinality operators: `?`(0..1), `+`(1..n) and ∗(0..n).
In contrast to element type declarations, which define the structure between the start and end tag of an element, attribute type declarations are used to define attributes for a certain element type. For example defining an attribute `id` for the element type `person` looks like `<!ATTLIST person id ID #REQUIRED>`. Since attribute values are limited to basic types, the only valid attribute types in a DTD are predefined, e.g. `CDATA`, `ID` and `IDREF`. `CDATA` stands for character data, `ID` defines an unambiguous XML name within a document, which can be used as a key and `IDREF` defines a reference to a key value within this document. The last position of an attribute list declaration contains the attribute's usage restriction, which can be one of `#REQUIRED`, meaning the attribute must be given, `#IMPLIED`, meaning the attribute is optional and `#FIXED` followed by a certain value, which indicates that the attribute is required and has to have the given value.
The example 2.2.2 presents the parts of the XMark DTD defining people and open auctions .

Example 2.2.2

```
<!ELEMENT site          (regions, categories, catgraph, people, open_auctions, closed_auctions)>
...
```

```
<!ELEMENT people          (person*)>
<!ELEMENT person          (name, emailaddress, phone?, address?,
                           homepage?, creditcard?, profile?, watches?)>
<!ATTLIST person          id ID #REQUIRED>
<!ELEMENT emailaddress    (#PCDATA)>
<!ELEMENT phone           (#PCDATA)>
<!ELEMENT address         (street, city, country, province?, zipcode)>
<!ELEMENT street          (#PCDATA)>
<!ELEMENT city            (#PCDATA)>
<!ELEMENT province        (#PCDATA)>
<!ELEMENT zipcode         (#PCDATA)>
<!ELEMENT country         (#PCDATA)>
<!ELEMENT homepage        (#PCDATA)>
<!ELEMENT creditcard      (#PCDATA)>
<!ELEMENT profile         (interest*, education?, gender?, business, age?)>
<!ATTLIST profile         income CDATA #IMPLIED>
<!ELEMENT interest        EMPTY>
<!ATTLIST interest        category IDREF #REQUIRED>
<!ELEMENT education       (#PCDATA)>
<!ELEMENT income          (#PCDATA)>
<!ELEMENT gender          (#PCDATA)>
<!ELEMENT business        (#PCDATA)>
<!ELEMENT age             (#PCDATA)>
<!ELEMENT watches         (watch*)>
<!ELEMENT watch           EMPTY>
<!ATTLIST watch           open_auction IDREF #REQUIRED>

<!ELEMENT open_auctions   (open_auction*)>
<!ELEMENT open_auction    (initial, reserve?, bidder*, current, privacy?,
                           itemref, seller, annotation, quantity, type, interval)>
<!ATTLIST open_auction    id ID #REQUIRED>
<!ELEMENT privacy         (#PCDATA)>
<!ELEMENT initial         (#PCDATA)>
<!ELEMENT bidder          (date, time, personref, increase)>
<!ELEMENT seller          EMPTY>
<!ATTLIST seller          person IDREF #REQUIRED>
<!ELEMENT current         (#PCDATA)>
<!ELEMENT increase        (#PCDATA)>
<!ELEMENT type            (#PCDATA)>
<!ELEMENT interval        (start, end)>
<!ELEMENT start           (#PCDATA)>
<!ELEMENT end             (#PCDATA)>
<!ELEMENT time            (#PCDATA)>
<!ELEMENT status          (#PCDATA)>
<!ELEMENT amount          (#PCDATA)>
```

As one can see the `people` element describes an optional sequence of `person` elements. A person element's content itself contains a sequence of name, email address, phone, address, homepage, creditcard, profile and watches element types. Whereas only name and emailaddress are required constituents, the others are optional. The example DTD also defines an attribute list for the `person` element, which consists of one single required attribute called `id` of type ID.
A document can be associated with a DTD by a document type definition within the document's

prolog, e.g. `<!DOCTYPE auction SYSTEM ''auction.dtd''>`. A document's prolog can contain at most one document type definition.
DTDs are rather easy to write and read and often used in practice. Nevertheless DTDs possess some important drawbacks. First DTDs only allow for the two basic data types #PCDATA in the context of elements and #CDATA in the context of attributes. Especially basic data types derived from integers are missing. Another aspect is that element types can only be declared globally, which makes it impossible to define local element types with the same name but different content models. Last it is not possible to limit key references to keys of a certain element type. In the following section another W3C schema language is introduced, which is more expressive than the DTD but also much more complicated.

2.2.2 XML Schema

The XML Schema gained W3C recommendation status on May 2nd, 2001. The specification consists of three parts [100], [101] and [102]. In the following only the most important aspects, which are part of the first specification partition, will be given. While DTDs are written in a proprietary format, XML Schemas are written themselves in XML. In contrast to the DTD XML Schema predefines a larger number of basic data types, e.g. integer, double, time, date among many others. Additional user defined basic data types can be added. Like in most programming languages XML Schema allows to define list and union types from basic data types. Also element cardinalities can be defined more precisely with the help of minimum and maximum attributes. XML Schema defines a certain kind of inheritance, which is rather limited compared to the object-oriented understanding. In XML Schema types can only be restricted or extended by certain parts. Elements and types are clearly separated and namespaces are supported optimally. Due to namespaces different XML Schemas or parts of them can be composed to new schemas.
Since XML Schema definitions are themselves well-formed and valid XML documents, they contain a single root element called `xsd:schema`. The basic structure of an XML Schema describing the auction site is given in listing 2.1.

Listing 2.1: Basic XML Schema structure

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- An XML Schema for the auction site -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="..."
            ... >
        <!-- Auction XML Schema declarations -->
        ...
</xsd:schema>
```

The namespace `xmlns:xsd=''http://www.w3.org/2001/XMLSchema''` is used in XML Schema definitions, while the namespace
`xmlns:xsi=''http://www.w3.org/2001/XMLSchema-instance''`
is used in XML document instances, which are validated in reference to a declared XML

Schema.
XML Schema differentiates between simple and complex types. In contrast to simple types defining attributes and elements with simple content and without attributes, complex types define elements with at least one attribute or content containing subelements. For example the simple type element `name` is defined as follows
`<xsd:element name=''name'' type=''xsd:string''/ >` and the previously mentioned attribute `id` is defined by
`<xsd:attribute name=''id'' type=''xsd:ID''/>`.
Attributes as well as elements can be declared to have either a default or a fixed value. As mentioned before cardinalities known from the DTD are replaced and extended by `minOccurs` and `maxOccurs` attributes. The DTD cardinality operator `(?)` for example can be expressed in XML Schema syntax for the persons optional subelement `creditcard` as `<element name=''creditcard''`
`minOccurs=''0'' maxOccurs=''1''>...</element>`. The literal `unbounded` stands for an arbitrary element occurence. In XML Schema complex types are needed to define elements with attributes or content containing subelements. Any complex type declaration is enclosed by tags named `xsd:complexType`. Complex types can be named or unnamed. In the latter case these types are defined anonymously inside an element declaration. In the first case the complex type can be defined globally and can be reused for different element content types. The content models of complex types are mostly equivalent to those of the DTD. The `xsd:anyType` model does not restrict element content in any way. An `empty` element content model is rather difficult to define in XML Schema, an example will be given later in this section. Finally there are four most used content models, namely `xsd:sequence` for the sequence operator, `xsd:choice` for the choice operator, `xsd:all` allowing to order subelements arbitrarily and finally a mixed content model can be defined by setting the `mixed` attribute of the `xsd:complexType` element to true.
XML Schema also introduces some non-object-oriented concepts like restrictions. Restrictions in XML Schema subtract some facets of the regarding super type. More common to programmers might be inheritance by extension. In this case attributes as well as subelements can be added to super types. The XML Schema attributes `final`, `abstract` and `block` complete this part.
Reusability is achieved by the definition of so called `xsd:group` elements to reuse groups of elements and respectively `xsd:attributeGroup` elements to reuse groups of attributes.
Besides supporting the simple types `ID`, `IDREF` and `IDREFS` known from the DTD, XML Schema also introduces a concept to define and reference composite keys. The `xsd:unique` element allows to define uniqueness constraints. The `xsd:key` and `xsd:keyref` elements define primary and foreign keys analogously as known from databases.
As a conclusion a part of the auction DTD given in section 2.2.1 in example 2.2.2 is presented in XML Schema syntax in the example 2.2.3 .

Example 2.2.3

```
<xsd:schema ... >

 ...
<xsd:element name="open_auctions" type="open_auctionsType"/>

<xsd:complexType name="open_auctionsType">
        <xsd:sequence>
                <xsd:element name="open_auction"
                             type="open_auctionType"
                             minOccurs="0"
                             maxOccurs="unbounded"/>
         </xsd:sequence>
 </xsd:complexType>
<xsd:complexType name="open_auctionType">
        <xsd:sequence>
                <xsd:element name="initial" type="xsd:double"/>
                <xsd:element name="reserve" type="xsd:boolean"
                        minOccurs="0"/>
                <xsd:element name="bidder" type="bidderType"
                        minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="current" type="xsd:double"/>
                <xsd:element name="privacy" type="xsd:boolean"
                  minOccurs="0"/>
                <xsd:element name="itemref" type="itemrefType"/>
                <xsd:element name="seller" type="sellerType"/>
                <xsd:element name="annotation" type="annotationType"/>
                <xsd:element name="quantity" type"xsd:int"/>
                <xsd:element name="type" type="xsd:string"/>
                <xsd:element name="interval">
                        <xsd:complexType>
                         <xsd:sequence>
                          <xsd:element name="start" type="xsd:time"/>
                          <xsd:element name="end"   type="xsd:time"/>
                         </xsd:sequence>
                        </xsd:complexType>
                </xsd:element>
         </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"
                        use="required/>
 </xsd:complexType>

 ...

 </xsd:schema>
```

Example 2.2.3 focuses on the XML Schema definition of the `open_auctions` element containing a sequence of open auction elements. The element is declared to be of the named global complex type `open_auctionsType`. If neither the `minOccurs` nor `maxOccurs` attribute is explicitly specified, the element's cardinality is implicitly set to one. In other words, the default value for both cardinality attributes is one. In contrast to the named `open_auctionsType` complex type, the complex type of the element named `interval` is defined anonymously and locally inside the interval element's start and end tag. In this example all complex types rely on the sequence content model, e.g `open_auctionType` and the com-

plex type of the interval element. The cardinality of the bidder subelement is set to unbounded and the open auction's attribute `id` is defined to be of type `xsd:ID`. The `id` attribute is required for every open auction element by setting the attribute `use` to *required*.
So far the construction of XML documents and the definition of certain classes of XML documents have been discussed. Now it is going to be introduced how required information can be extracted from an XML document. Section 2.3 presents the most important aspects of the XML query language XPath, while section 2.4 goes on with the XML programming language XQuery. XQuery is based on XPath. Furthermore both query languages are recommended by the W3C.

2.3 XPath

XPath is an expression language proposed by the W3C [99] that allows to address and select values or nodes of an XML document. Originally XPath was designed as a part of another XML processing language. The result of an XPath expression may be a selection of nodes from the input document, or an atomic value, or more generally, any sequence allowed by the XML data model. The name *XPath* derives from its most significant feature, the path expression, which provides a facility of hierarchically addressing the nodes in an XML tree. As a first approximation an XPath expression can be compared to a directory path in a file system. XPath expressions are often used as attribute values in other XML vocabularies, e.g. defining composite keys in XML Schema.
In XPath an XML document is considered as a tree consisting of several different node types, which are listed in table 2.2.

Table 2.2: XPath node types

node type	**explanation**
root node	the whole XML document not to be confused with the document element
element node	represent an element
text node	represents textual content (character data) in an element or attribute
attribute node	represents an attribute
namespace node	represents a namespace
processing instruction node	represents a processing instruction
comment node	represents a comment

The root node as defined in XPath has got the document element, comment nodes and processing instruction nodes as direct subelements. Each XML document node in the XPath data model is characterized by the following set of features: node type (see table 2.2), name, e.g. an element's or attribute's name, content, e.g. for an attribute node it's the attribute's value, parent and child, e.g. only the root node and some element nodes might have child nodes. Attribute nodes especially are not child nodes of the regarding element node. Throughout this section the example XML document 2.3.1 is used to demonstrate the effect of different XPath expressions. This document is a slightly extended version of the XML document given in section 2.2 example 2.2.1.

Example 2.3.1

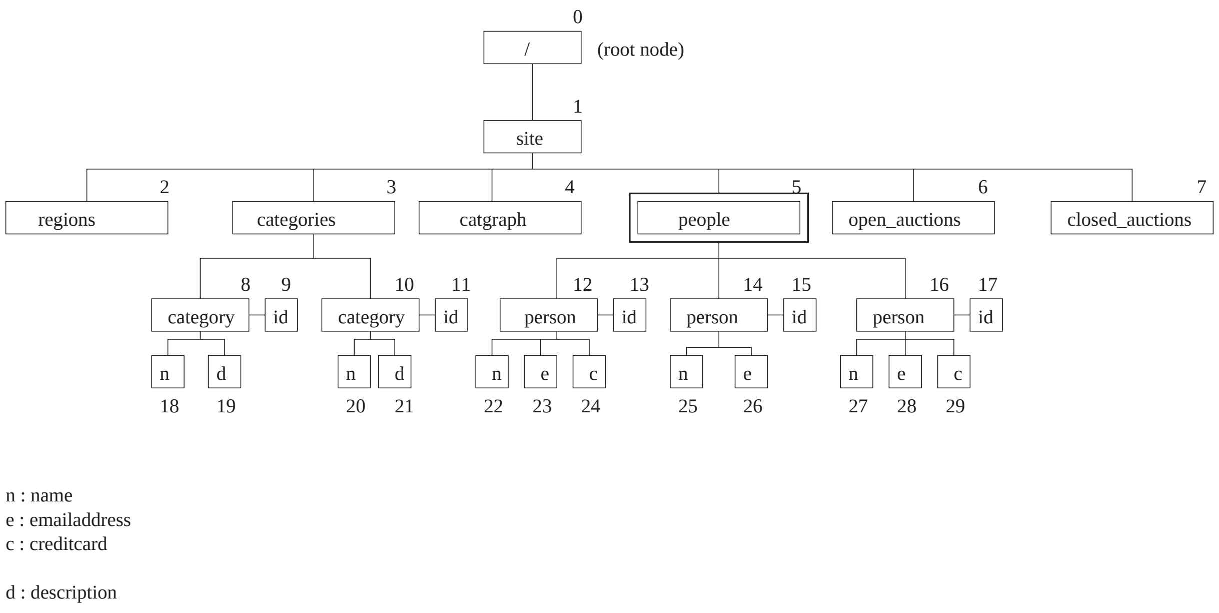
```
<site>
       <regions>
        <africa/>
        <asia/>
        <australia/>
        <europe/>
        <namerica/>
        <samerica/>
       </regions>
       <categories>
                <category id="c00001">
                        <name>Software</name>
                        <description>Business, education and games</description>
                </category>
                <category id="c00002">
                        <name>DVD</name>
                        <description>movies</description>
                </category>
       </categories>
       <catgraph/>
       <people>
                <person id="p00001">
                        <name>Mary Fernandez</name>
                        <emailaddress>fernandez@xquery.com</emailaddress>
                        <creditcard>123-2345-235</creditcard>
                </person>
                <person id="p00002">
                        <name>Jim Gray</name>
                        <emailaddress>jim@gray.com</emailaddress>
                </person>
                <person id="p00003">
                        <name>Ed Roman</name>
                        <emailaddress>ed@roman.com</emailaddress>
                        <creditcard>1246-334-845748</creditcard>
                </person>
       </people>
       <open_auctions/>
       <closed_auctions/>
</site>
```

To address the open auctions element that is a child element of `site` the XPath location path `/child::site/child::open_auctions` would apply. XPath location paths always result in a single node or value or in a list of them. Furthermore each location path consists of an arbitrary sequence of `location steps`. Location steps are separated from each other by slashes (/). Location paths may be relative or absolute. An absolute location path starts by addressing the document's root node, e.g. `/location_step_1/.../location_step_n`. Contrastly relative location paths are formulated in reference to a context node, e.g. `location_step_1/.../location_step_n`. Each location step consists of three constituents at maximum:

- axis specifier, selection by navigating the tree structure
- node test, selection by node type or name
- predicate, selection by testing specific node characteristics

These constituents are connected as follows `axis_specifier::node_test[predicate]`$*$ within each location step. The following table 2.3 lists and demonstrates the axis specifiers defined in XPath for the given example auction document . For this purpose the auction document of example 2.3.1 is represented as a tree in figure 2.1.
Text nodes are left out for readability. The context node is marked.

Figure 2.1: The example auction XML document represented as tree



Following the axis specifier in a location step, the node test filters the selected nodes of an XML document further. The existing kinds of node tests are listed in table 2.4. Finally predicates can be given to select specific nodes. Each location step can have an arbitrary number

Table 2.3: XPath axis specifiers

axis	**selected nodes** in general	**selected nodes** in example
self	context or current node	5
child	direct child elements	12,14,16
parent	direct ancestor	1
descendant	descendant nodes including children, grandchildren etc.	12,14,16,22,23,24,25,26,27,28,29
descendant-or-self	descendant nodes and current node	5,12,14,16,22,23,24,25,26,27,28,29
ancestor	ancestor nodes including parent, grandparents etc.	1,0
ancestor-or-self	ancestors and current node	5,1,0
following	all nodes following the current node in document order	6,7
following-sibling	all following nodes, which are siblings of the current node	6,7
preceding	all nodes preceding the current node in document order	2,3,8,9,10,11,18,19,20,21,4
preceding-sibling	all preceding nodes, which are siblings of the current node	2,3,4
attribute	the attributes of the current node	none
namespace	the current node's namespace	none

of predicates. The evaluation happens from left to right. The value of each predicate and node either results in true or false. XPath offers several abbreviations for the most frequently used axis specifiers or respectively location paths as listed in table 2.5. In addition XPath provides some functions working on node sets, single nodes or strings. `Number`,`boolean`,`string`, `node-set` and `object` are supported as data types. For example in case of node sets the functions `name(`*`node-set`*`)`, delivering the name of the first node, and `position()`, returning the position of the current node, are provided.

Example 2.3.2 The following XPath location path evaluated upon the context node in 2.1

child::person[@id="p_0001"]/child::name

returns the name element of the person with the id `p_00001`. As can be seen in the document representation in 2.3.1 this is <`name`>`Mary Fernandez`</`name`>.

Table 2.4: XPath node tests

node test	**description**
comment()	filters comment nodes
text()	filters text nodes
processing-instruction()	filters processing instructions nodes
node()	arbitrary nodes, e.g. elements, attributes.
$*$	the wildcard, depending on the corresponding axis specifier, e.g. `attribute` - filters all attribute nodes otherwise all element nodes
`a_name`	filters all nodes with the given name

Table 2.5: Abbreviations for axis specifiers and location steps

detailed syntax	**abbreviated syntax**
child::	(default, can be left out)
attribute::	@
/descendant-or-self::node()/	//
self::node()	.
parent::node()	..

Any XPath expression in this work is built up according to the grammar given in definition 2.3.1.

Definition 2.3.1 (XPath Grammar) The grammar for an XPath expression is defined as follows:

location_path	$\longrightarrow$	*relative_location_path*
relative_location_path	$\longrightarrow$	*step*|*relative_location_path* **’/’** *step*
step	$\longrightarrow$	*axis_specifier node_test predicate*∗
axis_specifier	$\longrightarrow$	*axis_name* **’::’**
axis_name	$\longrightarrow$	**ancestor** | **ancestor-or-self** | **attribute**
		| **child** | **descendant** | **descendant-or-self**
		| **following** | **following-sibling** | **parent**
		| **preceding** | **preceding-sibling** | **self**
node_test	$\longrightarrow$	*name_test*|*node_type* **’(’’)’**
predicate	$\longrightarrow$	**’[’** *predicate_expression ’]’*
predicate_expression	$\longrightarrow$	*expression*
name_test	$\longrightarrow$	**’∗’** | *name*
node_type	$\longrightarrow$	**comment** | **text** | **node**

The nonterminal *expression* denotes a boolean expression as it is known in many programming languages and therefore not defined in more detail. The nonterminal `name` represents an element name. □

XPath expressions working upon an XML document are formulated independently of its corresponding XML language description. In the next section XQuery, which is based on XPath, is introduced. In contrast to XPath, XQuery is a full, functional programming language.

2.4 XQuery

XQuery has gained working draft status in the W3C [105]. XQuery is a declarative query language for XML documents and XML data repositories and can be compared with the Structured Query Language (SQL) for relational databases. XQuery 1.0 uses a sub set of the node selection functionality included in XPath 2.0. Although XQuery is a powerful XML query language any update capability of XML documents or data is missing so far. There exist many proposals for updating XML. Details upon XML updates are given later in sections 2.9 and 3.3.2. The latter section describes the solution realized in **XOBE**$_{\text{DBPL}}$.
XQuery offers several kinds of expressions, namely variables, path expressions, constructors, embedded expressions and FLWOR expressions. Since path expressions are explained in section 2.3 and variables, constructors and embedded expressions are well known from other programming languages, this section will concentrate on FLWOR expressions. FLWOR expressions are analogously defined to SQL expressions. FLWOR expressions support iteration and binding of variables to intermediate results. This kind of expression is often useful for computing joins between two or more documents and for restructuring data. The name `FLWOR`, pronounced ”flower”, is suggested by the keywords `for`, `let`, `where`, `order by` and `return`. FLWOR expressions are constructed according to the grammar given in the definition 2.4.1.

Definition 2.4.1 (FLWOR grammar) FLWOR expressions in XQuery are built according to the following grammar:

flwor_expression	→	(*for_clause* | *let_clause*)+ *where_clause*? *order_by_clause*? **RETURN** *expr_single*
for_clause	→	**FOR ’$’** *var_name* *type_declaration*? *positional_var*? **IN** *expr_single* (**’,’’$’** *var_nametype_declaration*? *positional_var* **IN** *expr_single*)$*$
let_clause	→	**LET ’$’** *var_name* *type_declaration*? **’:=’** *expr_single* (**’,’’$’** *var_nametype_declaration*? **’:=’** *expr_single*)$*$
type_declaration	→	**AS** *sequence_type*
positional_var	→	**AT ’$’** *var_name*
where_clause	→	**WHERE** *expr_single*
order_by_clause	→	(**ORDER BY** | **STABLE ORDER BY**) *order_spec_list*
order_spec_list	→	*order_spec* (**’,’** *order_spec*)$*$
order_spec	→	*expr_single order_modifier*

The nonterminal *var_name* designates a variable name. The nonterminal *order_modifier* defines an order constraint upon the result tuples, e.g. ascending or descending. With the exception of the special type `void()`, a *sequence_type* consists of an item type that constrains the type of each item in the sequence, and a cardinality that constrains the number of items in the sequence. The nonterminal *expr_single* denotes an expression that does not contain a top-level comma operator. The comma operator offers one way to construct a sequence. Despite its name, an *expr_single* may evaluate to a sequence containing more than one item. □

The purpose of the `for` and `let` clauses in a FLWOR expression are to produce a tuple stream in which each tuple consists of one or more bound variables. Unlike a `for` clause, a `let` clause binds each variable to the result of its associated expression without iteration. The following simple example 2.4.1 demonstrates the different semantics. The optional `where` clause provides a filter for the tuples of variable bindings generated by the `for` and `let` clause. Only if the effective boolean value of the where expression is `true` the tuple is retained and its variable bindings are used in an execution of the `return` clause. With the help of the optional `order by` clause tuples in the tuple stream can be reordered. A `return` clause of a FLWOR expression is evaluated once for each tuple in the tuple stream. The results of these evaluations are concatenated, as if by a comma operator, to form the FLWOR expression's result.

Example 2.4.1 In this example a given node sequence consisting of the empty elements `one`, `two` and `three` is bound to a let variable and to a for variable. The characteristic results reflecting their different bindings are shown as well.

```
LET $s := (<one/>,<two/>,<three/>)   FOR $s IN (<one/>,<two/>,<three/>)
RETURN <out>{$}</out>                RETURN <out>{$}</out>
                          results in:
<out>                                <out><one/><out/>
<one/>                               <out><two/><out/>
<two/>                               <out><three/><out/>
<three/>
</out>
```

The next FLWOR example 2.4.2 could be formulated upon the example XML document given in section 2.3 example 2.2.1.

Example 2.4.2 The following FLWOR expression

```
FOR $p IN fn:doc("auction.xml")//person
WHERE fn:exists($p/creditcard)
RETURN <member> {$p/name}</member>
```

selects all persons of the auction site with a registered creditcard number. Each name of these selected persons is then returned enclosed between `member` tags. Applied to the example auction XML document in 2.2.1, the result is:

```
<member><name>Mary Fernandez</name></member>
<member><name>Ed Roman</name></member>
```

In contrast to XPath XQuery offers the possibility of statically type checking programs in reference to declared XML Schema or DTD instances. Static type checking of XQuery is described in section 3.2 in more detail.

2.5 The Document Object Model

The Document Object Model (DOM) gained W3C recommendation status on October 1st, 1998 and consists of several levels. The DOM levels denote certain versions. In this work it is focused on the core DOM level [97]. There exist two basic models for an XML programming language interface. On the one hand the Simple API for XML Parsing (SAX) [110] , which uses event driven parsing. A SAX parser is instantiated with an XML document and started. Then the XML document is read as a data stream from the beginning to its ending. During the processing the parser notifies all registered listeners via a uniform interface upon the occurence of XML document specific parts, e.g. the beginning of the document, the beginning and ending of an element. The advantages of the SAX model are a thin and easy to use API, as well as a fast XML document processing with minimum main memory usage. Since no parts of an XML document are kept in memory, this processing model of the SAX API may also be its disadvantage. By contrast the DOM provides a language-neutral interface that allows programs to dynamically access and update the content, structure and style of an XML document. The DOM is a model for tree-oriented XML document processing and representation. Any XML document is represented as a hierarchical structure consisting of nodes with exactly one root node. XML documents can be read in and offered to application programs for navigational processing including reading and writing operations. Furthermore DOM trees can be created by application programs and converted into XML documents. DOM trees can be transformed into other DOM trees. An important concept of the DOM is that everything is a node. Thus each more specific interface extends the most general `Node` interface. The `Node` interface provides methods to navigate and manipulate a DOM tree and is given in listing 2.2.

Listing 2.2: The DOM Node interface

```
public interface Node{
        ...
        //navigational methods
        public Node getParentNode();
        public NodeList getChildNodes();
        public Node getFirstChild();
        public Node getLastChild();
        public Node getPreviousSibling();
        public Node getNextSibling();
        //manipulating methods
        public Node insertBefore(Node newChild, Node refChild)...;
        public Node replaceChild(Node newChild, Node oldChild)...;
        public Node removeChild(Node oldChild)...;
        public Node appendChild(Node newChild)...;
        ...
}
```

The DOM's most important interfaces to represent an XML document as a tree are illustrated in figure 2.2. With the help of the interfaces listed in figure 2.2 the XML auction document in example 2.2.1 section 2.2 is represented as a DOM tree as shown in figure 2.3. In contrast to the SAX model, which uses stream based XML document processing, the DOM model represents an XML document as a tree in memory. With increasing size of XML documents the DOM model might become inefficient. In this case SAX processing may be more suitable for XML applications.

Figure 2.2: The DOM's interfaces for representing XML documents

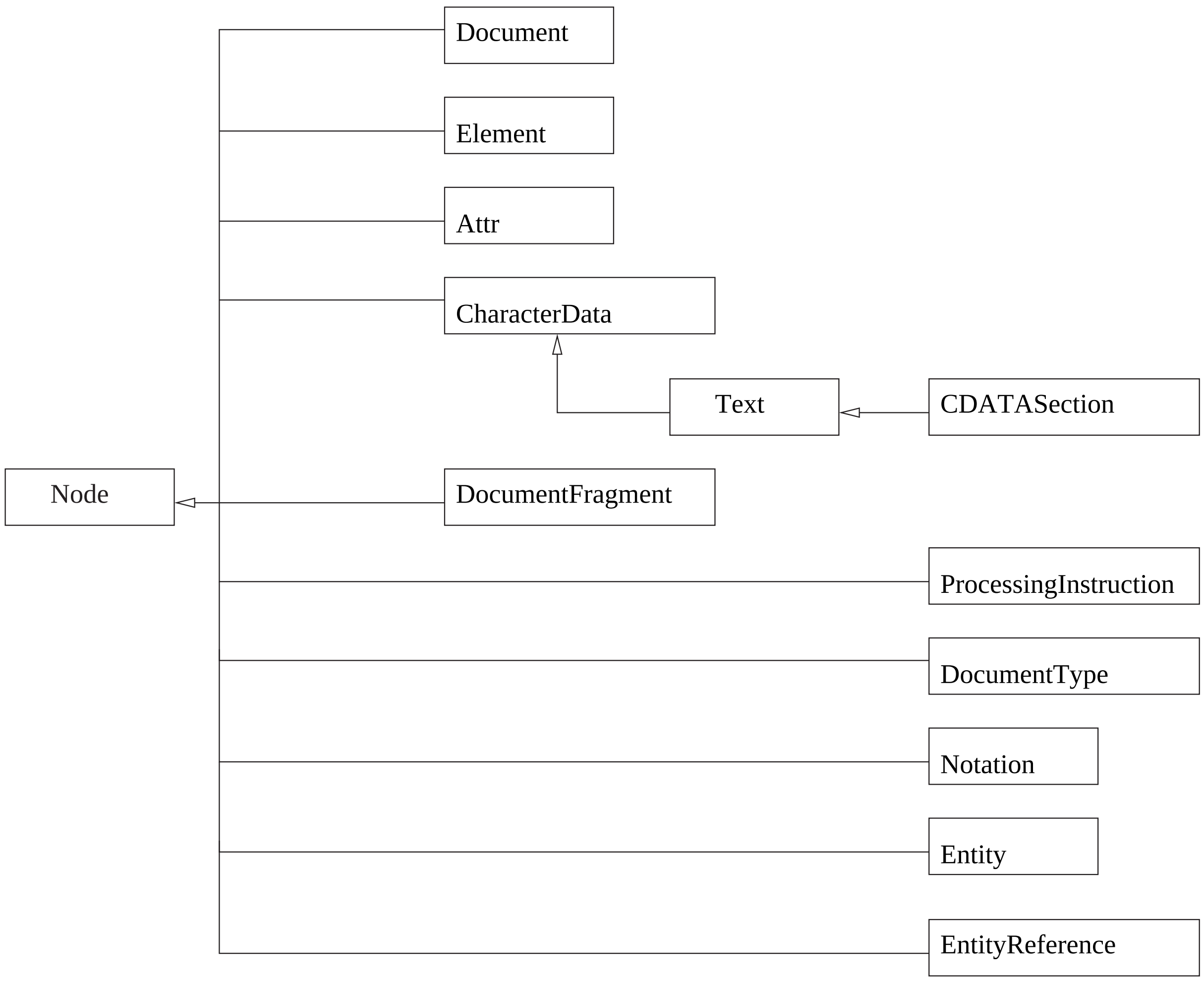
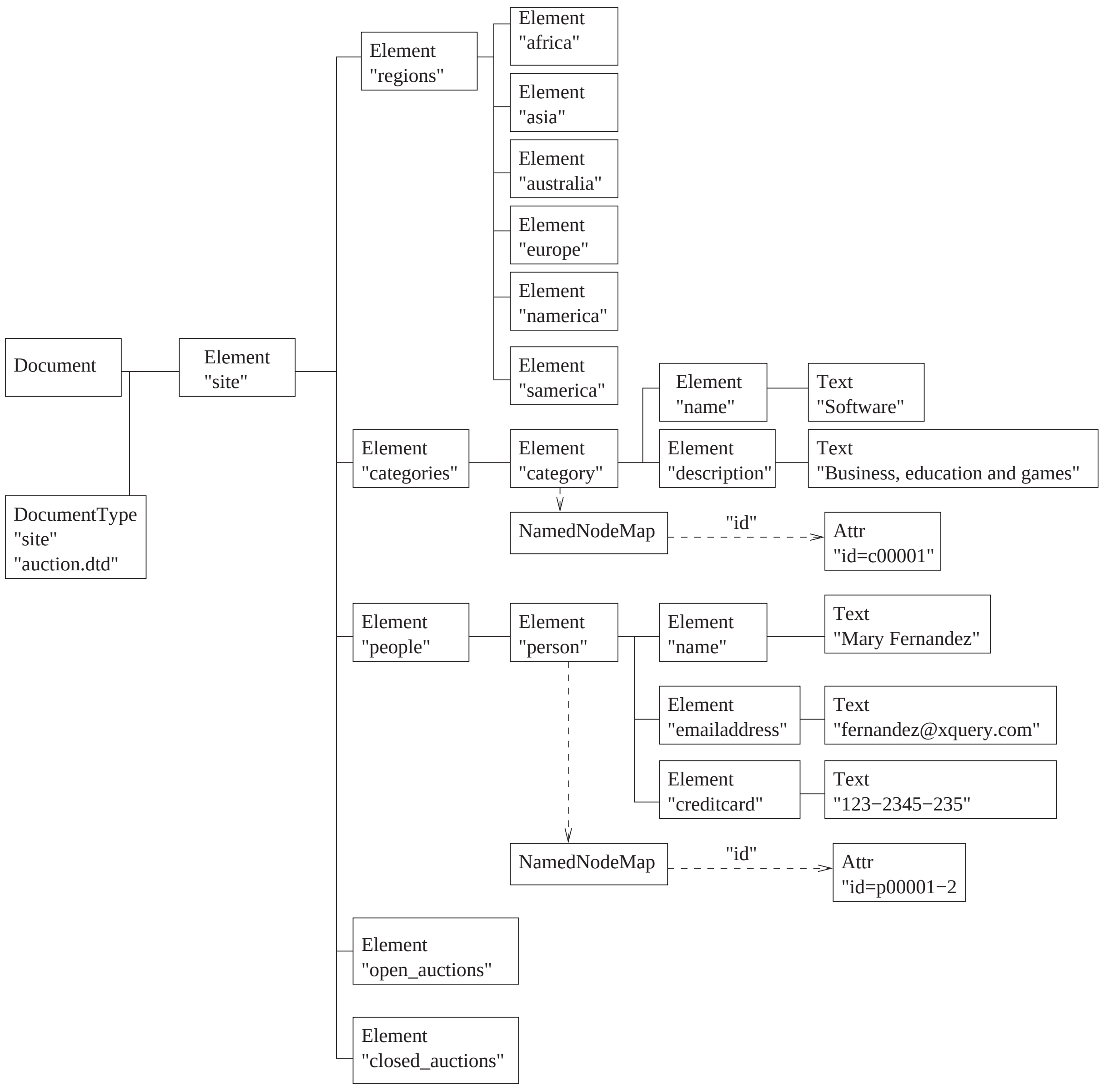


Figure 2.3: Example XML auction document represented as a DOM tree



2.6 XMark

Throughout this work the XMark XML language description [67] is used for examples and later on for experimental results. The aim of the XMark project is to provide a benchmark suite that allows users and developers to gain insights into the characteristics of their XML repositories. The XMark tool kit for evaluation consists of a workload specification, a scalable benchmark document and a comprehensive set of characteristic queries. The structure of the benchmark document, which is modeled after a database deployed by an internet auction site, is defined by the `auction` DTD already shown in example 2.2.2 in section 2.2.1. The main entities come in two groups: `person`, `open auction`, `closed auction`, `item` and `category` on the one side and `annotation` and `description` on the other side. While the first entity group belongs to a data centric XML model, the second group concentrates on natural language and contains document centric element structures.
The semantics of the mentioned entities is as follows. `Items` represent objects, which are on sale or have already been sold. Each item consists of a unique identifier and properties like payment, e.g. credit card, money order, a reference to the seller and a description. This information is all coded as element. In addition each item is assigned to a certain region, e.g. africa, asia, europe, by its parent. `Open auctions` are currently ongoing auctions. Properties of open auctions are for example the bid history along with references to the bidders and the seller, the current bid, the time interval within bids are accepted and a reference to the item being sold. Contrastly `closed auctions` are auctions that are already finished. Closed auctions contain references to the bidder and seller, which are both persons. Furthermore information like the reference to the regarding item, the final price, the number of sold items, the closing date and several annotations made during and after the bidding process are included. `Persons` are registered with their name, email address, credit card number and the possibly empty set of open auctions they are interested in among others. With the help of `categories` items are classified. The `category graph` links all categories in a network. Several subelements can occur optionally or arbitrarily often. By this, the XML document structure is not fully predictable. In the document centric part `annotation` and `description` elements are defined such that the length of strings and the internal structure of subelements varies enormously. Among others the markup contains itemized lists, keywords and even formatting instructions. Any XML document that is valid according to the auction DTD covers the full range of kinds of XML instances, from marked up data structures to traditional text. Nevertheless in this work the focus rather lies on the data centric parts of the auction DTD.
The XMark benchmark data generator `xmlgen` produces XML documents, which are valid according to the auction DTD. The sizes of generated XML documents can range from MBs to several GBytes. Even the generation process of large XML documents only needs low memory requirements. The number and type of elements are chosen according to a template and parameterized with certain probability distributions. The words of text paragraphs are taken from Shakespeare's plays. For simplicity the user can solely adjust the size of generated XML documents.

2.7 XML Updates

For the relational data model SQL is the standard query language including update operations. As mentioned before in sections 2.3 and 2.4 high level query languages like XQuery and XPath for XML data do not provide update operations explicitly. One might argue that XQuery might not need any update capability since it is possible to transform any given input document into an arbitrary output XML document. Even if by this procedure one might be able to simulate any update operation, the overhead is enormous. Low level updates for XML are provided by the DOM as described in section 2.5 . This section about XML updates is going to be twofold. On the one hand several characteristic high level approaches are introduced that make proposals for XML update syntax and semantics. On the other hand the problem of checking the validity of XML updates is discussed. An update upon an XML document is valid if the XML document always remains valid according to the declared XML schema. For the most significant syntactical and semantical approaches in the context of **XOBE**$_{\text{DBPL}}$ an insert and a delete example are shown. Both operations are performed upon the example XML auction document shown in 2.2.1 section 2.2. The first update inserts a new category into the auction's categories element. The second update deletes an existing person from within the people element.
The W3C itself made some unpublished proposals to extend XQuery to support updates in [94]. The new functional XML language introduced in [87] is based upon it and tightly coupled with XQuery itself. Validity checking is done during the update execution phase. The corresponding insert and delete example can be seen in 2.7.1.

Example 2.7.1 The example insert and delete operation in the XML language introduced in [87]:

```
insert
<category id="c_00002">
   <name>DVD</name>
   <description>movies</description>
</category>
as last into
document("xmark.xml")//categories;

delete
document("xmark.xml")//person[@id="p_00001"];
```

Another declarative XML update language is proposed in [50]. At the time of writing, another W3C working draft is available concerning XQuery update facility requirements [106]. The status of this document is work in progress. Another approach for XML updates in the context of the XQuery language that is often cited is [89]. Parts of the syntactical proposals made in [89] are adopted in this work, for details see section 3.3.2. The corresponding insert and update operations can be seen in the example 2.7.2.

Example 2.7.2 The example insert and delete operation in the extended XQuery update syntax made in [89]:

```
FOR $p IN document("auction.xml")//categories
UPDATE $p
INSERT
<category id="p_00002">
  <name>DVD</name>
  <description>movies</description>
</category>

FOR $p IN document("auction.xml")//people
UPDATE $p
DELETE $p/person[@id="p_00001"]
```

Basically [89] deals with the mapping of these update expressions to SQL. Any validation is omitted. Finally XML updates in XML syntax are proposed in the `XUpdate` project [109] . The corresponding insert and update operations can be seen in the example 2.7.3.

Example 2.7.3 The example insert and delete operation formulated in the XUpdate [109] XML language:

Listing 2.3: The example insert operation

```
<xupdate:modifications version="1.0"
  xmlns:xupdate="http://www.xmldb.org/xupdate">
        <xupdate:append select="document('auction.xml')//categories">
         <xupdate:element name="category">
           <xupdate:attribute name="id">c_00001</xupdate:attribute>
           <name>DVD</name>
           <description>movies</description>
          </xupdate:element>
        </xupdate:append>
</xupdate:modifications>
```

Listing 2.4: The example delete operation

```
<xupdate:modifications version="1.0"
  xmlns:xupdate="http://www.xmldb.org/xupdate">
        <xupdate:remove
          select="document('auction.xml')//person[@id='p_0001']"/>
</xupdate:modifications>
```

All approaches have got in common that they offer update operations like insert, delete, rename and replace. While the XML language described in [87] supports the validation of updates, the others do not. Several approaches in the context of XML updates solely concentrate on the validation aspect of updates. In [10] updates are checked before execution, but this happens at runtime. In [77] updates as defined in [89] are rewritten at compile time, but validity checking is also done at runtime. In [45],[63] and [111] validity is checked upon the resulting XML documents or respectively the resulting XML data. In contrast to all approaches mentioned in this section, **XOBE**$_{\text{DBPL}}$ integrates high level XML updates into the existing programming language Java and checks their structural validity at compile time. Details about updates and their validiation in **XOBE**$_{\text{DBPL}}$ are given in section 3.3.2.

2.8 Web Services

In this work web services denote SOAP based web services only. Related technologies are for example CORBA [59], DCOM [53] and Java RMI [81]. XML plays an important role in the context of web services. Furthermore the realization of **XOBE**$_{\text{DBPL}}$'s persistency layer is based on web services, for details see chapter 6. Web services can be described by two definitions. On the one hand web services provide a powerful architecture to dynamically integrate heterogeneous applications across networks using open internet technologies. On the other hand, from a more technical point of view, web services consist of the following elements:

XML formats, for details see section 2.2

HTTP the **H**yper **T**ext **T**ransfer **P**rotocol. HTTP is a protocol used to transmit web pages between internet servers and browsers [93].

SOAP a protocol that enables method invocations across single computers and systems [95, 96]. For example SOAP provides an XML schema describing the format of method invocations and parameter passings. Furthermore it defines the connection to HTTP or SMTP.

WSDL stands for **W**eb **S**ervice **D**escription **L**anguage and is another XML schema language describing the interface of a web service instance [103, 104]. Parts of the interface description refer for example to the provided methods and parameter types.

UDDI stands for **U**niversal **D**escription, **D**iscovery and **I**ntegration and is a registry service supporting the publication, search and binding of web service instances [57, 56].

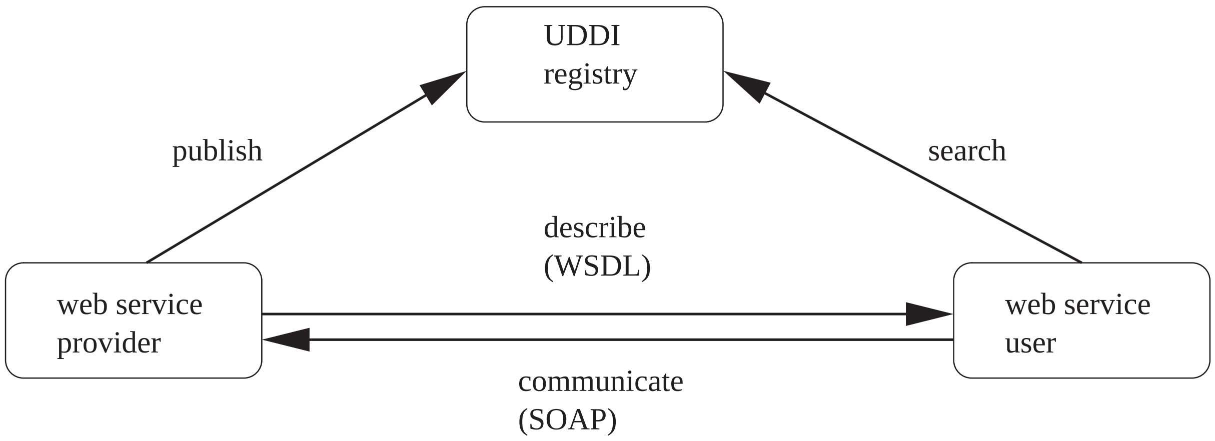
The following table 2.6 lists the web service constituents as levels with their appropriate internet protocols.

Table 2.6: Web service layers and protocols

web service layer	**protocol**
service discovery and publication	UDDI
service description	WSDL
XML based messaging and packaging	SOAP
internet transfer protocols	HTTP/SMTP
internet network protocol	TCP/IP

As can be derived from the first definition, web services enable applications written in an arbitrary language to communicate and exchange information. This is because every message is passed in XML format (SOAP) and the corresponding application interfaces are described in XML (WSDL) as well.
An XML message in SOAP format consists of an envelope, an optional header and a body containing the data. SOAP distinguishes between three different kinds of messages: requests, responses and fault descriptions. Passing complex objects or data via SOAP means nothing else than that these objects or data are serialized. SOAP messages are not written manually but rather generated automatically by software tools. Further details can be found in the W3C's SOAP specifications.
As mentioned before a WSDL description enables an application to integrate a web service just in time. For this purpose WSDL defines the following central parts of a web service, e.g. the interface of the provided methods, the data types of the submitted messages, including both responses and requests, information concerning the specific protocol, addresses of the specific services.
UDDI is itself a web service. Its functionality can be explained very well in the context of the three participants, which are in general the UDDI web service, the web service provider and the web service user. As can be seen in figure 2.4 UDDI offers two basic service methods. A web service provider can publish its web service instance via UDDI and a web service user can query UDDI for a desired web service instance. If UDDI finds a suitable web service instance, it passes a reference to the user. The user can then bind to the web service instance. The web service registry plays an important role in UDDI and works analogously to the yellow, white and green pages.

Figure 2.4: UDDI triangle



2.9 XML in Programming Languages

The most elementary way to deal with XML fragments is to use ordinary strings without any structure, e.g. Java Servlets [108]. Java Server Pages [64] provide an improvement by allowing to switch between XML parts and Java using special markings. All these approaches share the disadvantage that not even well-formedness is checked at compile time.
Low-level-binding approaches like the DOM or Java DOM (JDOM) [40] provide classes for nodes in an XML document, thus allowing to access and manipulate arbitrary XML fragments by object-oriented programming. Low-level bindings ensure well-formedness of dynamically generated documents at compile time, but defer validation until runtime. The ECMAScript language extension for XML [19] integrates the construction and manipulation based on a tree-like navigation of XML objects into ECMAScript [18], but validation against an XML language description is not supported.
High-level bindings [11] like Sun's JAXB, Microsoft's .Net Framework, Exolab's Castor , Delphi's Data Binding Wizard and Oracle's XML Class Generator [86, 54, 21, 9, 60] assume that all processed documents follow a given schema. This description is used to map the document structure onto language types or classes reproducing directly the semantics intended by the schema. Apache's XML Beans [4] offer a DOM-like tree navigation API to underlying XML documents or additionally generate a set of Java classes and interfaces corresponding to XML schemas, if XML schemas are compiled. A third API supports XQuery. Validity at compile time is only supported up to a limited extent depending on the selected language mapping.
Recently, the aspect of guaranteeing the validity of XML structures at compile time gained some interest. The XDuce language [34], [33] is a special functional language developed as an XML processing language. XML elements are created by specific constructors. The content can be accessed through pattern matching. XDuce supports type inference and performs a subtyping

analysis to ensure validity of XML expressions at compile time. The subtyping algorithm is implemented on the basis of a regular tree automaton. The Xtatic project [26] is the successor of XDuce. The main purpose of Xtatic is to couple the concepts of XDuce with the object-oriented programming language C#. Xtatic has similar goals like **XOBE**$_{\mathrm{DBPL}}$. However, Xtatic is still in an early stage and, in contrast to **XOBE**$_{\mathrm{DBPL}}$, Xtatic does not support XPath. BigWig [12] is a special programming language for developing interactive web services. JWig [17] is the successor of BigWig integrating the XML-specific parts of BigWig into Java. JWig is quite close to **XOBE**$_{\mathrm{DBPL}}$. The main difference in JWig is that there is only one XML type. Typed XML document templates with gaps are introduced. In order to generate XML documents dynamically, gaps can be substituted at runtime by other templates or strings. For these templates JWig validates all possibly dynamically computed documents according to a given abstract DTD by data flow analysis. This data flow analysis is rather time consuming. In the Xact project [46] JWig's validation algorithm is extended to the problem of static analysis of XML transformations in Java. Like in **XOBE**$_{\mathrm{DBPL}}$, XPath is used for expressing XML transformations. XJ [31] , a new project by IBM research, integrates XML into Java concentrating on traversing XML structures by using XPath. The distinguishing characteristic of XJ is its support for in place updates. Xen [52] is an integration of XML into popular object-oriented programming languages such as C# or Java currently under development at Microsoft Company. Xen uses XML constructors similar to **XOBE**$_{\mathrm{DBPL}}$'s XML object constructors. XL [23] is a special programming language for the implementation of web services. It provides high-level and declarative constructs adopting XQuery. Additionally imperative language statements are introduced making XL a combination of an imperative and a declarative programming language.

2.10 Persistent XML

For storing XML structures persistently several approaches exist. One approach is to map XML structures to relational tuples or blobs and store them in an object-relational database. For example Oracle [61, 55] and DB2 [35, 16] belong to this group of systems. HyperJAXB [38] is an add-on for Sun's reference implementation of JAXB (Java Architecture for XML Binding) [86]. It provides JAXB objects with a relational persistence layer using Hibernate [32] . Hibernate is an object-relational mapping solution for Java environments. Although Oracle extends SQL to provide support for XPath, application programs work with the provided structures by using conventional tools like Java Database Connectivity (JDBC). JDBC does not provide any facilities to guarantee that the application program works only with valid XML structures. This means that either there is no validation or validation has to be performed on demand at runtime for the whole document. In contrast to **XOBE**$_{\mathrm{DBPL}}$ this means that validity cannot be guaranteed at each step of the XML generation and manipulation. The same holds for the case when the application program does not work with tables, but uses exported XML structures. In this case, conventional tools like DOM or SAX are used. DOM and SAX do not allow to check the validity of XML structures at compile time, thus runtime validation is necessary.

Another group of approaches is known as the group of native XML database systems. Prominent examples are Tamino [69], Xindice [4] , Infonyte DB [36] and Natix [22]. These systems do not use relations or objects, instead, they store XML structures directly. Most systems use application programming interfaces that base on DOM or SAX, e.g. there is no validity checking at compile time. Moreover, DOM and SAX are rather low level interfaces requiring a lot of programming. An exception is Xindice [4] which supports XUpdate [109]. To the best of our knowledge, there is no system with an application programming interface that allows checking the validity of XML structures statically at compile time of the application program.

2.11 Database Programming Languages

A number of attempts have been made to construct programming languages with completely integrated database management systems. These languages are called `database programming languages`. Important topics of any useful and consistent database programming language are persistence, inheritance, polymorphism, modules, implementation, transaction handling and concurrency.
In programming languages objects normally have a well-defined lifetime. A variable declared in a block or method will persist during the activation of that code segment and thereafter be inaccessible. If an object is created as part of another data structure, its persistence, from the programmer's point of view, is the duration for which it remains accessible. In contrast to these transient objects, the treatment of data required to last longer than a single program execution is less uniform. Especially the set of types that may persist is often a small subset of, or even different from, the types available for transient objects, e.g object-oriented programming languages and relational databases.
A first definition of a database programming language is given in [14] and shown in 2.11.1.

Definition 2.11.1 In addition to data, data flow and concurrency abstractions provided by third generation programming languages (3GL), database programming languages offer transparent persistency. In contrast to fourth generation programming languages (4GL) persistency is orthogonally integrated into the language. □

The complete set of requirements for a database programming language is not clearly defined. Nevertheless a basis can be found in [6]. The following list of requirements given in 2.11.2 is slightly revised and extended.

Definition 2.11.2 The majority of features commonly used in programming languages might be inherited except some important differences mentioned below.

data model integration

type checking Besides dynamic binding and incremental type checking, static and strong type checking is particularly important due to the possible long lifetime of data.

type independent persistency The type system must be entirely consistent for persistent as well as for transient data. In particularly persistence must be independent of type.

consistent naming A consistent naming system for objects, program components and types is required independent of longevity.

polymorphism and inheritance The type system must support polymorphism and inheritance.

orthogonal persistence Persistence must be orthogonal to type, and it should be possible to write programs without taking into account the longevity of data.

transparent persistency The programmer should not have to organize placement or movement of data but should be provided with operations to copy, protect and secure data.

transparent transactions Transparent notations and support for transactions should be provided.

transparent concurrency hides that resources might be used by several users simultaneously. Transparent concurrency should be supported.

transparent distribution hides the location of ressources. Transparent distribution mechanisms for data and programs should be provided.

□

Traditionally the database and programming language communities have taken different approaches to concurrency control. In programming languages, concurrency control is based upon the concept of the co-ordination of a set of co-operating processes by synchronisation. Language constructs such as for example semaphores, monitors and mutual exclusion have been provided to support this concept. Contrastly, in databases, concurrency is viewed as a system efficiency activity allowing parallel execution and parallel access to the data. Nevertheless each database process may be aborted to keep the illusion of non interference. In databases the key concept is serializability [20] leading to the notion of atomic transactions [20, 47]. Atomic transactions are realized by locking [20] or optimistic concurrency control methods [47]. Transactions, concurrency and their support in **XOBE**$_{\text{DBPL}}$ are discussed in more detail in section 5. As can be derived from above, the concept of a database programming language includes especially transparent and type independent persistency as well as a transaction concept. Persistency in **XOBE**$_{\text{DBPL}}$ is introduced in detail in chapter 4. In this work we will concentrate on object-oriented programming languages that are extended with the requirements listed in definition 2.11.2. A good survey on existing database programming languages can be found in [6] and [49]. These works concentrate on comparing the type systems and persistency mechanisms. Design principles for persistent object systems and a motivation for orthogonal persistency is reviewed in [7]. Among the approaches which integrate the relational model is DBPL [68] and its successor project Tycoon [51] . In the DBPL project Modula-2 is extended by parametric bulk types for relations. Its successor project Tycoon is based on an object-oriented programming language. To the best of our knowledge there is no database programming language which integrates the XML data model into an object oriented language like Java. Section 2.12 introduces the concept of distributed and persistent objects along with some selected approaches.

2.12 Distributed and Persistent Objects

Distributed Objects. CORBA [59] is an OMG specification and basis to develop distributed applications, which can exchange messages independant of hardware, operating system, programming languages. Besides incompatibility, it lacks of an own object oriented programming language interface. In contrast to CORBA Java Remote Method Invocation (Java RMI) [81] was developed for a heterogeneous Java environment. It does not need a new interface description language like CORBA since Java already includes interfaces. CORBA as well as RMI do not support persistency aspects explicitly, they both realize distributed object architectures.

Persistent Objects. JDO [80] and Java Beans [85] provide a framework for persistent objects in Java. More recent versions include web service capabilities and try to reduce programming difficulties and the exception overhead. JDO supports client-side application development for persistent Java objects. Once again programmers are forced to write Java classes that are limited. Persistence capable classes must be enhanced and persistent object manipulation implies connecting to a persistence manager explicitely. Mapping of persistent classes to relational databases is defined in XML files analogous to the approaches mentioned so far.

Hibernate [32] is an object-oriented mapping program in Java. Hibernate provides a three layered model to map Java bean classes to relational databases. The first step of this process is to write a Java class keeping all bean rules. This implicates for example that the class must be directly derived from `java.lang.Object`. Moreover a standard constructor and getter and setter methods for each persistent attribute must be available. In a second step the programmer has to formulate mapping rules with the help of an XML file. The class's attributes are mapped to columns of the database in example. Tools like XDoclet [76] generate these files for simple cases. The third step consists of the generation process of data access classes. These classes then provide operations to create, read, actualize and delete persistent objects.

Distributed and Persistent Objects. The Java Spaces project, which has been developed in context of the Jini network technology [82], provides a distributed storage for Java objects, in particular for message exchange. The idea is based on Tuple Spaces [27]. Accordingly, a distributed storage offers operations to read, write and store an object. Additionally, the take operation loads and deletes an object from the storage. Java Spaces stores objects persistently and keeps their identity. An important limitation is that an object can only be retrieved by its object id. All objects of the storage have to be Java serializable.

EJB [79] supports server-side application development for distributed and persistent Java objects . In EJB 2.x there are three bean types. Session beans represent behaviour and functionality. Message beans are used in message-oriented systems and entity beans represent data. In EJB 3.x entity beans are replaced by plain old Java bean classes. EJB 2.x seems to have become too complex for most developers. It is interesting to notice that EJB 3.x tries to integrate the most successful parts of JDO and Hibernate as well. Nevertheless EJB is not developed as a database programming language. Therefore, the EJB programmer is forced to make objects persistent and to connect to a central persistency instance explicitly by writing specific code for this task. The most important limitation regarding this work is that in the case of EJB 2.x entity beans (persistence capable classes) do not support inheritance and in the case of EJB 3.x inheritance for attributes of persistent classes is not fully supported. If an attribute of such a class is defined to have an interface type it can only be mapped to a blob in the database.

Chapter 3

XML Integration

3.1 XML Objects

In this section the syntax and semantics of XML objects already defined in **XOBE** are briefly summarized in an informal manner. **XOBE**$_{\text{DBPL}}$ is based upon **XOBE** . A more detailed introduction can be found in [43] and [42]. **XOBE** extends the object-oriented programming language Java by language constructs to process XML fragments in particular XML documents. XPath expressions are used for traversing XML objects.
In **XOBE** XML fragments, e.g. trees corresponding to a given schema, are represented by XML objects. Therefore, XML objects are first-class data values that may be used like any other data value in Java. DTDs as well as XML Schemas are supported schema languages in **XOBE**. The declared schema is used to type different XML objects.
Listing 3.1 introduces the most important features of **XOBE**. Again the XML objects are typed with the help of the auction DTD, for details see the sections 2.2.1 and 2.6.

Listing 3.1: **XOBE** method `createPerson`

```
person createPerson(String p_name, String p_email, String p_id){
      person p;
      xml<person*> plist = $auctionSite//person[/@id={p_id}]$;
      if(plist.size()>0)return null; //person already exists

      //new person is created
      p = <person @id={p_id}>
                <name>{p_name}</name>
                <emailaddress>{p_email}</emailaddress>
           </person>;

      return p;
}
```

In line 7 an XML object of type `person` is created with a so-called XML constructor. The type declaration `person` of variable `p` in line 2 is an abbreviated version. In general the type declaration of an XML variable starts with the keyword `xml` followed by square brackets. Within the brackets an arbitrary regular expression is used to type the XML variable. XML objects can be accessed by an XPath expression. In line 3 a previously declared XML object `auctionSite` of type `site` is searched to determine whether the new person already exists. Only if the result is negative, the new XML object person is created and returned. After this short introduction into **XOBE**, we come to the most important aspect in **XOBE** as well as **XOBE**$_{\text{DBPL}}$ regarding the integration of XML. This aspect is static type checking for XML objects, queries and updates and is the topic of section 3.2.

3.2 Static Type Checking

To explain how **XOBE**$_{\text{DBPL}}$ statically checks the validity of XML objects, queries and updates, we need to explain its type system first.
XOBE$_{\text{DBPL}}$'s type system is an extension of the **XOBE** type system, which is described in detail in [42]. It is built on top of the standard Java type system. Checking type correctness of a **XOBE**$_{\text{DBPL}}$ program consists of three parts .

Formalization translates the declared schema description into a more formal representation.

Type inference is used to determine XML types and differs for XML constructors, query and update expressions in a **XOBE**$_{\text{DBPL}}$ program.

Subtype algorithm checks if the inferred XML types are valid according to their formalized schema description. The explanation of the subtype algorithm is not part of this work, but is in fact part of the work in [42, 43].

In **XOBE**$_{\text{DBPL}}$ we formalize and represent types as regular hedge expressions representing regular hedge languages [13]. Consequently a schema is formalized and represented internally by a regular hedge grammar.
It is important to notice that any resulting regular hedge grammar in **XOBE**$_{\text{DBPL}}$ solely covers structural constraints on XML types. In particular this means that the value-based constraints implied for example by IDs or IDREFs in DTDs as well as ID/IDREFs and key/keyrefs constraints in XML Schemas are not preserved by the formalization process. Since **XOBE**$_{\text{DBPL}}$ intents to check static validity, this is not a limitation. In general such value-based constraints cannot be checked at compile time at all.
Regular hedge expressions and regular hedge grammars are used in **XOBE**$_{\text{DBPL}}$ like in **XOBE** [43, 42]. For readability, the corresponding definitions are repeated here.

Definition 3.2.1 A regular hedge grammar is defined by $G = (T, N, s, P)$ with a set $T = B \cup E$

and $B \cap E = \emptyset$ of terminal symbols, consisting of simple type names B and a set E of element names (tags), a set N of nonterminal symbols (names of groups and complex types), a start expression s and a set P of rules or productions of the form $n \rightarrow r$ with $n \in N$ and r is a regular hedge expression over $T \cup N$ and $T \cap N = \emptyset$. We restrict r to be recursive in tail position only. This ensures regularity. $\Box$

The definition of regular hedge expressions, referred to in short as regular expressions, is similar to the notation used in [105].

Definition 3.2.2 Given a set of terminal symbols $T = B \cup E$ and a set N of nonterminal symbols, the set Reg of regular hedge expressions is defined recursively as follows:

$\emptyset \in Reg$	the empty set,
$\epsilon \in Reg$	the empty hedge,
$b \in Reg$	the simple types,
$n \in Reg$	the complex types,
$e[r] \in Reg$	the elements,
$r \mid s \in Reg$	the regular union operation,
$r; s \in Reg$	the concatenation operation and
$r* \in Reg$	the Kleene star operation.

for all $b \in B, n \in N, e \in E, r, s \in Reg$. $\Box$

Attributes are treated as element types with simple content having a name prepended by '@'. Disorder constraints of attributes can be simulated by generating a choice type of all possible sequences. However, this is implemented more efficiently in **XOBE**$_{\text{DBPL}}$.
The formalization step applied to the XMark auction DTD given in 2.2.2 yields the following regular hedge grammar shown in the example 3.2.1.

Example 3.2.1 As explained above only structural constraints of schemas are formalized. Element names and simple types are **boldfaced**, nonterminal symbols are *italic*. An '@' marks an attribute. The start expression s is *auction_dtd*.

auction_dtd	$\rightarrow$	**document**[*site*]
site	$\rightarrow$	**site**[*regions;categories;catgraph;people;open_auctions;closed_auctions*]
people	$\rightarrow$	**people**[(*person*)$*$]
person	$\rightarrow$	**person**[**@id**[string];*name*;*emailaddress*;($\epsilon \mid$ *phone*);
		($\epsilon \mid$ *address*);($\epsilon \mid$ *homepage*);($\epsilon \mid$ *creditcard*);($\epsilon \mid$ *profile*);($\epsilon \mid$ *watches*)]
name	$\rightarrow$	**name**[string]
emailaddress	$\rightarrow$	**emailaddress**[string]
phone	$\rightarrow$	**phone**[string]
...	$\rightarrow$	...
open_aucions	$\rightarrow$	**open_auctions**[(*open_auction*)$*$]
open_auction	$\rightarrow$	**open_auction**[**@id**[string];*initial*;($\epsilon \mid$ *reserve*);
		(*bidder*)$*$;*current*;($\epsilon \mid$ *privacy*);
		itemref;*seller*;*annotation*;
		quantity;*type*;*interval*]
initial	$\rightarrow$	**initial**[string]
reserve	$\rightarrow$	**reserve**[string]
bidder	$\rightarrow$	**bidder**[*date*;*time*;*personref*;*increase*]
current	$\rightarrow$	**current**[string]
privacy	$\rightarrow$	**privacy**[string]
itemref	$\rightarrow$	**itemref**[**@item**[string]]
seller	$\rightarrow$	**seller**[**@person**[string]]
annotation	$\rightarrow$	**annotation**[string]
quantity	$\rightarrow$	**quantity**[string]
type	$\rightarrow$	**type**[string]
interval	$\rightarrow$	**interval**[*start*;*end*]
start	$\rightarrow$	**start**[string]
end	$\rightarrow$	**end**[string]
date	$\rightarrow$	**date**[string]
time	$\rightarrow$	**time**[string]
personref	$\rightarrow$	**personref**[**@person**[string]]
increase	$\rightarrow$	**increase**[string]
...	$\rightarrow$	...

The regular expression type of the start expression s is implicitly defined by the schema. The *auction_dtd* type represents the condition that each element, which is defined as a direct child of the schema root element, can be used as a valid root element within a corresponding schema instance (document). The XMark auction schema defines a single document root element $site$, therefore *auction_dtd* is derived to **document**[*site*]. If more root elements are defined, the content type becomes a choice type, e.g. **document**[*root_type_1*$\mid$...$\mid$...*root_type_n*].

In a next step after the formalization process XML types in a **XOBE**$_{\text{DBPL}}$ program are inferred.
In **XOBE**$_{\text{DBPL}}$ all variables have to be declared, therefore type inference of variables is simple. In listing 3.1 variable `p` is declared of type *person* as well as the result type of the method `createPerson`. The variables `p_email`, `p_name` and `p_id` are declared to be of type *string*. Based on variable and result types, types of whole XML constructors on the right hand side of an assignment can be inferred quite intuitively. In the XML constructor of listing 3.1 line 7 it is:
person[**@id**;**name**[string];**emailaddress**[string]]
After inferring the types of the left and right hand side, the **XOBE**$_{\text{DBPL}}$ type system checks if the type on the right hand side is a subtype of the type on the left hand side. In this case **XOBE**$_{\text{DBPL}}$ has to check if:
person[**@id**;**name**[string];**emailaddress**[string]]
is a valid subtype of *person* according to the formalized XMark auction DTD given in example 3.2.1.

3.2.1 XPath Expressions

As introduced by an example in the last section, type inference for XML object constructors in **XOBE**$_{\text{DBPL}}$ can be understood quite intuitively. XPath expressions in **XOBE**$_{\text{DBPL}}$ are constructed according to the grammar given in definition 3.2.3.

Definition 3.2.3 In **XOBE**$_{\text{DBPL}}$ an XPath expression is always formulated upon a predefined context variable, except if the path expression is part of a predicate. In the latter case the context variable can be left out, since it is implicitly defined to be the selected node list of the step the predicate belongs to.

expression	→	... | *xpath_expression*
xpath_expression	→	*name*? **’/’** *location_path*

The nonterminal *location_path* is defined according to the W3C’s XPath grammar already defined in 2.3.1. □

Type inference for XPath expressions in **XOBE**$_{\text{DBPL}}$ always starts with the context variable and proceeds by inferring recursively the types of selected nodes by each step. This is slightly different for XPath expressions within a predicate or update operation. In this case the context variable is implicitly given. It is the currently selected node list of the step the predicate belongs to or the target variable of the update operation. The final type is inferred after the last XPath expression step is handled. An example of the XPath type inference rules applied to the expression in listing 3.1 line 3 is given in example 3.2.2.

Example 3.2.2 The type of the XPath expression in listing 3.1 line 3:
auctionSite//person[/@id={p_id}]
is inferred as follows. The context variable is declared to be of type *site*. In this case the location path consists of one further step selecting all descendant elements with tag name **person**. Thus the result type is inferred as *person∗*, which is of course not the best type. Since **XOBE**$_{\text{DBPL}}$'s type system is limited to structural validity, the predicate filtering at most one person element by its unique id cannot be taken into account.

Although XPath type inference rules can be found in [42] in detail, the most important ones are repeated here, because the type inference rules for complex queries and updates described in section 3.3.1 and 3.3.2 are based on them.
In case of XPath expressions, types possess the specific characteristic that they are always of the form ($e_1[r_1] \mid e_2[r_2] \mid ... \mid e_n[r_n]$), with $e_i \in E$ and $r_i \in Reg$. This is because by definition XPath expressions return a list or rather a set of nodes. In the following the regular union of element types within a Kleene star operator is also called *XPath type*.
To define the type inference rules for XPath expressions in **XOBE**$_{\text{DBPL}}$ several auxiliary functions have to be defined. The auxiliary function `nodeTest` 3.2.4 is used to infer the XPath type of a node test. Furthermore we need functions for every XPath axis, e.g. `self` 3.2.5, `child` 3.2.6, `descendant` 3.2.7 and `parent` 3.2.8.

Definition 3.2.4 The function `nodeTest` : $E \times Reg \rightarrow Reg$ yields for a given element name $e \in E$ those types from an XPath type $r \in Reg$ that have got the element name e. The function is defined recursively:

$$\begin{array}{lcl} \text{nodeTest(e,}\emptyset\text{)} & = & \emptyset \\ \text{nodeTest(e,f[r])} & = & \begin{cases} e[r] & \text{if} \quad e = f \\ \emptyset & \text{else} \end{cases} \\ \text{nodeTest(e,r} \mid \text{s)} & = & \text{nodeTest(e,r)} \mid \text{nodeTest(e,s)} \end{array}$$

□

Definition 3.2.5 The function `self` : $Reg \rightarrow Reg$ yields the XPath type of a regular expression $r \in Reg$ and is defined by a two parameter auxiliary function `self`:

$$\text{self(r)} \quad = \quad \text{self(r,}\{\}\text{)}$$

The two parameter auxiliary function `self` : $Reg \times P(N) \rightarrow Reg$ is recursively defined as:

$$
\begin{array}{lcl}
\text{self}(\emptyset,N_v) & = & \emptyset \\
\text{self}(\epsilon,N_v) & = & \emptyset \\
\text{self}(\text{b},N_v) & = & \emptyset \\
\text{self}(\text{n},N_v) & = & \begin{cases} \emptyset & \text{if} \quad n \in N_v \\ \text{self}(r, N_v \cup \{n\})\text{with} \quad n \to r \in P & \text{else} \end{cases} \\
\text{self}(\text{e[r]},N_v) & = & \begin{cases} e[r] & \text{if} \quad e \neq @f \\ \emptyset & \text{else} \end{cases} \\
\text{self}(\text{r|s},N_v) & = & \text{self}(\text{r},N_v) \mid \text{self}(\text{s},N_v) \\
\text{self}(\text{r;s},N_v) & = & \text{self}(\text{r},N_v) \mid \text{self}(\text{s},N_v) \\
\text{self}(\text{r}*,N_v) & = & \text{self}(\text{r},N_v)
\end{array}
$$

with $b \in B, n \in N, e, @f \in E, r, s \in Reg$ and $N_v \subset N$. □

The auxiliary function `self` derives the element types of a given regular expression. These element types are then combined by the regular union operator to form the required XPath type. The set N_v collects the names of nonterminals that have already been processed. Only if a non processed nonterminal is discovered the production rules of the formalized XML schema grammar are used to apply the `self` function recursively. The set N_v helps to avoid endless loops. Attribute types, basic types as well as the regular concatenation and Kleene star operator remain unconsidered.

Definition 3.2.6 The function `child` : $Reg \to Reg$ yields the XPath types of the child nodes of an XPath type $r \in Reg$ and is defined recursively:

$$
\begin{array}{lcl}
\text{child}(\emptyset) & = & \emptyset \\
\text{child(e[r])} & = & \text{self(r)} \\
\text{child(r|s)} & = & \text{child(r)} \mid \text{child(s)}
\end{array}
$$

□

Definition 3.2.7 The function `descendant` : $Reg \to$ ***Reg*** yields the XPath types of the descendants of a regular expression $r \in Reg$ and is defined recursively:

$$
\begin{array}{lcl}
\text{descendant}(\emptyset) & = & \emptyset \\
\text{descendant(e[r])} & = & \text{descendantOrSelf(r,}\{\}) \\
\text{descendant(r|s)} & = & \text{descendant(r)} \mid \text{descendant(s)}
\end{array}
$$

for all $e \in E$ and $r, s \in Reg$. The auxiliary function `descendantOrSelf` : $Reg \times P(N) \to Reg$ is defined analogously to the two parameter auxiliary function `self` except:

$$
\text{descendantOrSelf(e[r]},N_v) \quad = \quad \begin{cases} \text{descendantOrSelf}(r, N_v) \mid e[r] & \text{if} \quad e \neq @f \\ \text{descendantOrSelf}(r, N_v) & \text{else} \end{cases}
$$

□

Definition 3.2.8 The function `parent` : $Reg \rightarrow Reg$ yields the XPath types of the parent nodes of a regular expression $r \in Reg$ and is defined with the help of the four parameter function `parent`:

$$\text{parent(t)} = |_{n_i \in N} \text{ parent(t,}N_s\text{,}\emptyset\text{,}r_i\text{) with } n_i \rightarrow r_i \in P$$

and $N_s = \{n \mid \text{t is subtype of self(r) with } n \rightarrow r \in P\}$.
The auxiliary funtion `parent` : $Reg \times P(N) \times Reg \times Reg \rightarrow Reg$ is recursively defined as follows:

$$\begin{array}{lcl}
\text{parent(t,}N_s\text{,p,}\emptyset\text{)} & = & \emptyset \\
\text{parent(t,}N_s\text{,p,}\epsilon\text{)} & = & \emptyset \\
\text{parent(t,}N_s\text{,p,b)} & = & \emptyset \\
\text{parent(t,}N_s\text{,p,n)} & = & \begin{cases} p & \text{if} \quad n \in N_s \\ \emptyset & \text{else} \end{cases} \\
\text{parent(t,}N_s\text{,p,e[r])} & = & \begin{cases} \text{parent}(t, N_s, e[r], r) \mid p & \text{if} \quad \text{t is subtype of} \quad e[r] \\ \text{parent}(t, N_s, e[r], r) & \text{else} \end{cases} \\
\text{parent(t,}N_s\text{,p,r|s)} & = & \text{parent(t,}N_s\text{,p,r)} \mid \text{parent(t,}N_s\text{,p,s)} \\
\text{parent(t,}N_s\text{,p,r;s)} & = & \text{parent(t,}N_s\text{,p,r)} \mid \text{parent(t,}N_s\text{,p,s)} \\
\text{parent(t,}N_s\text{,p,r}*\text{)} & = & \text{parent(t,}N_s\text{,p,r)}
\end{array}$$

with $b \in B, n \in N, e \in E, p, r, s, t \in Reg$ and $N_s \subset N$. □

The function `parent` is based on the four parameter auxiliary function. The parameter t is the element type for which the parent element types are searched. The set N_s consists of the nonterminal symbols that t encloses as subtypes in the XPath type of the self axis. The parameter p accumulates the current parent type, while r is the currently examined regular hedge expression.
The idea of the parent type computation is that each nonterminal symbol being part of the formalized hedge grammar is checked whether its production contains the given element type as subtype. If such a nonterminal is found, the regarding accumulated parent type is returned. Thus nonterminal symbols are not examined recursively. Instead it is checked if a nonterminal symbol is in the set N_s, which is created before.
Finally, with the help of these functions the type inference rules for XPath expressions in **XOBE**$_{\text{DBPL}}$ can be formulated. The complete set of rules can be found in [42]. Definition 3.2.9 only lists some examples.

Definition 3.2.9 The following XPath type inference rules are only a subset of the complete list given in [42].

$$\frac{\text{variable} : \text{r} \in Reg}{\text{variable} : \texttt{self}(\text{r})*} \quad \textbf{(VAR)}$$

$$\frac{\text{ls} : \text{r}*}{\text{ls/child} : \texttt{child}(\text{r})*} \quad \textbf{(CHILD)}$$

$$\frac{\text{ls} : \text{r}*}{\text{ls/descendant} : \texttt{descendant}(\text{r})*} \quad \textbf{(DESC)} \qquad \square$$

$$\frac{\text{ls} : \text{r}*}{\text{ls/parent} : \texttt{parent}(\text{r})*} \quad \textbf{(PAR)}$$

$$\frac{\text{e} \in \text{E}, \text{ls/axis} : \text{r}*}{\text{ls/axis::e} : \texttt{nodeTest}(\text{e,r})*} \quad \textbf{(TEST)}$$

Since predicates in an XPath expression only work as a filter upon the selected nodes, they must not be taken into account for type inference. The type of an arbitrary XPath expression, without taking into account any predicates, is always a super type of its exact type. Now the basis for static type checking, in particular type inference and the type inference rules for XPath expressions in **XOBE**$_{\text{DBPL}}$, is given. The next sections will introduce syntax, semantics and type inference rules for complex queries and updates in **XOBE**$_{\text{DBPL}}$ which are also the topic of [72]. It is important to notice that the subtype algorithm explained in [42, 43] can be transferred without any modification.

3.3 Extended FLWOR Expressions

Simple queries upon XML objects can already be formulated as XPath expressions in **XOBE**. In **XOBE**$_{\text{DBPL}}$ e**x**tended FLWOR (xFLWOR) expressions supporting complex queries as well as updates for XML objects are added. xFLWOR expressions in **XOBE**$_{\text{DBPL}}$ adopt syntactical proposals of [89] to extend XQuery's well-known FLWOR expression construct, for details see sections 2.7 and 2.4 respectively. In this approach any return clause can optionally be replaced by an update clause. The corresponding part of the **XOBE**$_{\text{DBPL}}$ grammar is given in definition 3.3.1.

Definition 3.3.1 An xFLWOR expression can be derived from an expression and is defined by the following grammar:

$$
\begin{array}{lll}
\textit{expression} & \rightarrow & ...|\textit{xpath\_expression}|\textit{xflwor\_expression} \\
\textit{xflwor\_expression} & \rightarrow & (\textit{let\_clause}|\textit{for\_clause})+ \\
 & & \textit{where\_clause}? \\
 & & ((\textit{order\_by\_clause}?\ \textit{return\_clause})\ |\textit{update\_clause}) \\
\textit{let\_clause} & \rightarrow & \textbf{LET}\ \textit{var\_name}\ \textbf{':='}\ (\textit{var\_name}\ |\ \textit{xpath\_expression}) \\
\textit{for\_clause} & \rightarrow & \textbf{FOR}\ \textit{var\_name}\ \textbf{IN}\ (\textit{var\_name}\ |\ \textit{xpath\_expression})
\end{array}
$$

□

The nonterminals *where_clause* and *order_by_clause* are defined exactly like in the FLWOR expression grammar given in definition 2.4.1. The grammar for *return_clause*s and *update_clause*s is given in definitions 3.3.3 and 3.3.5. Since an *order_by_clause* generates a certain order of the resulting tuples, it does not make much sense in the context of an *update_clause*. Updates do not produce any return values.
Type inference rules for the type of a let as well as for a for clause variable can be given without further auxiliary functions. The rules presented in definition 3.3.2 solely rely on those for XPath expressions given in definition 3.2.9.

Definition 3.3.2 The type inference rules for variables in a let and a for clause are:

$$\frac{\text{variable : r} \in Reg}{\text{for\_variable : r}} \quad \frac{\text{xpath expression : r}* \in Reg}{\text{for\_variable : r}} \qquad \textbf{(FOR)}$$

$$\frac{\text{variable : r} \in Reg}{\text{let\_variable : }\texttt{self}(\text{r})*} \quad \frac{\text{xpath expression : r}* \in Reg}{\text{let\_variable : r}*} \qquad \textbf{(LET)}$$

□

According to the semantics of for and let variables in XQuery (see section 2.4 for more details) a let variable is always bound to an XPath type of either the assigned variable or the XPath expression. On the contrary a for variable that is iterated is always bound to the variable type itself or respectively to the XPath's inner choice type.
In section 3.3.1 complex query expressions in **XOBE**$_{\text{DBPL}}$ that are formulated as xFLWOR expressions containing a return clause are explained in more detail. In section 3.3.2 the same is done with update expressions in **XOBE**$_{\text{DBPL}}$ expressed by xFLWOR expressions with an update clause.

3.3.1 Query Expressions

In **XOBE**$_{\text{DBPL}}$ xFLWOR expressions containing a return clause are called flwor expressions. Up to now, XML objects can only be queried with the help of simple XPath path expressions. Flwor expressions enable the formulation of complex queries including joins across several objects and the construction of complex query results. Query results are constructed by the result clause. A result clause in **XOBE**$_{\text{DBPL}}$ is mainly based on XPath expressions and XML object constructors. The syntax of a return clause is given in definition 3.3.3.

Syntax and Semantics

Definition 3.3.3 A return clause in **XOBE**$_{\text{DBPL}}$ is defined by the following grammar:

return_clause	→	**RETURN** *return_expression*
return_expression	→	(*xml_object*|*xpath_expression*|*var_name*)+

with the nonterminals *var_name* standing for a variable name, *xpath_expression* as defined in the grammar 3.2.3 and *xml_object* standing for an XML object constructor as defined in [42]. □

The grammar states that a return clause is initiated by the keyword **RETURN** followed by a sequence consisting of XML object constructors, XPath path expressions or variable names. At least one XML object constructor, path expression or variable name must be given. A variable that is used within a return clause must be formerly defined and has to be of an XML type including lists, single XML objects and basis types. Return clauses in **XOBE**$_{\text{DBPL}}$ can be nested implicitly if a variable name is used that references another flwor expression.
Listing 3.2 gives an impression how flwor expressions can be used in a **XOBE**$_{\text{DBPL}}$ program.

Listing 3.2: An example **XOBE**$_{\text{DBPL}}$ method using a flwor expression

```
 1 public void printCurrentSellerItems(){
 2  //Lists the names of persons and the quantity of items they
 3  //are currently selling
 4  xml<(name;quantity*)*> namesAndNumbers =
 5     $FOR i IN auctionSite//person
 6     LET j := auctionSite//open_auction[/seller/@id=i/@id]
 7     RETURN i/name j/quantity$;
 8
 9       //Prints the query result on the screen
10  for(int i=0; i<namesAndNumbers.size(); i++){
11    System.out.println("name of seller and quantity of items");
12    System.out.println("currently sold at auction: ");
13    System.out.println(namesAndNumbers.get(i));
14  }
15 }
```

The **XOBE**$_{\text{DBPL}}$ method `printCurrentSellerItems` in listing 3.2 searches for names of sellers and the quantity of items that they are currently selling at auction. The xFLWOR query in lines 4-8 is similar to the Q8 query described in the XMark paper [67]. In our example the

query joins persons with open auction elements. Finally in lines 10-13 the query result is printed on the screen.
The next paragraph introduces type inference rules to statically type check these flwor expressions.

Type Inference Rules

In this section we will concentrate on type inference of xFLWOR expressions containing a return clause. Such an expression consists of one to many let and for clauses defining local variables, their corresponding types can be inferred with the known rules 3.2.9 and 3.3.2. Consequently a new rule is merely needed for the return clause itself. Finally the type of the whole result clause is defined to be the result type of the whole expression. The type inference rule for a result clause is given in definition 3.3.4. Analogously to predicates in XPath expressions, neither where nor order by clauses have got influence on the type inference process.

Definition 3.3.4

$$\frac{\forall c_i, 0 \leq i \leq n \in \textit{var\_name} \cup \textit{xml\_object} \cup \textit{xpath\_expression} : tc_i \in Reg}{\textbf{RETURN}\ c_0 ... c_n : (tc_0 \ ; ... ;\ tc_n)*} \quad \textbf{(RETURN)}$$

□

The main idea of the type inference rule for a return clause is that the types of the single constituents can be inferred with known rules. Within one execution of a return clause these component types are concatenated in a sequence type (;). Finally the enclosing Kleene star operator is added, because as far as a flwor expression contains at least one for clause, the return clause is iterated arbitrary times. Thus the single type of the return clause is repeated arbitrarily often. For an example let us infer and check the type of the complex flwor query in listing 3.2.

Example 3.3.1 This example infers the type of the flwor expression given in listing 3.2:

```
FOR i IN auctionSite//person
LET j := auctionSite//open_auction[/seller/@id=i/@id]
RETURN i/name j/quantity
```

The type of the for variable i can be inferred as *person* and the type of the let variable j as *open_auction*$*$ by the required type inference rules of definition 3.3.2. Then $i/name$ has got the XPath type *name*$*$ and $j/quantity$ the XPath type *quantity*$*$ applying the rules of 3.2.9. Finally we can apply the **RETURN** type inference rule yielding the final type:

$(name;quantity*)*$

This type is valid since it is the same type as the declared type $(name;quantity)*$ of the left hand side of the assignment in line 4 of listing 3.2.

The next section will cover syntax and semantics of update expressions in **XOBE**$_{\text{DBPL}}$ and most importantly, it introduces how these update expressions can be checked statically for structural validity.

3.3.2 Update Expressions

In **XOBE**$_{\text{DBPL}}$ xFLWOR expressions containing an update clause are called update expressions. Up to now XML objects can either be constructed or queried but not manipulated. Update expressions enable to insert, delete, rename and replace arbitrary subelements of an XML object. Once again these target sub elements are selected by XPath expressions. The syntax of an update clause is given in definition 3.3.5. An xFLWOR expression with an update clause is also called a flwu expression.

Syntax and Semantics

Definition 3.3.5 An update clause is defined by the following grammar:

update_clause	→	**UPDATE** *var_name suboperation* (,*suboperation*)∗
suboperation	→	*insert_operation*
		| *delete_operation*
		| *rename_operation*
		| *replace_operation*
insert_operation	→	**INSERT** *content*
		((**INTO**|**BEFORE**|**AFTER**)*location_path*)?
delete_operation	→	**DELETE** *location_path*
rename_operation	→	**RENAME** *location_path* **TO** *name*
replace_operation	→	**REPLACE** *location_path* **WITH** *content*
content	→	*var_name* | *xpath_expression* | *xml_object*

□

The grammar states that an update clause is initiated by the keyword **UPDATE** followed by a variable name upon which the update is performed. In the following, this variable is also called `update target`. The update operation is described by a sequence of fundamental

sub-operations. At least one sub-operation must be given. The target variable of the update operation must be formerly defined and has to be either an XML object or a list of XML objects. Each sub operation is performed successively upon the XML object(s). The *content* in **XOBE**$_{\text{DBPL}}$ is either a previously defined XML variable, an XML object or an XPath expression. The *location_path*s are XPath expressions selecting arbitrary **descendant** objects of the update variable as **implicit** context variable. This means that *location_path*s used to select the target nodes of an update operation always have to start with a slash **'/'**. It is important to notice that XPath expressions are limited to those selecting descendants, attributes or childs, because any other context cannot be guarenteed to exist for the XML objects within a **XOBE**$_{\text{DBPL}}$ program at runtime.
Listing 3.3 gives an impression how update expressions upon XML objects are used in a **XOBE**$_{\text{DBPL}}$ program.

Listing 3.3: An example **XOBE**$_{\text{DBPL}}$ method using an update expression

```
 1 synchronized int bid(String p_id, int incr, String a_id){
 2  //calculate new current
 3  xml<current*> cur =
 4   $auctionSite//open_auction[/@id={a_id}]/current$;
 5       current new_current =
 6   <current>{cur.itemAsInt(0)+incr}</current>;
 7
 8       //create new bidder
 9       bidder bid = <bidder>
10                     <date>{getDate()}</date>
11                     <time>{getTime()}</time>
12                     <personref person={p_id}/>
13                     <increase>{incr}</increase>
14                    </bidder>;
15
16  //update auction
17  $LET i := auctionSite//open_auction[/@id={a_id}]
18   UPDATE i INSERT {bid} BEFORE /current,
19   REPLACE /current WITH {new_current}$;
20
21  return new_current;
22 }
```

The **XOBE**$_{\text{DBPL}}$ method `bid` in listing 3.3 registers a new bid for an auction. The bidder and the auction are selected by their ids. Additionally the increase is passed as parameter as well. In lines 3-6 the new current bid is calculated and in a second step in lines 9-14 the new bidder is created as an XML object. Finally in line 17-19 the update operation upon the auction site is executed. The update operation itself consists of an insert before and a replace. The first is needed to insert the new bidder at the right place and the second to replace the old with the new current. Please notice that this method is declared as `synchronized`. `synchronized` is used in connection with Java threads and guarantees that methods are not executed in parallel. Next it is introduced how these updates can be checked at compile time. A statically checked update operation is guaranteed to only produce structurally valid XML objects at runtime.

Type Inference Rules

In this section we will concentrate on type inference of xFLWOR expressions containing an update clause. Such an expression consists of one to many let and for clauses defining local variables, their corresponding types can be inferred with the known rules 3.2.9 and 3.3.2. Consequently a new set of rules is merely needed for the update clause itself. Finally the type of the updated variable is defined to be the result type of the whole expression. Like for where clauses of flwor expressions, where clauses being part of an update do not influence the type inference process too.
In the following we will concentrate on inferring types of update clauses with the original type of variable i already given.
The basic idea in the context of the following update type inference rules is to infer the parent type of the XML objects that are the target of the update operation. In a second step the update operation is applied to the parent type. Finally, it is checked if the manipulated parent type is still valid according to the corresponding formalized schema type. According to the restriction that only descendants, childs and their attributes of an XML object can be updated, the required parent context can be guaranteed.
In general there are seven different update cases:

1. delete: UPDATE i DELETE xpath

2. simple insert: UPDATE i INSERT content

3. insert: UPDATE i INSERT content INTO xpath

4. insert after: UPDATE i INSERT content AFTER xpath

5. insert before: UPDATE i INSERT content BEFORE xpath

6. replace: UPDATE i REPLACE xpath WITH content

7. rename: UPDATE i RENAME xpath WITH name

For each update case a specific type inference function is required and will be given later in this section. Then statically checking updates is done in three steps:

1. Infer the XPath type r with the set of XPath type inference rules given in 3.2.9 and determine the nodeTest $test$ of the update operation. The determination of r and $test$ depends on the regarding update case:

1. **delete:** r is the type of *i*/*xpath*/parent::$*$ and $test$ is the node test of the last location step in *xpath*.
2. **simple insert:** r is the type of the declared variable i and $test$ is the wildcard $*$.
3. **insert:** r is the XPath type of *i*/*xpath* and $test$ is the wildcard $*$.
4. **insert after:** r is the type of *i*/*xpath*/parent::$*$ and $test$ is the node test of the last location step in *xpath*.
5. **insert before:** same as `insert after`.
6. **replace:** same as `insert after`.
7. **rename:** same as `insert after`.

2. Apply the corresponding new update type inference rule to the type r and node test $test$ gained in the first step. The result of this step is a type r' that is the updated parent type.

3. Perform the subtype check. If r' is still a valid subtype of r, the update is statically valid!

The node test of the last location step of an XPath path expression can be evaluated by the following function defined in `lastTest` 3.3.6.

Definition 3.3.6 The function `lastTest` : `xpath_expression` $\rightarrow E$ determines the node test of the last location step of an **XOBE**$_{\text{DBPL}}$ XPath expression and is defined:

lastTest(name)	=	$*$
lastTest(xpath / axis::test)	=	test
lastTest(xpath /axis::test[p])	=	test

with `name` $\in$ *var_name*, `xpath` $\in$ *xpath_expression*, `axis` $\in$ *axis_specifier*, `test` $\in$ *node_test* and `p` $\in$ *predicate* as defined by **XOBE**$_{\text{DBPL}}$ XPath expression grammar 3.2.3. □

An example demonstrating how the function `lastTest` works is given below.
Example 3.3.2 The last node test of the XPath expression $auctionSite//open\_auction[/@id ='' ...'']$ is calculated as:

lastTest($auctionSite/descendant$ :: $open\_auction[/@id ='' ...''])$ =
$open\_auction$

After describing the process in principal, we will go into more detail, by giving the new set of update type inference rules. Before these rules can be defined some auxiliary functions need to be introduced. The auxiliary type inference function in case of a delete operation is defined in 3.3.7.

Definition 3.3.7 The function $delete : Reg \times E \rightarrow Reg$ infers the updated parent type of a delete operation upon the child nodes satisfying the nodetest $test$ of the original parent type r.

delete(r,test) = delete(r,test,$\{\}$)

The function $delete$ is based on the three parameter function $delete : Reg \times E \times P(N) \rightarrow Reg$ and $P(N) = \{M \mid M \subseteq N\}$. The three parameter $delete$ function collects the already processed nonterminals to avoid infinite loops. This function is defined recursively:

$$\begin{array}{lcl}
\text{delete}(\emptyset,\text{test},N_v) & = & \emptyset \\
\text{delete}(\epsilon,\text{test},N_v) & = & \epsilon \\
\text{delete}(\text{b},\text{test},N_v) & = & \text{b} \\
\text{delete}(\text{n},\text{test},N_v) & = & \begin{cases} \emptyset & \text{if} \quad n \in N_v \\ \text{delete}(r, \text{test}, N_v \cup n)\text{with} \quad n \rightarrow r \in P & \text{else} \end{cases} \\
\text{delete}(\text{e[r]},\text{test},N_v) & = & \text{e[r'] with r':=del(r,test,}N_v) \\
\text{delete}(\text{r}|\text{s},\text{test},N_v) & = & \text{delete(r,test,}N_v) \mid \text{delete(s,test,}N_v) \\
\text{delete}(\text{r;s},\text{test},N_v) & = & \text{delete(r,test,}N_v) \text{ ; delete(s,test,}N_v) \\
\text{delete}(\text{r}*,\text{test},N_v) & = & \text{delete(r,test,}N_v)
\end{array}$$

The function `del` : $Reg \times E \times P(N) \rightarrow Reg$ is defined analogously to the three parameter function `delete`, except:

$$\text{del(e[r],test,}N_v) = \begin{cases} e[r] & \text{if} \quad e \neq test \\ \epsilon & \text{else} \end{cases}$$

with $b \in B, n \in N, test, e \in E, r, s \in Reg$ and $N_v \subset N$. □

Example 3.3.3 Let us look at a delete operation example and see how the above rule infers the updated parent type. The following update deletes a person element with a given id. It is assumed that `auctionSite` is a previously defined variable of the XML type *site*.

$**LET** i:=auctionSite/people
UPDATE i **DELETE** i/person[/@id="p001"]$

First the XPath type r is inferred according to the **delete** case. In the example it is the type of $auctionSite/people/person[/@id =''\ 001'']/parent :: *$, which is $people*$. The last test *test* is `person`. Now the new type inference rule as defined in 3.3.7 can be applied.

$$\begin{array}{lcl}
\text{delete(people}*\text{,person)} & = & \text{delete(people}*\text{,person,}\{\}\text{)} \\
\text{delete(people}*\text{,person,}\{\}\text{)} & = & \text{delete(people,person,}\{\}\text{)} \\
 & = & \text{delete(}\textbf{people}\text{[person}*\text{],person,}\{\text{people}\}\text{)} \\
 & = & \textbf{people}\text{[r']} \\
\text{r'} & = & \text{del(person}*\text{,person,}\{\text{people}\}\text{)} \\
 & = & \text{del(person,person,}\{\text{people}\}\text{)} \\
 & = & \text{del(}\textbf{person}\text{[}\textbf{@id}\text{[string];...],person,}\{\text{people,person}\}\text{)} \\
 & = & \epsilon
\end{array}$$

The updated parent type is inferred as **people**[ϵ], which is the expected result. Predicates as explained in [42] as well are not taken into account since they only act as a filter on preselected nodes. Static type checking ignoring predicates implicitly assumes worst cases. Since the updated parent type is a valid subtype of the auction *people* type, the update is statically checked to be valid.

In case of a simple insert the corresponding type inference function is given in 3.3.8.

Definition 3.3.8 The function `insert` : $Reg \times Reg \rightarrow Reg$ infers the updated type of a simple insert operation that inserts the given content of type c as child into the update target of the original type r:

$$\text{insert(r,c)} \quad = \quad \text{insert(r,c,}\{\}\text{)}$$

The three parameter auxiliary function `insert` : $Reg \times Reg \times P(N) \rightarrow Reg$ is defined recursively:

$$\begin{array}{lcl}
\text{insert(}\emptyset\text{,c,}N_v\text{)} & = & \emptyset \\
\text{insert(}\epsilon\text{,c,}N_v\text{)} & = & \epsilon \\
\text{insert(b,c,}N_v\text{)} & = & \text{b} \\
\text{insert(n,c,}N_v\text{)} & = & \begin{cases} \emptyset & \text{if} \quad n \in N_v \\ \text{insert}(r, c, N_v \cup n)\text{with} \quad n \rightarrow r \in P & \text{else} \end{cases} \\
\text{insert(e[r],c,}N_v\text{)} & = & \text{e[r;c]} \\
\text{insert(s|t,c,}N_v\text{)} & = & \text{insert(s,c,}N_v\text{) | insert(t,c,}N_v\text{)} \\
\text{insert(s;t,c,}N_v\text{)} & = & \text{insert(s,c,}N_v\text{) ; insert(t,c,}N_v\text{)} \\
\text{insert(s}*\text{,c,}N_v\text{)} & = & \text{insert(s,c,}N_v\text{)}
\end{array}$$

with $b \in B, n \in N, e \in E, r, s, c \in Reg$ and $N_v \subset N$. □

Example 3.3.4 The following update operation is an example for a simple insert operation. The update inserts a person element into the people element. It is assumed that the variable `auctionSite` has got the XML type *site*.

person p = ...;

$**LET** i:=auctionSite//people
UPDATE i **INSERT** {p} $

First r and $test$ have to be calculated according to the **simple insert** rule. Thus, r is the type of $auctionSite//people$ which is *people*∗ and $test$ is the wildcard $*$. The type of the content which is going to be inserted is *person*. Next the simple insert type inference rule can be applied.

$$\begin{array}{lcl} \text{insert(people}*\text{,person)} & = & \text{insert(people}*\text{,person,}\{\}) \\ \text{insert(people}*\text{,person,}\{\}) & = & \text{insert(people,person,}\{\}) \\ & = & \text{insert(}\textbf{people}\text{[person}*\text{],person,}\{\text{people}\}) \\ & = & \textbf{people}\text{[person}*\text{;person]} \end{array}$$

The updated parent type is inferred as **people**[person∗;person] which is a valid subtype of the auction schema type *people*. According to this the example update operation is checked successfully.

The type inference function for an insert operation is given in 3.3.9.

Definition 3.3.9 The function `insert` : $Reg \times Reg \times E \to Reg$ infers the updated parent type in case of an insert operation given the type of the content c and the node test $test$ of the target child nodes.

$$\text{insert(r,c,test)} \quad = \quad \text{insert(r,c,test,N)}$$

The four parameter auxiliary function `insert` : $Reg \times Reg \times E \times P(N) \to Reg$ is defined recursively:

$$\begin{array}{lcl} \text{insert(}\emptyset\text{,c,test,}N_v) & = & \emptyset \\ \text{insert(}\epsilon\text{,c,test,}N_v) & = & \epsilon \\ \text{insert(b,c,test,}N_v) & = & \text{b} \\ \text{insert(n,c,test,}N_v) & = & \begin{cases} \emptyset & \text{if} \quad n \in N_v \\ \text{insert}(r, c, test, N_v \cup n)\text{with} \quad n \to r \in P & \text{else} \end{cases} \\ \text{insert(e[r],c,test,}N_v) & = & \begin{cases} e[r] & \text{if} \quad e \neq test \\ e[r;c] & \text{else} \end{cases} \\ \text{insert(s|t,c,test,}N_v) & = & \text{insert(s,c,test,}N_v) \mid \text{insert(t,c,test,}N_v) \\ \text{insert(s;t,c,test,}N_v) & = & \text{insert(s,c,test,}N_v) \text{ ; insert(t,c,test,}N_v) \\ \text{insert(s}*\text{,c,test,}N_v) & = & \text{insert(s,c,test,}N_v) \end{array}$$

with $b \in B, n \in N, test, e \in E, r, s, c \in Reg$ and $N_v \subset N$. □

The type inference function for an insert before update is presented in definition 3.3.10. Since the type inference function for an insert after operation works equivalently to an insert before it is not given here.

Definition 3.3.10 The function `insertBefore` $Reg \times Reg \times E \rightarrow Reg$ infers the updated parent type of an insert before operation given the original parent type r, the type of the content c and the node test $test$ selecting the target children.

insertBefore(r,c,test) = insertBefore(r,c,test,$\{\}$)

The four parameter function `insertBefore` : $Reg \times Reg \times E \times P(N) \rightarrow Reg$ is defined analogously to the four parameter function `insert`, except:

insertBefore(e[r],c,test,N_v) = e[r'] ,with r':=insBef(r,c,test,N_v)

The auxiliary function `insBef` : $Reg \times Reg \times E \times P(N) \rightarrow Reg$ is defined recursively:

$$\begin{array}{lcl}
\text{insBef}(\epsilon,\text{c,test},N_v) & = & \begin{cases} c & \text{if test is wildcard} \\ \epsilon & \text{else} \end{cases} \\
\text{insBef(b,c,test},N_v) & = & \begin{cases} c; b & \text{if test is wildcard} \\ b & \text{else} \end{cases} \\
\text{insBef(n,c,test},N_v\text{)} & = & \begin{cases} n & & \text{if} \quad n \in N_v \\ \text{insBef}(r, c, test, N_v \cup n)\text{with} & n \rightarrow r \in P & \text{else} \end{cases} \\
\text{insBef(e[r],c,test},N_v) & = & \begin{cases} c; e[r] & \text{if} \quad e = test \\ e[r] & \text{else} \end{cases} \\
\text{insBef(r|s,c,test},N_v\text{)} & = & \text{insBef(r,c,test},N_v\text{) | insBef(s,c,test},N_v) \\
\text{insBef(r;s,c,test},N_v) & = & \text{insBef(r,c,test},N_v\text{) ; insBef(s,c,test},N_v) \\
\text{insBef(r}*\text{,c,test},N_v) & = & \text{insBef(r,c,test},N_v)
\end{array}$$

with $b \in B, n \in N, test, e \in E, r, s, c \in Reg$ and $N_v \subset N$. □

The type inference function in case of a replace update operation is defined in definition 3.3.11.

Definition 3.3.11 The function `replace` : $Reg \times Reg \times E \rightarrow Reg$ infers the type of a replace operation upon children selected by the node test $test$ of the original parent type r. The type of the new content is c.

replace(r,c,test) = replace(r,c,test,$\{\}$)

The four parameter function `replace` : $Reg \times Reg \times E \times P(N) \rightarrow Reg$ is defined analogously to the four parameter function `insertBefore`, except:

replace(e[r],c,test,N_v) = e[r'] , with r':=rep(r,c,test,N_v)

And the auxiliary function `rep` : $Reg \times Reg \times E \times P(N) \rightarrow Reg$ is defined recursively:

$$
\begin{array}{lcl}
\text{rep}(\epsilon,\text{c,test},N_v) & = & \epsilon \\
\text{rep(b,c,test},N_v) & = & \begin{cases} c & \text{if b satisfies test} \\ b & \text{else} \end{cases} \\
\text{rep(n,c,test},N_v\text{)} & = & \begin{cases} n & \text{if} \quad n \in N_v \\ \text{rep}(r, c, test, N_v \cup n)\text{with} \quad n \rightarrow r \in P & \text{else} \end{cases} \\
\text{rep(e[r],c,test},N_v) & = & \begin{cases} c \quad \text{if} \quad e = test \\ e[r] \quad \text{else} \end{cases} \\
\text{rep(r|s,c,test},N_v\text{)} & = & \text{rep(r,c,test},N_v) \mid \text{rep(s,c,test},N_v) \\
\text{rep(r;s,c,test},N_v) & = & \text{rep(r,c,test},N_v) \text{ ; rep(s,c,test},N_v) \\
\text{rep(r}*\text{,c,test},N_v) & = & \text{rep(r,c,test},N_v)
\end{array}
$$

with $b \in B, n \in N, test, e \in E, r, s, c \in Reg$ and $N_v \subset N$. □

Example 3.3.5 An example update operation performing a replace operation is shown below. The update replaces the email address of a given person element. It is assumed that the variable `auctionSite` is previously defined and has got the type *site*.

emailaddress email = ...;
$**LET** i:=auctionSite//person[/@id=”p001”]
UPDATE i **REPLACE** i/emailaddress **WITH** {email}$

According to the **replace** case r is the type of
$auctionSite//person[/@id ='' p001'']/emailaddress/parent :: *$ which is *person*$*$
and the last node test $test$ is $emailaddress$. The type of the content is *emailaddress*. Now the replace type inference rule to infer the updated parent type can be applied.

replace(person$*$,emailaddress,emailaddress) =

replace(person$*$,emailaddress,emailaddress,$\{\}$)
= replace(person,emailaddress,emailaddress,$\{\}$)
= replace(**person**[**@id**[string];name;emailaddress;...],
emailaddress,emailaddress,$\{$person$\}$)
= **person**[r']
r' = rep((**@id**[string];name;emailaddress;...),
emailaddress,emailaddress,$\{$person$\}$)
= **@id**[string];name;
rep(emailaddress,emailaddress,emailaddress,$\{$person$\}$);...
= **@id**[string];name;
rep(**emailaddress**[string],emailaddress,emailaddress,$\{$person,emailaddress$\}$);...
= **@id**[string];name;
emailaddress;...

Finally the updated parent type is inferred as **person**[**@id**[string];name;emailadress;...], which is exactly the *person* XML type. Thus, the update is valid.

Finally the type inference rule for a rename update operation is given in definition 3.3.12.

Definition 3.3.12 The function `rename` : $Reg \times E \times E \rightarrow Reg$ infers the updated parent type of a rename operation given the node test $test \in E$ selecting the target child elements, the new name $name \in E$ and the original parent type $r \in Reg$.

rename(r,name,test) = rename(r,name,test,N_v)

The four parameter function `rename` : $Reg \times E \times E \times P(N) \rightarrow Reg$ is defined analogously to the four parameter function `insertBefore`, except:

rename(e[r],name,test,N_v) = e[r'] , with r':=ren(r,name,test,N_v)

The auxiliary function `ren` : $Reg \times E \times E \times P(N) \rightarrow Reg$ is defined recursively:

ren(ϵ,name,test,N_v) = ϵ
ren(b,name,test,N_v) = b
ren(n,name,test,N_v) = $\begin{cases} n & \text{if} \quad n \in N_v \\ \text{ren}(r, c, test, N_v \cup n)\text{with} \quad n \rightarrow r \in P & \text{else} \end{cases}$
ren(e[r],name,test,N_v) = $\begin{cases} name[r] & \text{if} \quad e = test \\ e[r] & \text{else} \end{cases}$
ren(r|s,c,test,N_v) = ren(r,c,test,N_v) | ren(s,c,test,N_v)
ren(r;s,c,test,N_v) = ren(r,c,test,N_v) ; ren(s,c,test,N_v)
ren(r$*$,c,test,N_v) = ren(r,c,test,N_v)

with $b \in B, n \in N, test, name, e \in E, r, s \in Reg$ and $N_v \subset N$. □

Now, that we have defined all necessary type inference functions for updates, we can give the final type inference rules for update operations. Similar to those for XPath they are listed in the definition 3.3.13.

Definition 3.3.13 The type of the updated target variable of an xFLWOR expression with an update clause is defined by the following type inference rules:

$$\frac{\text{i} : r \in Reg, \quad \textit{path}\text{/parent::}* : \text{p}* \in Reg}{\textbf{UPDATE}\ \text{i}\ \textbf{DELETE}\ \textit{path} : \texttt{delete}(\text{p}*, \texttt{lastTest}(\textit{path})) \in Reg} \quad \textbf{(DELETE)}$$

$$\frac{\text{i} : r \in Reg, \quad \text{content} : c \in Reg}{\textbf{UPDATE}\ \text{i}\ \textbf{INSERT}\ \text{content} : \texttt{insert}(\text{r},\text{c}) \in Reg} \quad \textbf{(SIMPLE INSERT)}$$

$$\frac{\text{i} : r \in Reg, \quad \textit{path} : \text{p}* \in Reg, \quad \text{content} : c \in Reg}{\textbf{UPDATE}\ \text{i}\ \textbf{INSERT}\ \text{content}\ \textbf{INTO}\ \textit{path} : \texttt{insert}(\text{p}*,\text{c},'*') \in Reg} \quad \textbf{(INSERT)}$$

$$\frac{\text{i} : r \in Reg, \quad \textit{path}\text{/parent::}* : p* \in Reg, \quad \text{content} : c \in Reg}{\textbf{UPDATE}\ \text{i}\ \textbf{INSERT}\ \text{content}\ \textbf{BEFORE}\ \textit{path} : \texttt{insertBefore}(\text{p}*,\text{c},\texttt{lastTest}(\textit{path})) \in Reg} \quad \textbf{(INSERT BEFORE)}$$

$$\frac{\text{i} : r \in Reg, \quad \textit{path}\text{/parent::}* : p* \in Reg, \quad \text{content} : c \in Reg}{\textbf{UPDATE}\ \text{i}\ \textbf{INSERT}\ \text{content}\ \textbf{AFTER}\ \textit{path} : \texttt{insertAfter}(\text{p}*,\text{c},\texttt{lastTest}(\textit{path})) \in Reg} \quad \textbf{(INSERT AFTER)}$$

$$\frac{\text{i} : r \in Reg,\quad \textit{path}/\text{parent::}* : p* \in Reg,\quad \text{content} : c \in Reg}{\textbf{UPDATE}\ \text{i}\ \textbf{REPLACE}\ \textit{path}\ \textbf{WITH}\ \text{content} : \texttt{replace}(p*, c, \texttt{lastTest}(\textit{path})) \in Reg}\ \textbf{(REPLACE)}$$

$$\frac{\text{i} : r \in Reg,\quad \textit{path}/\text{parent::}* : p* \in Reg,\quad \text{name} : e \in E}{\textbf{UPDATE}\ \text{i}\ \textbf{RENAME}\ \textit{path}\ \textbf{TO}\ \text{name} : \texttt{rename}(p*, e, \texttt{lastTest}(\textit{path})) \in Reg}\ \textbf{(RENAME)}$$

□

If an update clause consists of more than one basic update operation, e.g. delete, insert, replace, then the rules given in 3.3.13 are applied consecutively upon the current referenced static parent type. Since each sub operation can be applied to different descendants or child nodes of the update context variable and corresponding parent types may differ as well, parent types of each sub operation have to be valid according to the corresponding schema type. For an example let us infer and check the type of the update operation given in listing 3.3.

Example 3.3.6 This example infers the type of the update expression given in listing 3.3:

LET i:= auctionSite//open_auction[/@id={a_id}]
UPDATE i **INSERT** {bid} **BEFORE** /current,
REPLACE /current **WITH** {new_current}

The type of the local let variable i is inferred to be *open_auction*$*$ applying the rules of 3.3.2. As mentioned before, if an update clause consists of more than one update operation, the final type is inferred consecutively. Thus we start by inferring the type of the insert operation applied to i still having the original type. The content that is going to be inserted is a variable named `bid` and this variable is declared to be of type *bidder* in 3.3 line 9. The parent type of the target child nodes of the insert before operation selected by $/current$ can be inferred as *open_auction*$*$ with the XPath type inference rules. The preconditions to apply the desired type inference rule `(INSERT BEFORE)` are given. The required node test of the last location step in $/child :: current$ is evaluated as '$current$'. Applying the rule yields to the execution of the function `insertBefore` 3.3.10 with the following parameters and results in:

insertBefore(*open_auction*$\ast$,*bidder*,’current’) =
insertBefore(*open_auction*$\ast$,*bidder*,’current’,{}) =
insertBefore(*open_auction*,*bidder*,’current’,{}) =
insertBefore(**open_auction**[**@id**[string];*initial*;(*reserve*| ϵ);*bidder*$\ast$;*current*;
(*privacy*| ϵ);*itemref*;*seller*;*annotation*;*quantity*;*type*;
interval],*bidder*,’current’,{*open_auction*}) = **open_auction**[r’]
r’ =
insBef(**@id**[string];*initial*;(*reserve*| ϵ);*bidder*$\ast$;*current*;(*privacy*| ϵ);
itemref;*seller*;*annotation*;*quantity*;*type*;
interval,*bidder*,’current’,{*open_auction*}) =
insBef(**@id**[string],*bidder*,’current’,{*open_auction*}) ; ... ;
insBef(*interval*,*bidder*,’current’,{*open_auction*}) =
@id[string] ; ... ;
insBef(**current**[string],*bidder*,’current’,{*open_auction*,*current*}) ; ... ;
interval[string] =
@id[string]; ... ; *bidder* ; **current**[string] ; ... ; **interval**[string]

The updated parent type of the first insert before operation is inferred as:

open_auction[
@id[string];*initial*;(*reserve*| ϵ);
bidder$\ast$;*bidder*;*current*,(*privacy*| ϵ);
itemref;*seller*;*annotation*;*quantity*;*type*;*interval*
]

which is a valid subtype of the corresponding schema type *open_auction*.
To check the second replace operation, its updated parent type needs to be inferred first. The original parent type of this sub operation is inferred as *open_auction*$\ast$, because the child nodes are selected by $/current$. The type of the content is inferred as *current*, since the variable `new_current` is declared to be of type *current* in listing 3.3 line 5. Finally the required node test is once again $'current'$. Now that the preconditions are given, the rule **REPLACE** can be applied and results in:

replace(*open_auction*$\ast$,*current*,’current’) =
replace(*open_auction*$\ast$,*current*,’current’,{}) =
replace(*open_auction*,*current*,’current’,{}) =
replace(**open_auction**[c],*current*,’current’,{*open_auction*}) = **open_auction**[r’]
r’ =
rep(c,*current*,’current’,{*open_auction*}) =
... =
... ; rep(**current**[string],*current*,’current’,{*open_auction*,*current*}) ; ... =
... ; *current* ; ...

The updated parent type of the replace operation is inferred as:

open_auction[
@id[string];*initial*;(*reserve*| ϵ);
bidder;*current*,(*privacy*| ϵ);
itemref;*seller*;*annotation*;*quantity*;*type*;*interval*
]

which is valid, because it is exactly the corresponding schema type *open_auction*. After checking each update sub operation successfully for structural validity, the whole xFLWOR expression is determined to be valid.

In section 9.1 XML update tests that are checked for validity are performed. Several approaches mentioned in the sections about related work 2.7 and 2.9 are analyzed and compared with **XOBE**$_{\text{DBPL}}$.
As mentioned before, static type checking in general has got its limitations. In section 3.4 some details about adding dynamic type checking capabilities that support the static type checking process in **XOBE**$_{\text{DBPL}}$ are given.

3.4 Dynamic Type Checking

Static type checking is limited in many ways. One kind of problems arises, if it is based on type inference, because types are inferred and described using regular (hedge) expressions. In general true types belong to a more powerful grammar class, in particular context free grammars. Thus the true type can only be approximated, which leads to false negative type errors. Contrarily, by using more powerful types, subtype checking becomes undecidable. Details about the type checking problem in XML are given in [78]. In this section we will rather concentrate on the problem, that statically inferred types may be too restrictive. Statically inferred types describe the most general type. This general type can cause a structural type error, even if at runtime a structural constraint won't ever be violated. Besides the lack of checking value-based constraints, there are even structural conditions that cannot be checked statically in general. In **XOBE**$_{\text{DBPL}}$ statically detected structural type errors can be divided into two groups. On the one hand there are type errors that will always lead to a runtime type error, e.g. inserting an emailaddress element before the name element of a person. And on the other hand there are type errors that may, but do not necessarily lead to a runtime error, e.g. deleting an open auction element inside the open auctions element, if the type would be defined as <!ELEMENT

open_auctions (open_auction+)> in DTD notation. The latter case can be detected and distinguished by the static type checking analysis. A static error, which may but does not necessarily result in a type error at runtime, is given if all occurence constraints in the supertype containing an unbounded maximum are replaced by Kleene Star types (minimum occurence is zero and maximum occurence is unbounded) and the subtype algorithm with the original inferred subtype and the modified supertype succeeds. Thus, the **XOBE**$_{\text{DBPL}}$ parser returns structural type warnings instead of errors.
The main idea is that if the parser detects a statical type error, it inserts specific code right after the corresponding XML expression, e.g. an XPath expression, an XML object constructor, XML update or query. In **XOBE**$_{\text{DBPL}}$ XML objects are translated into an object framework similar to the DOM, details about the transformation process can be found in [42] and 7.2. Thus the code contains the following directions.

- The statically required regular hedge type is assigned to the critical XML object
- a `check` method is invoked upon this XML object. At runtime the `check` method infers the dynamic regular hedge expression type and checks if the dynamic type is a valid subtype of its static type.

The `Node` interface used in the **XOBE**$_{\text{DBPL}}$ DOM version needs to be extended by the following methods listed in the table 3.1.

Table 3.1: Additional `Node` methods required for dynamic type checking in **XOBE**$_{\text{DBPL}}$

method	**description**
void check() throws xobe.exception.XMLTypeException;	checks if the dynamic type is a valid subtype of the statically required type, raises an XML type exception, if the dynamic type is not valid
xobe.types.RegType infer();	infers the dynamic type of this node
void setHedgeType(xobe.types.RegType type);	sets the statically required type of this node
xobe.types.RegType getHedgeType();	returns the statically required type of this node

The dynamic type of an XML object can be inferred with the same rules given in [42] for XML object constructors. An example is given in 3.4.1.

Example 3.4.1 Let us consider the following XPath expression and assignment:

xml<person> person = $auctionSite//person[/@id='p_00001']$;

Since the right hand side type is inferred as *person*∗ and the left hand side type is declared as *person*, a statical type error will always be risen. Contrarily, at runtime the dynamic type of the right hand side will always be *person*, because the attribute id is unique, and therefore valid. In **XOBE**$_{\text{DBPL}}$ the static error is detected by the parser and the code is changed to:

xml<person> person = $auctionSite//person[/@id='p_00001']$;
person.setHedgeType(*person as hedge type*);
person.check();

An implementation of the methods `check` and `infer` declared by the `Node` interface is outlined in listing 3.4. The implementation of the `infer` method in 3.4 is suitable for an XML object representing an XML element.

Listing 3.4: Implementation of the `Node`'s interface methods `check` and `infer`

```
void check() throws xobe.exception.XMLTypeException{
 xobe.types.RegType dynType = this.infer();
 //call subtype check algorithm
 //check if dynamic type is subtype of static type
 Inequality ineq =
  new Inequality(dynType, this.statType , this.schema, this.schemaIneq);
  if(!ineq.check()){
   //if dynamic type is not subtype of static type
   //raise error
   throw new XMLTypeException("...");
  }
}

xobe.types.RegType infer(){

 xobe.types.RegType r = new xobe.types.ElementType(this.name);

 //infer and set the attribute types

 //infer and set the content type recursively

 return r;
}
```

This section about dynamic type checking in **XOBE**$_{\text{DBPL}}$ finishes the part about XML integration. The next chapter 4 will introduce persistency in **XOBE**$_{\text{DBPL}}$.

Chapter 4

Persistency

Up to now XML objects as well as general Java objects are transient meaning that these objects have got application lifetime and data gets lost each time an application finishes. As defined in the requirements for a database programming language in section 2.11, persistency is expected to be orthogonal, type independent and transparent. In contrast to tools offering type dependent persistency, where objects of some types can become persistent while others not, **XOBE**$_{\text{DBPL}}$ offers transparent, orthogonal and type independent persistency by introducing a persistent environment called *database*. In case of persistency frameworks class declarations often have to fullfill certain conditions, e.g. descriptor files are needed for listing persistent capable types as it is done in JDO. Moreover, sometimes these approaches do not even support inheritance among persistent types, e.g. entity beans in EJB. Details about these related approaches can be found in section 2.12. A *database* declaration in **XOBE**$_{\text{DBPL}}$ is used analogously to class declarations well known from Java. The extended grammar part can be seen in definition 4.1.1.

4.1 Syntax and Semantics

Definition 4.1.1 A persistent environment or database in **XOBE**$_{\text{DBPL}}$ is constructed according to the following grammar:

type_declaration	→	';' | *modifiers*(*class_or_database_interface_declaration* | *enum_declaration* | *annotation_type_declaration*)
class_or_database_interface_declaration	→	(**class**|**database**|**interface**) *identifier* [*type_parameters*] [*extends_list*] [*implements_list*] *type_body*

The nonterminal *type_declaration* is extended so that it is now possible to declare a type within a persistent environment. Its declaration is marked by the keyword **database**. Everything else remains unchanged in reference to the Java 1.5 language grammar. □

Since a database declaration is equivalent to a class declaration the most important difference is that the keyword **database** implies that generated objects of this database implicitly become persistent. Member variables of such database declarations become persistent by reachability **regardless of type** and without modifying formerly defined class declarations. A database member becomes persistent except when it is declared as `transient`. The `transient` modifier is already defined in Java. Another possibility for an extension of the Java grammar had been to introduce a new member variable modifier, e.g **persistent** that is used contrarily to the existing modifier **transient**. This latter kind of extension would rather fit to a persistent object oriented programming language than to a database programming language. In contrast to mark single members of an object as persistent, the database declaration environment allows to define a persistent environment. Modifying members within such an environment, e.g. invoking methods within a persistent environment, can be handled as transactions implicitly. Thus, this approach provides the required user transparency. Besides, most concepts and aspects discussed in the rest of this work would have stayed the same. Now we want to model the auction scenario in **XOBE**$_{\mathrm{DBPL}}$. By defining the auction elements within a persistent environment, its objects and sub objects will be kept persistent. From the programmer's point of view it is sufficient to declare the `AuctionSite` declaration in example 4.1.1 as *database*, everything else is done automatically.

Example 4.1.1

```
public database AuctionSite{
 private site auctionSite;
 private String description;
 4
 5  //an example constructor
 6  public AuctionSite(String description){
 7   this.description = description;
 8   this.auctionSite = <site/>;
 9  }
10
11  //an example getter
12  public String getDescription(){
13   return this.description;
14  }
15
16  //an example setter
17  public void setDescription(String description){
18   this.description = description;
19  }
20
21  //an example method that registers a new person
22  public boolean registerPerson(String p_name,
23                               String p_email,
24                               String p_id){
25
26   //create new person
27   person p = createPerson(p_name,p_email,p_id);
28
29   //person already exists
30   if(p==null)return false;
31
32   //register the new person to the auction's
33   //people element
34   $LET i:= auctionSite/people
35    UPDATE i INSERT {p}$;
36
37    return true;
36   }
37
38  //all methods from previous listings
39  //can remain unchanged and could be
40  //added here
41 }
```

The corresponding method to the method invocation `createPerson` in line 27 is given in listing 3.1. Please notice that the lifetime of local variables and formal parameters is the same as in classes. Only member variables are affected by the persistent environment, which is a consistent extension in syntax and semantics.

4.1.1 Creation and Retrieval

Any constructor being part of a database declaration, e.g. `AuctionSite(String description)` in line 6 of listing 4.1.1, generates new persistent objects. Method invocations upon these objects persistently modify them. Each method invocation is implicitly handled as a transaction as mentioned before. An example application program that generates a new persistent auction site object and registers a person is shown in the example 4.1.2.

Example 4.1.2

```
public class MainCreation{
 public static void main(String[] args){

  //creates a new persistent auction site object
  AuctionSite auction =
   new AuctionSite("an example auction site");

  //registers a person to the auction site object
  //thus modifying the object persistently
  auction.registerPerson("Mary Fernandez",
                         "fernandez@xquery.com",
                         "p_00001");
 }
}
```

To retrieve and access an already existing persistent object **XOBE**$_{\text{DBPL}}$ provides an XPath expression searching for all objects of a given class with given member variable values. This kind of retrieval and access is offered in addition to reference passing as it is known in the case of transient objects. An example how it is at any time possible to find and access the persistent auction site object created by the `MainCreation` application 4.1.2 is given in listing 4.1.3.

Example 4.1.3

```
public class MainSearch{
 public static void main(String[] args){
   ...
   //Search for the specific auction site(s)
   String description = "an example auction site";
   List auctionSites =
    $AuctionSite[/description={description}]$;
   AuctionSite theAuctionSite = null;
   //if there is at least one, get the first
   if(auctionSites.size()>0)
```

```
      theAuctionSite = (AuctionSite)auctionSites.get(0);

    //do anything with the auction site
    ...
  }
}
```

4.1.2 Deletion

Analagous to member variables of a transient object, Java member variables of databases are deleted implicitly. They are removed if no persistent object references them any longer. Contrastly, database objects have to be deleted explicitly, since they exist independently and can be seen as the entry point to persistent object trees. Any database declaration will implicitly extend the `Database` interface and implement its `delete` method. This happens at parse time and is explained in 4.2 in more detail. Invoking the `delete` method upon a database object means there will be an attempt to delete it. If the corresponding database object was deleted successfully, each succeeding access will cause a null pointer exception. An example demonstrating the different deletion types is given in 4.1.4.

Example 4.1.4

```
public class MainDeletion{
 public static void main(String[] args){
   ...
   AuctionSite site;
   ...
   //get a reference to the desired
   //auction site element
   ...
   if(site!=null){
    //implicitly deletes the previous description
    //of the auction site object
    site.setDescription("another description");

    //explicitly deletes the persistent
    //auction site database
    site.delete();

    //checks, if deletion was successful
    if(site==null){
     System.out.println("this auction site
       was successfully deleted");
    }else{
```

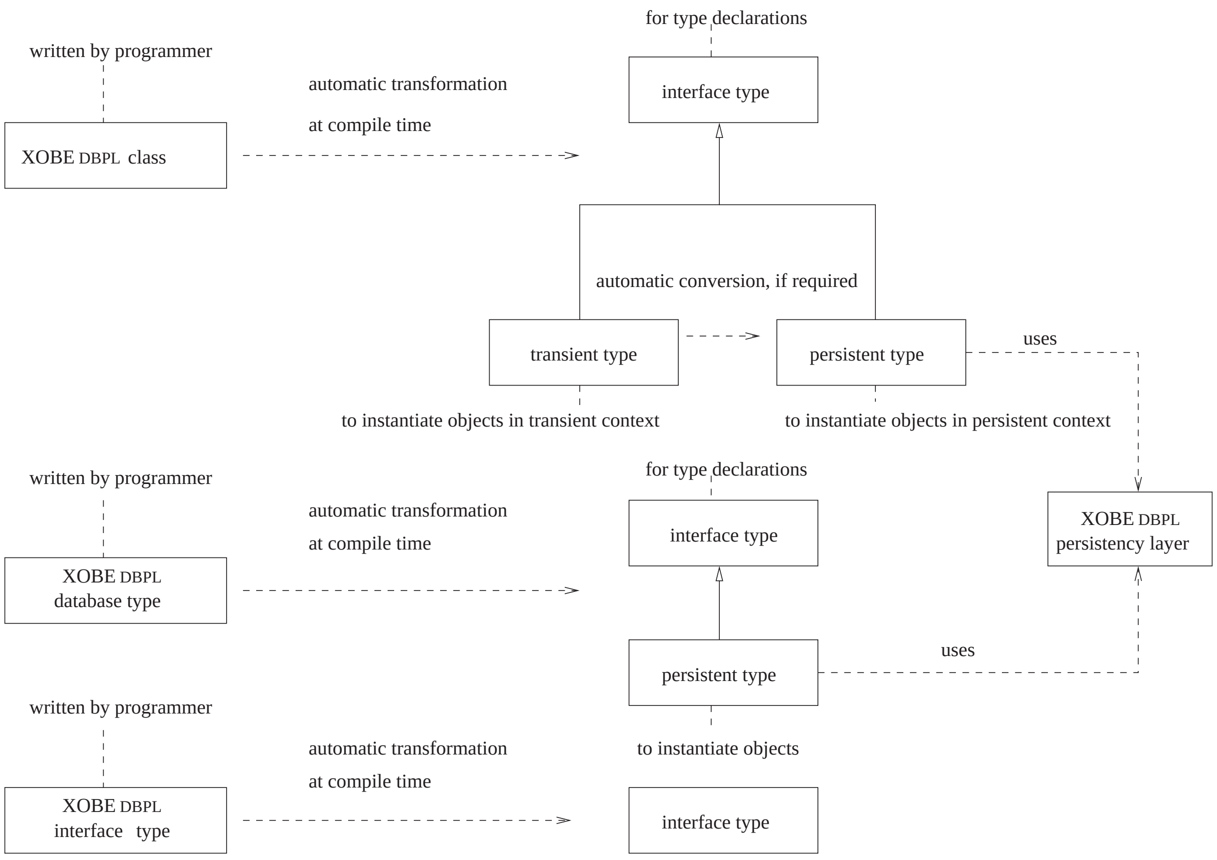
```
23        System.out.println("this auction site
24          could not be deleted");
25        System.out.println("another user might keep a lock");
26      }
27      ...
28    }
29  }
30 }
```

Line 12 shows how to delete a member variable of a database object. Line 16 presents an explicit delete invocation as it is necessary in case of a database object itself. Since the deletion of a persistent object is a special kind of write operation, it might not always succeed, e.g. line 20. It might happen that another application program holds a lock for this object. Transactions and concurrency are introduced in chapter 5. If a **XOBE**$_{\text{DBPL}}$ program is running in a single application environment each manipulating operation always succeeds including any deletes. This behaviour naturally changes if there are several applications that work with the same data and objects. Now that syntax and semantics are introduced the next section 4.2 will present corresponding realization concepts.

4.2 Realization Concepts

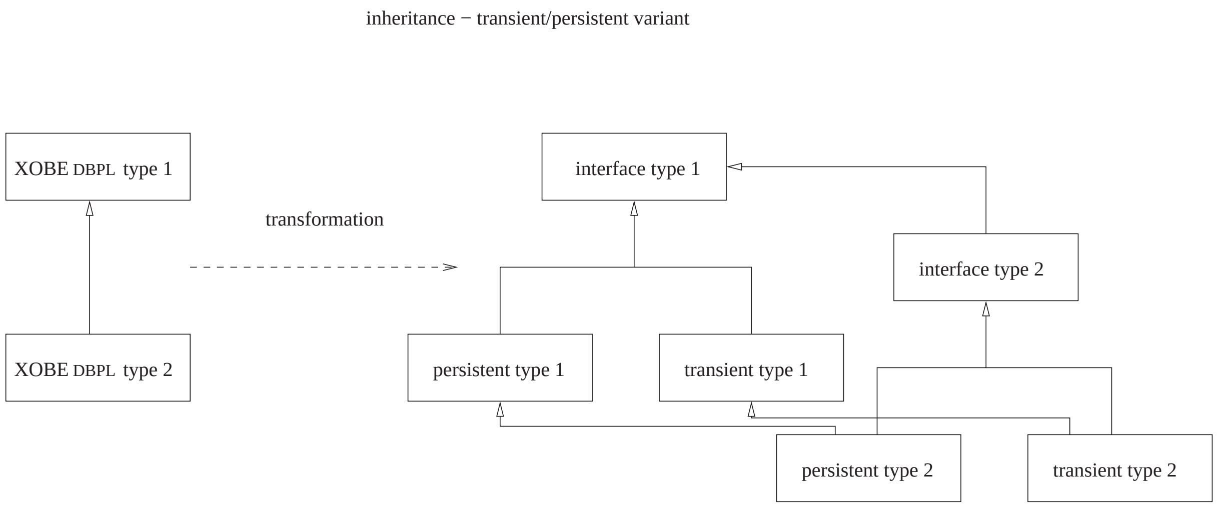
The basic concepts to realize the type independent, orthogonal and transparent persistency are shown in figure 4.1. Both generated variants implement a common interface containing all public non-static methods of the original **XOBE**$_{\text{DBPL}}$ type. As illustrated in figure 4.1 the interface type is used throughout the transformed code to declare types of local variables, formal parameters, return types and member variables. While the transient type is instantiated in transient environments, e.g. class declarations and methods, the persistent type is instantiated in persistent environments, e.g. constructors of databases. In case an instantiated transient object is assigned later on to a variable within a persistent environment, it is transformed into an equivalent persistent object. Detection and transformation into a persistent object happen automatically. The generation process of interface, transient and persistent types is done automatically at compile time. Implementation details are given in chapter 7. Furthermore, persistent variants have an associated `descriptor` class that is also generated automatically by the precompiler. A descriptor class is used later by the persistency layer to store objects of a given type efficiently and to avoid reflection. A descriptor class contains information about the inheritance structure, types and names of member variables. It is also important to notice that any persistent variant includes code to communicate with the persistency layer that is responsible for storing and updating persistent data in the background persistency layer. Chapter 6 focuses on the web-service-based persistency layer used in this work. **XOBE**$_{\text{DBPL}}$ database type declarations as well as class declarations can inherit from other classes or databases respectively. In particular, it is also possible that classes inherit from databases and vice versa.

Figure 4.1: Main persistency realization concepts



Inheritance , including those among interfaces, is supported quite intuitively and is illustrated by figure 4.2. The concepts presented so far stay the same and in addition, the generated interface of the subtype extends the corresponding interface of the supertype. This also holds for persistent and transient variants. In **XOBE**$_{\text{DBPL}}$ classes and databases can implement arbitrary interfaces. The generation process for the implementation structure is straightforward. The generated interfaces extend the interface generated from the original one. Finally let's look at our example database `AuctionSite` listed in 4.1.1 and see the resulting Java classes and interfaces of the transformation process that are given in figure 4.2. The auction site database is transformed into an interface `AuctionSiteInterface` that contains all public methods, e.g. `getDescription`, `setDescription` and `registerPerson`. As mentioned before, types of formal parameters and return types change into interface types, e.g. `String` becomes `XobeString` and XML types become `XMLObjectInterface`s. In this example the generated interface needs only to be implemented by the persistent variant `AuctionSitePersistent`. The persistent variant implements the `Database` in-

Figure 4.2: Inheritance

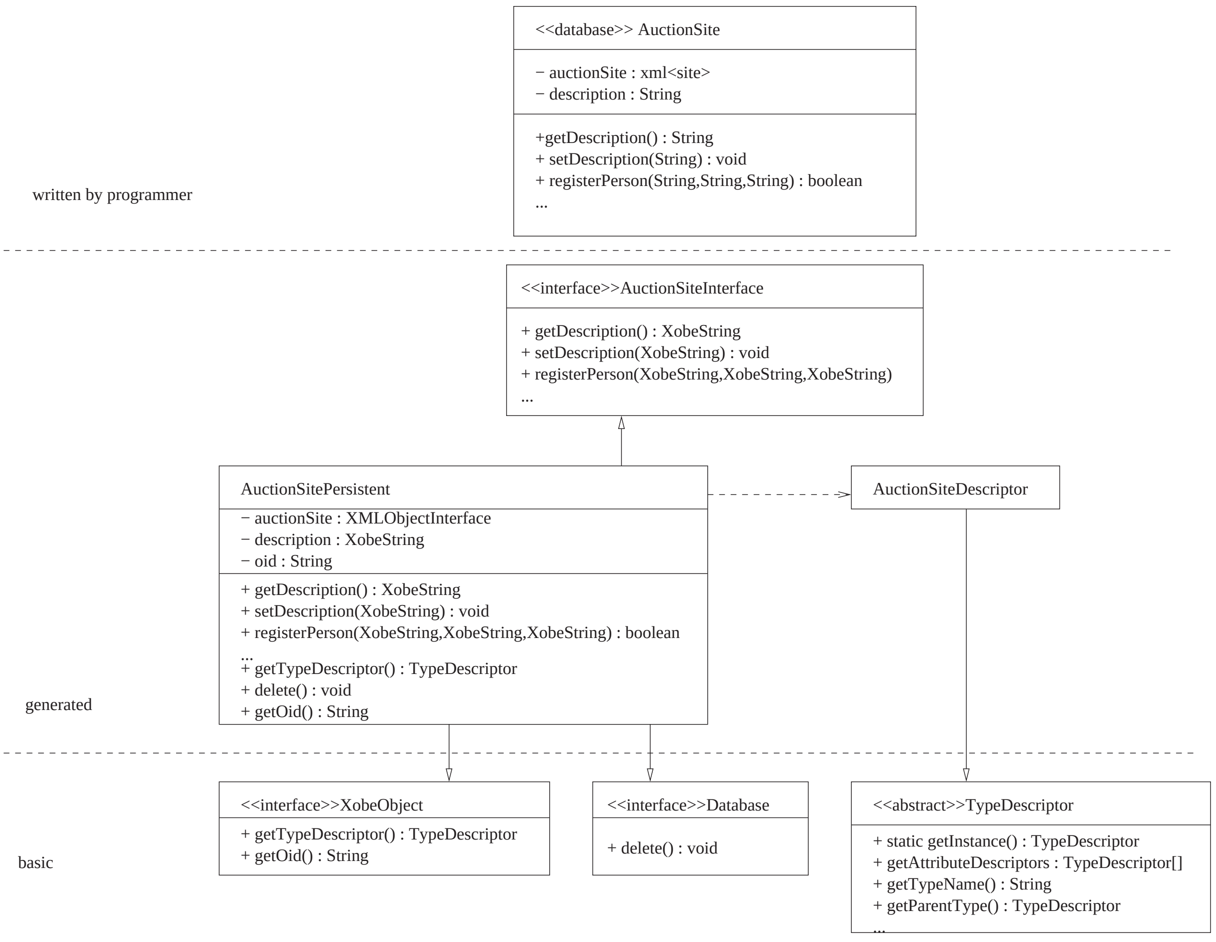


terface containing the required `delete` method and the `XobeObject` interface that implies among others an object id. Persistent objects in **XOBE**$_{\text{DBPL}}$ are given a unique identity. How these ids are generated is described in section 6.6. Another important aspect is the `getTypeDescriptor` method that returns a `TypeDescriptor`. As described before, this is used to avoid reflection and contains type information. Each persistent variant gets its own generated type descriptor, e.g. `AuctionSiteDescriptor`. The three exemplarily illustrated basic classes and interfaces are not generated each time a transformation process is executed. These classes belong to the core **XOBE**$_{\text{DBPL}}$ persistency classes and interfaces. Further information can be found in [74, 71].

4.2.1 Consistency

Since persistent variants contain code communicating with the **XOBE**$_{\text{DBPL}}$ persistency layer to load and write persistent data, they are stubs to the persistent objects. Persistent objects are manipulated by more than one application in general. Consequently, the aspect of client-side consistency must be taken into account and dealt with. A first approach in **XOBE**$_{\text{DBPL}}$ has been that every read or write operation upon a persistent object causes immediate communication with the persistency layer. This kind of client-side consistency is called strict consistency. Strict consistency suffers from enormous performance problems and is only considered theoretically. Consequently, **XOBE**$_{\text{DBPL}}$ has to use another, weaker client-side consistency model which is chosen to be the so-called `FIFO` consistency. In the context of FIFO consistency the following constraints must be kept. Write operations which are performed by a single application are seen in the order of their appearance by other applications. Contrarily, write operations

Figure 4.3: Transformation of the `AuctionSite` database



<<database>> AuctionSite
− auctionSite : xml<site>
− description : String
+getDescription() : String
+ setDescription(String) : void
+ registerPerson(String,String,String) : boolean
...
written by programmer
<<interface>>AuctionSiteInterface
+ getDescription() : XobeString
+ setDescription(XobeString) : void
+ registerPerson(XobeString,XobeString,XobeString)
...
AuctionSitePersistent
− auctionSite : XMLObjectInterface
− description : XobeString
− oid : String
+ getDescription() : XobeString
+ setDescription(XobeString) : void
+ registerPerson(XobeString,XobeString,XobeString) : boolean
...
+ getTypeDescriptor() : TypeDescriptor
+ delete() : void
+ getOid() : String
AuctionSiteDescriptor
generated
<<interface>>XobeObject
+ getTypeDescriptor() : TypeDescriptor
+ getOid() : String
<<interface>>Database
+ delete() : void
<<abstract>>TypeDescriptor
+ static getInstance() : TypeDescriptor
+ getAttributeDescriptors : TypeDescriptor[]
+ getTypeName() : String
+ getParentType() : TypeDescriptor
...
basic

which are performed by different applications may be seen in an arbitrary order by other applications. Therefore persistent objects might be copied to a cache on client-side. In **XOBE**$_{\text{DBPL}}$, code responsible to realize FIFO consistency is generated and inserted automatically. Transactions, which are part of the next chapter, are realized with additional synchronization when entering and leaving. Further consistency models can be found in [62]. An important topic in future work is to provide semi-automatic consistency to the programmer and thus improving performance. The programmer should then be able to mark program points where persistent objects are written to the persistency layer or respectively read from it.

As mentioned previously, tests with an example application and related approaches such as EJB and JDO are performed and evaluated in section 9.2. The next chapter 5 will introduce how concurrency and transaction concepts are realized and handled in **XOBE**$_{\text{DBPL}}$.

Chapter 5

Transactions

A concept which is closely connected to persistency, which was described in the last chapter, is transactions. Transactions are used to protect objects which might be used by more than one application. Transactions provide the possibility of mutual exclusion. Moreover, they allow processes to manipulate mulitiple objects within one atomic operation. A well known example demonstrating the need for transactions is a modern bank application. The application actualizes an online database. The customer can use the online bank application via a web browser and is also able to transfer money from one bank account to another. The whole operation consists of two steps. During the first step an amount `a` of money is taken from bank account 1. During the second step the amount `a` is paid into the bank account 2. If the connection fails between these two steps the money gets lost. This problem would not have occurred if the two basic operations were part of the same transaction. Basic operations in context of transactions are to mark the beginning of a transaction, to end the transaction and causing the committal of all changes, to abort a transaction causing any changes to be undone as well as to read and to write objects. In context of transactions the four characteristic qualities are defined: atomicity, consistency, isolation and durability. These properties are often abbreviated by `ACID` . Atomicity means that the execution of a transaction seems to be one basic operation from outside. Consistency means that the transaction does not break system invariants. Isolation implicates that concurrent transactions do not influence each other and finally durability means that changes made by a transaction are persistent after the commit process. There are a lot of different transaction types. The most common and straightforward type is called a `flat` transaction . A `flat` transaction consists of a sequence of operations keeping the ACID properties. Other types include for example nested as well as distributed transactions. These are discussed in context of **XOBE**$_{\text{DBPL}}$ in the following sections. Further types are described in detail in [28]. Most related approaches, like EJB, if at all, only support flat or respectively distributed transactions.

5.1 Syntax and Semantics

The syntax for transactions in **XOBE**$_{\text{DBPL}}$ is defined by the grammar given in definition 5.1.1.

Definition 5.1.1 Transactions in **XOBE**$_{\text{DBPL}}$ are constructed according to the following grammar:

statement	$\rightarrow$	*block*
		| ...
		| *synchronized_statement*
		| *transaction_statement*
		| *try_statement*
transaction_statement	$\rightarrow$	**transaction**
		’(’
		name_list
		’)’
		block

□

As can be seen a transaction in **XOBE**$_{\text{DBPL}}$ is defined analogously to a try or synchronized statement in Java. A transaction statement also consists of a `name_list`. Within a name list the programmer has to list all database objects supposed to be part of the transaction. If an object occurs in this list the sequence of operations within the corresponding transaction block will maintain the ACID properties.

Listing 5.1 shows an example of a transaction block. The `bid`’s method body without a transaction block can also be found in 3.3.

Listing 5.1: An example transaction

```
...
current new_current = null;
transaction(auctionSite){
 //calculate new current
 xml<current*> cur =
   $auctionSite//open_auction[/@id={a_id}]/current$;
       current new_current =
  <current>{cur.itemAsInt(0)+incr}</current>;

 //create new bidder
       bidder bid = <bidder>
                        <date>{getDate()}</date>
                        <time>{getTime()}</time>
                        <personref person={p_id}/>
                        <increase>{incr}</increase>
16                          </bidder>;
17
18   //update auction
19   $LET i := auctionSite//open_auction[/@id={a_id}]
20   UPDATE i INSERT {bid} BEFORE /current,
21   REPLACE /current WITH {new_current}$;
22  }
23  ...
```

If applications use the method `bid` concurrently it might happen that the two operations, in particular *calculate the new current* and *update the auction site*, might be divided and could not be executed as desired. Let us imagine two processes are both starting to execute the `bid` method and are both reading for example a current amount of `100`. The first process continues and increases the bid to 150. The second process wants to increase the bid by 10. Since it still has got the old amount, it actualizes the bid to 110, which an observer notices as an illegal decreasing of the bid. By enclosing the two operations reading and updating within the transaction block in line 3 and 22 this problem can no longer occur. Please notice that the Java `synchronized` environment will not work if the processes do not operate on the same Java Virtual Machine.

As described before, a transaction comes with five basic operations which are `begin`, `end`, `write`, `read` and `abort`. The beginning and ending of a transaction is achieved by the opening and closing brackets of the transaction block. Write and read operations are then method invocations upon persistent objects registered to the transactional environment by the corresponding name list. Finally, an abort will be generated and will throw a transaction abort runtime exception (`TransactionAbortException`), if the runtime environment cannot successfully execute and commit the transaction. All transactional operations happened so far are rolled back. The program execution continues after the transaction block. Further actions in particular if the transaction is supposed to be tried again depend on the programmer. Let us take the `bid` example one more time. Now we want to catch a possible transaction abort exception and print a suitable error message. The resulting code is given in listing 5.2.

Listing 5.2: Catching the abort of a transaction

```
 1  ...
 2  try{
 3  transaction(auctionSite){
 4   //calculate new current
     ...
10   //create new bidder
     ...
18   //update auction
     ...
22  }
23  catch(TransactionAbortException e){
    System.err.println("... bid failed ...");
  }
  ...
```

Up to now we have only discussed flat transactions meaning that the transaction block only consists, for example of a sequence of method invocations. Indeed as can be seen from the grammar in 5.1.1 it is also possible to construct nested transaction statements. This possibility and the corresponding semantics are discussed in the following section. The transformation of transactions in **XOBE**$_{\text{DBPL}}$ into pure Java code is explained in section 7.2.2.

5.2 Classification of Transactions

There are quite a lot of different transaction types. In the context of **XOBE**$_{\text{DBPL}}$ flat, nested and distributed transactions are discussed and supported. More details about different transaction types can be found in [28].

As explained before, flat transactions consist of a sequence of flat operations. The most important limitation of flat transactions is that it is not possible to commit or abort partial results. If a transaction is a long sequence of operations and an interruption becomes more likely this will be a serious problem. An example of a flat transaction is dicussed in the previous section.

In contrast to flat transactions nested transactions try to overcome exactly these limitations. Instead of one very long flat sequence of transactional operations nested transactions provide the possibility to summarize logically separated parts of the whole transaction to separated nested ones. These logically separated units are then transactions within transactions. These subtransactions can be executed in parallel, perhaps on different hosts. Nested transactions implicate a subtile new problem. Let us imagine that all subtransactions are committed successfully, but the enclosing super transaction is rolled back. Consequently the subtransactions have to be undone. Thus, durability is only true for the enclosing supertransaction. Nested transactions demand a substantial administrative effort, but their semantics is rather intuitive. In **XOBE**$_{\text{DBPL}}$ nested transactions can be formulated by nesting several transaction blocks.

In general nested transactions are provided to support the logical separation of the original transaction. But the logical separation of a nested transaction does not automatically mean that the objects of one partial transaction are located suitably. Consequently, a flat transaction operating on objects or data that are distributed on different hosts is also called a distributed transaction . The difference between nested and distribute transactions is subtle and important. A nested transaction is separated logically in sub transactions. A distributed transaction is a logic flat transaction operating on distributed data. Therefore the **XOBE**$_{\text{DBPL}}$ programmer cannot explicitly formulate distributed transactions. One main concept of **XOBE**$_{\text{DBPL}}$ is transparency . According to the definition of a distributed transaction it depends on the underlying persistency layer whether a transaction is executed in a distributed manner. The persistency layer in **XOBE**$_{\text{DBPL}}$ is realized with the help of a web service which is introduced in chapter 6.

5.2.1 Realization Concepts

In principal there are two common approaches how transactions might be realized. A detailed description can be found in [88]. On the one hand transactions could be realized by a private workspace . The private workspace concept is used for example in Java Spaces [82]. Private workpaces are also suitable for distributed transactions. A transactional process on each host gets its own private workspace where the data is copied the process works with. Each read and write operation of the transactional process is then performed within the private workspace. Finally, the updates of the private workspace are either written through as a whole or simply discarded.

A second common approach is called the `Writeahead Protocol`. With this approach data is actually manipulated. But before the manipulation operation starts the change is registered as an entry in a protocol. The entry consists of the corresponding transaction identification, the corresponding object or data identification as well as the new and the old value. If the transaction is successfully committed and the transaction is a none subsidiary one the entries can be removed and nothing else has to be done anymore. If commiting the transaction fails all corresponding changes can be undone by analyzing the protocol entries. In **XOBE**$_{\text{DBPL}}$ the private workspace approach is realized since it is well suited in case of distributed transacions and has got less overhead in case a transaction is aborted. The protocol is implemented as part of the **XOBE**$_{\text{DBPL}}$ web service discussed in chapter 6.

5.3 Concurrency Control

So far it is possible to realize atomicity and durability. The remaining properties consistency and isolation are achieved by controlling the execution of transactions running in parallel. Consistency and isolation are guaranteed if the transactions access data objects in a certain order. The order must be equivalent to an order that occurs when executing these transactions one after another. This quality is also called serializability. In a distributed system each component has got its own transaction manager, scheduler and data manager. Scheduler and data manager are both responsible for keeping the data consistent. The scheduler is responsible for locking and releasing operations or respectively for time stamping operations. The data manager performs read and write operations. Finally the transaction manager initializes and ends a transaction. Each transaction is managed by a single transaction manager which in turn communicates with the schedulers of the corresponding components. A transaction's responsible transaction manager is also called the master. The de facto standard for the basic distributed transaction protocol is the `centralized two-phase commit protocol` first defined in [29]. Here, the master is acting as the commit coordinator. This protocol is used in conjunction with different concurrency control algorithms. If data isn't replicated, the protocol works as follows. When a subtransaction manager finishes executing its part of the distributed transaction, it sends an execution complete message to the master. When the master has received such a message from each corresponding subtransaction manager, it will initiate the commit protocol by sending the `prepare to commit` messages to all subtransaction managers. If a subtransaction manager is able to commit, it sends a `prepared` message back to the master. The master will send `commit` messages to each subtransaction manager after receiving prepared messaged from all subtransaction managers. The protocol ends with the master receiving `committed` messages from each of the subtransaction managers. If any of them is unable to commit, it will return a `cannot commit` message instead of a `prepared` message in the first phase. This will cause the master to send `abort` instead of the `commit` messages in the second phase of the protocol. A modified version of the two-phase commit protocol is available in case of redundant data.

5.3.1 Realization Concepts

The oldest known algorithm to control concurrency are locks . A well known algorithm is in particular the `two-phase-locking` algorithm. Details can be found in [88] for example. During the first phase the scheduler tries to get all required locks and during the second phase these locks are released again. Applying the `two-phase-locking` algorithm may result in deadlocks. Deadlocks may be detected by time outs. The algorithm is also adaptable to distributed systems.

The approach which is chosen in **XOBE**$_{\text{DBPL}}$ is called the distributed, timestamp-based, optimistic concurrency control algorithm (DTO) [75] . This approach is very different from a locking algorithm. The algorithm operates by exchanging certification information during the commit protocol. For each data item, a read timestamp and a write timestamp are maintained. Transaction may read and update data items freely. Updates are stored into a local, private workspace until commit time. Additionally, for each read the transaction must remember the version identifier, which could be the write timestamp, accociated with the item when it was read. Transactions are assigned a globally unique timestamp, e.g. using the Lamport algorithm [48] . This timestamp is sent in particular in the prepare to commit message and is used to locally certify all read and write operations as follows. A read request is valid if the version that was read is still the current version and no write with a newer timestamp has already been locally certified. A write request is valid and certified if no later reads have been certified and subsequently committed and no later writes have already been locally certified. The term later refers to the timestamp of the transaction. It is required that these local verification computations are performed within in a critical section. If all of the operations within the private workspace are valid, a prepared response is sent back to the master. The corresponding data items need to be locked until either an abort or a commit message is received. Then updates are either performed upon the real data or simply discarded. Since all items that need to be locked are known in advance, deadlocks cannot occur. According to [15] the performance of the DTO algorithm compared to the two-phase-locking algorithm is similar. Moreover, DTO has got less communication overhead using only the messages of the two-phase commit protocol. With an increasing number of involved subtransaction managers the likelihood of restarts for the DTO algorithm increases. The DTO algorithm always uses restarts to handle conflicts, checking for problems only when a transaction is ready to commit. The algorithm is implemented and realized within the **XOBE**$_{\text{DBPL}}$ web service persistency layer discussed in 6.

Chapter 6

Web Service for Distributed Persistent Objects

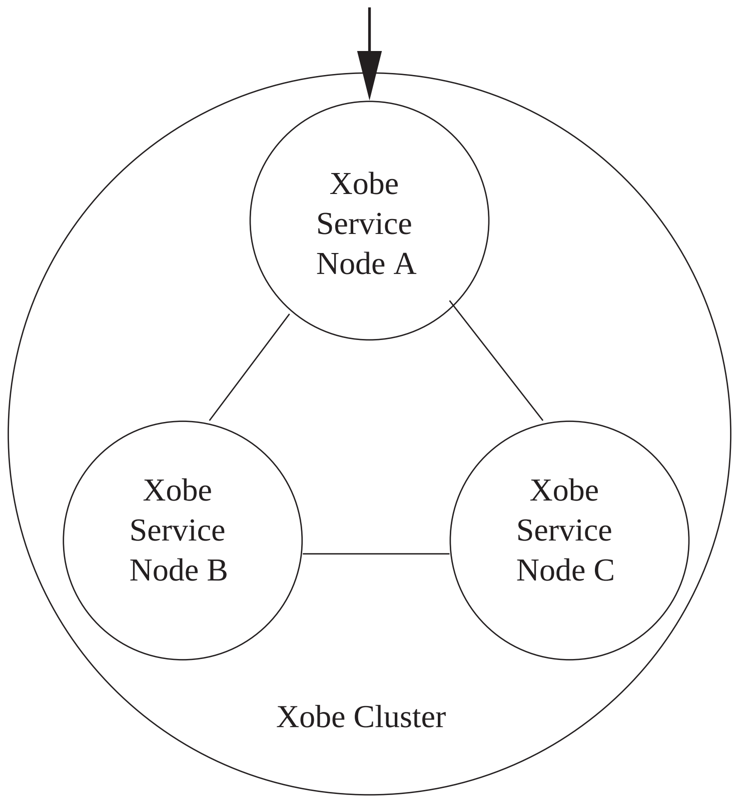
In previous chapters the transparent and type independent persistency concepts as well as transactions in **XOBE**$_{\text{DBPL}}$ are introduced. This chapter explains the realization of the underlying persistency layer used in **XOBE**$_{\text{DBPL}}$. In **XOBE**$_{\text{DBPL}}$ persistent objects are supposed to be distributed and shared among and with other applications. Since these applications should not be restricted to a certain programming language like **XOBE**$_{\text{DBPL}}$, objects and type information have to be stored in a neutral manner. A suitable approach is to store objects and types semi-structured, e.g. using XML. Thus, the persistency layer which is supposed to be transparent, distributed, programming language neutral and supporting object-oriented data best is developed and realized based on web services and therefore XML (SOAP) communication. Moreover, the persistency layer in **XOBE**$_{\text{DBPL}}$ has not only got an open interface to clients, e.g. programming languages, but also is not restricted to certain data storages. The persistency layer is designed so that arbitrary database paradigms, e.g. relational, object-oriented, XML, or even file systems can be integrated with minimum effort. The persistency layer offers a transparent consistent virtual view of distributed persistent data. In **XOBE**$_{\text{DBPL}}$ the persistency layer is totally hidden within the **XOBE**$_{\text{DBPL}}$ runtime environment and fully transparent to the **XOBE**$_{\text{DBPL}}$ programmer. Code communicating with the persistency layer is automatically added into the persistent class variants at compile time as described in chapter 4. The persistency layer introduced in the following sections has also become an independent project.

6.1 Architecture

This section describes the conceptual architecture used to realize the **XOBE**$_{\text{DBPL}}$ web service . A client connects to a single **XOBE**$_{\text{DBPL}}$ web service instance which is part of a component called **XOBE** service node (`XSN`). A **XOBE** service node is part of a network containing one or more other nodes. Figure 6.1 presents an example **XOBE** cluster consisting of three nodes.

Directed arrows within the picture indicate the direction of communication. Thick arrows represent communication of the component as a whole. Respectively they represent incoming and outgoing interfaces. A **XOBE** cluster can be interpreted as a `persistency grid`. Conse-

Figure 6.1: A **XOBE** cluster consisting of three **XOBE** service nodes

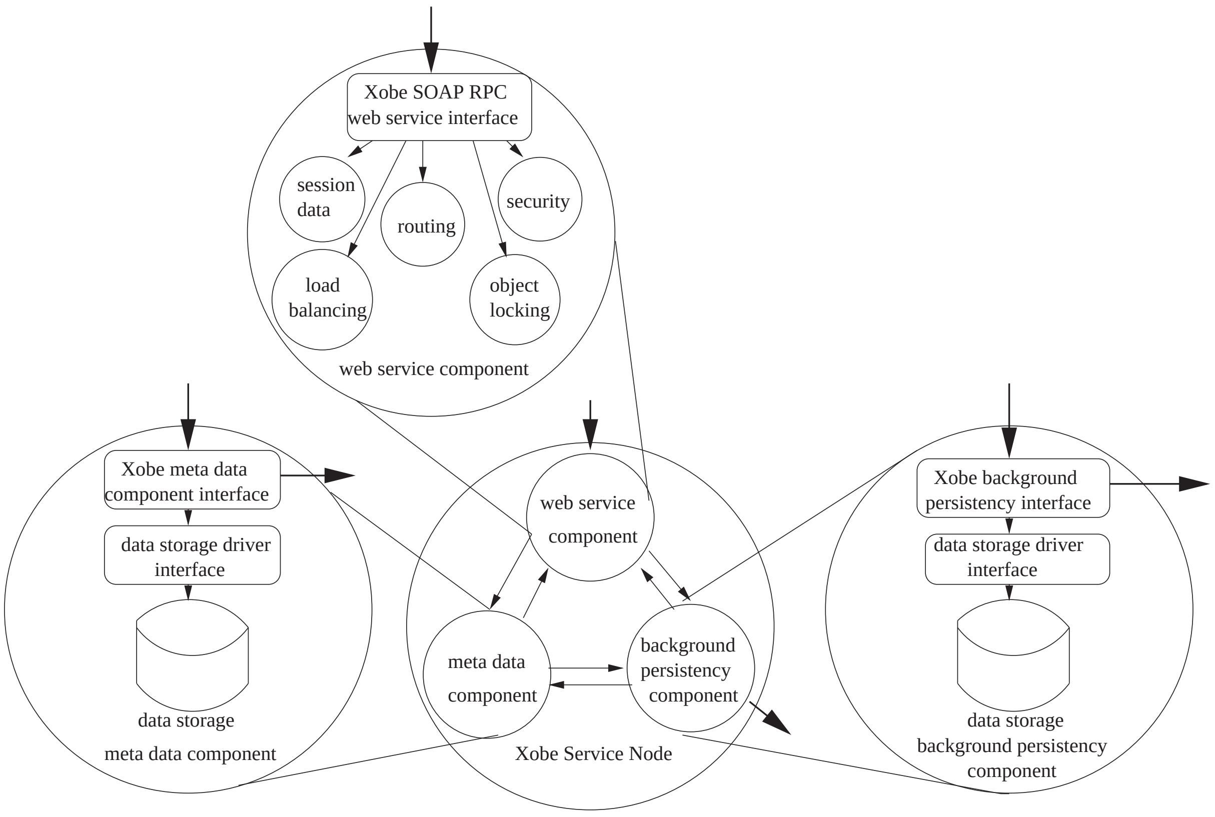


quently communication and organization aspects can be taken from grids as described in [25] and [24].

A single **XOBE** service node consists of at least three components: one web service component, one meta data component and one or more background persistency components. Incoming and outgoing communication of a **XOBE** service node is done via SOAP. Communication within the node itself is done in the specific implementation language. The architecture of these three characteristic parts of a **XOBE** service node and the **XOBE** service node itself are illustrated in figure 6.2.

The most important component is indeed the so-called web service component . This component is responsible for most of the tasks. The web service component processes incoming messages and queries. These messages are then passed to the several background persistency components. Moreover, any changes concerning, for example types, transactions or users are

Figure 6.2: A **XOBE** service node and its three main components


Xobe SOAP RPC web service interface
session data
routing
security
load balancing
object locking
web service component
Xobe meta data component interface
data storage driver interface
data storage
meta data component
web service component
meta data component
background persistency component
Xobe Service Node
Xobe background persistency interface
data storage driver interface
data storage
background persistency component

reported to the meta data component. Authentification, session data, object locking in case of transactions and security aspects, routing and load balancing are managed by the web service component as well.

The meta data component is responsible for keeping and offering all the persistent administration data of the **XOBE** service node. The corresponding data includes, among others, registration data about objects and types stored in this node, about available and registered background persistency components of this node and about available applications and users. The data storage is a common database system.

The background persistency component is nearly analogous to the meta data component. Instead of meta data the concrete object data are stored within its data storage. In contrast to the meta data component, this time the data storage could also be another **XOBE** cluster. The architecture of the background persistency component allows to integrate and use arbitrary storage paradigms, e.g. relational databases, XML databases, file systems. An object is always kept as a whole in one component meaning that all atomic attributes and parent objects are located at the same component. Complex typed attributes could also be located on different components. This component can use an arbitrary data storage as long as a driver in the specific programming language exists. Different data storage types are suitable for different data types. For example an XML database stores XML objects more efficiently than a relational database.

Separated **XOBE** clusters can be fusioned by defining bridge nodes. These are normal **XOBE** service nodes which are part of each **XOBE** cluster.

At the moment objects are kept without redundancy.

6.2 Interface

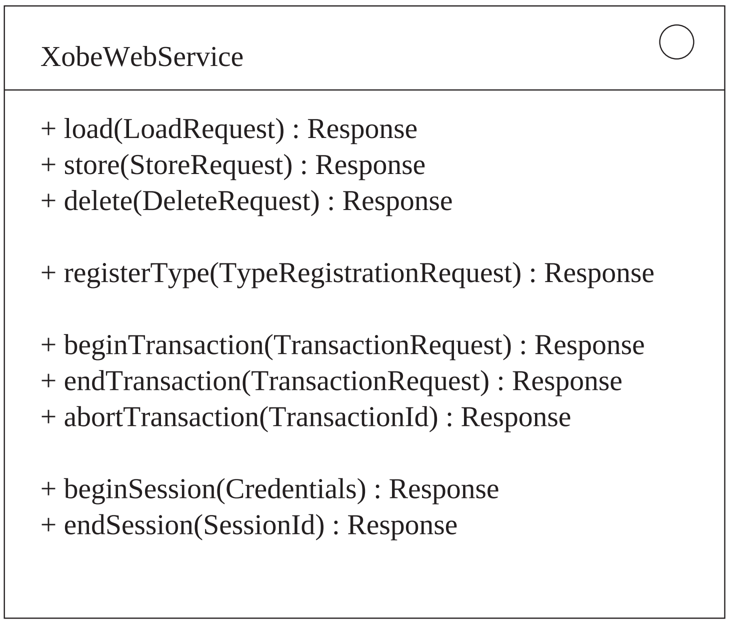
The **XOBE**$_{\text{DBPL}}$ web service interface offers methods for all essential tasks including retrieving persistent objects, storing or respectively updating an object and deletions. Moreover, there are a few more methods solving tasks arising in the context of the former operations. The interface is kept thin and generic. Since the web service is completely integrated into the **XOBE**$_{\text{DBPL}}$ runtime environment and requests are formulated as well as responses processed automatically, the interface must provide methods for being capable to deal with objects of **any possible type**. Furthermore, object references are represented with the help of IDs. The web service's clients in the **XOBE**$_{\text{DBPL}}$ runtime environment are so-called **XOBE** local servers encapsulating communication with running **XOBE**$_{\text{DBPL}}$ programs. Only the latter are written by the programmer. The communication code is generated and added at compile time. The web service itself is completely hidden from the programmer.

Figure 6.3 lists the available web service methods which are predominantly `load` to retrieve persistent objects, `store` to store or respectively update a persistent object and `delete` to delete a persistent object.

If a store request for an object of an unknown type occurs, the web service needs to know its structure. With the help of the method `registerType` the client can register this new type. A second important group of methods is provided for transactional support,
e.g. `beginTransaction`, `endTransaction` and `abortTransaction`. Finally, the last group consists of the methods `beginSession` and `endSession` providing the functionality to register (**XOBE**$_{\text{DBPL}}$) clients to the web service. Aspects concerning transactions and sessions are explained in section 6.6. Besides defining the available methods of the web servcice WSDL also defines parameter and return types that are understandable by the web service. The **XOBE**$_{\text{DBPL}}$ web service must be able to describe objects of any possible type. Accordingly the web service needs to know the super type `generic object`. The following grammar shows how arbitrary objects are represented and sent to the web service .

Definition 6.2.1 Objects of an object-oriented programming language have to be mapped to the following representation to be processable by the web service:

Figure 6.3: Interface of the **XOBE**$_{\text{DBPL}}$ web service


XobeWebService

+ load(LoadRequest) : Response
+ store(StoreRequest) : Response
+ delete(DeleteRequest) : Response

+ registerType(TypeRegistrationRequest) : Response

+ beginTransaction(TransactionRequest) : Response
+ endTransaction(TransactionRequest) : Response
+ abortTransaction(TransactionId) : Response

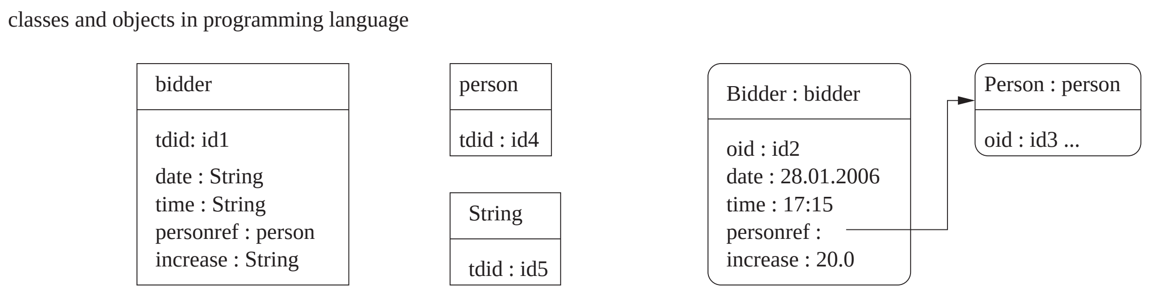
+ beginSession(Credentials) : Response
+ endSession(SessionId) : Response

data_object	→	*object_id type_descriptor_id* (*attribute_name*)∗ (*attribute*)∗
		|
		atomic
		|
		data_reference
attribute	→	*data_object*
data_reference	→	*object_id type_descriptor_id*
		|
		null
atomic	→	*string* | *integer* | ...

The terminal **null** stands for the empty reference pointing to no object. The nonterminals *object_id*, *type_descriptor_id* and *attribute_name* are mapped to a character sequence. An atomic value represents common basic types like string or integer. □

The optional sequence of attribute names allows to submit an object update. In case only some or a single attribute value of an object has changed only these new changed values have to be submitted followed by their corresponding attribute names. An example presenting the representation of **XOBE**$_{\text{DBPL}}$ classes and objects for the communication with the web service based persistency layer is given in 6.4. The representation according to the grammars in 6.2.1 and 6.2.2 is illustrated with XML tags. In figure 6.5 an object update can be seen. In the example the `increase` attribute's value is set to `40.0`. Since an object of type `bidder` has more than one attribute, transmitting solely an update of a subset of its attributes forces to add the corresponding attribute name(s), e.g. `increase`.

Figure 6.4: Representation of an object and its class



representation for the communication with the web service based persistency layer

```
<type_descriptor>
  <type_descriptor_id>id1</type_descriptor_id>
  <type_name>bidder</type_name>
  <application_id>...</application_id>
  <attribute_declaration>
   <attribute_name>date</attribute_name>
   <type_descriptor_id>id5</type_descriptor_id>
   <multiplicity>optional</multiplicity>
   <association>composition</association>
  </attribute_declaration>
  ...
  <attribute_declaration
   <attribute_name>personref</attribute_name>
   <type_descriptor_id>id4</type_descriptor_id>
   <multiplicity>optional</multiplicity>
   <association>aggregation</association>
  </attribute_declaration>
  ...
 </type_descriptor>
```

```
<data_object>
 <object_id>id2</object_id>
 <type_descriptor_id>
   id1
 </type_descriptor_id>
<attribute>   <data_object>
    <atomic>28.01.2006</atomic>
  </data_object>   </attribute>
 <attribute>   <data_object>
   <atomic>17:15</atomic>
  </data_object>   </attribute>
   <attribute>    <data_object>
     <data_reference>
          <object_id>id3</object_id>
          <type_descriptor_id>id4</type_descriptor_id>
      </data_reference>
   </data_object></attribute>
    <attribute><data_object>
          <atomic>20.0</atomic>
    </data_object></attribute>
</data_object>
```

Figure 6.5: Representation of an object update

```
<data_object>
 <object_id>id2</object_id>
  <type_descriptor_id>id1</type_descriptor_id>
  <attribute_name>increase</attribute_name>
  <attribute><data_object>
   <atomic>40.0</atomic>
  </data_object></attribute>
</data_object>
```

The type of an object is given by the object's type descriptor ID. This ID uniquely references a type descriptor object. A type description, in object-oriented programming language the class of an object, is mapped to its representation according to the following grammar. The resulting representation is understandable by the **XOBE**$_{\text{DBPL}}$ web service .

Definition 6.2.2 Types or classes of an object-oriented programming language have to be mapped to the following representation to be processable by the web service:

type_descriptor	$\longrightarrow$	*type_descriptor_id type_name application_id*
		parent_type_descriptor_id? *attribute_declaration*$*$
parent_type_descriptor_id	$\longrightarrow$	*type_descriptor_id*
attribute_declaration	$\longrightarrow$	*attribute_name type_descriptor_id*
		multiplicity association
multiplicity	$\longrightarrow$	**single_valued**
		|
		multi_valued
		|
		optional
		|
		bound
association	$\longrightarrow$	**composition**
		|
		aggregation

The nonterminals *type_descriptor_id*, *attribute_name*, *type_name* and *application_id* are mapped to character sequences. □

An application ID uniquely identifies the application this type belongs to. The multiplicity of an attribute defines its cardinality. The default value is optional which means the attribute either references a single value or none. Multivalued means that an attribute references a list of values and single valued that this attribute has to reference one non null value. With the help of the bounded multiplicity attributes referencing arrays can be represented.

The association feature of an attribute is very important in the context of deleting persistent objects. If an attribute object is marked as composition, it will be destroyed with its parent object. Contrastly an object referenced by an aggregation marked attribute exists independently and is not deleted with its referencing object.

If a type is an extension of another type the parent type descriptor ID has to be set. The parent type descriptor ID then references the corresponding type in the inheritance hierarchy.

Every complex data object as well as types need a unique ID that is generated in a decentralized manner. More information about implementing the generation of IDs is given in section 6.6.

Any **XOBE**$_{\text{DBPL}}$ object is transformed into a data object automatically. Type descriptions for the web service are generated for every **XOBE**$_{\text{DBPL}}$ class at compile time.

The **XOBE**$_{\text{DBPL}}$ web service interface is designed according to a request/response system. Requests are solely formulated on client side. The operations offered by the web service are available as local as well as global variants. In the first case a request is answered or processed only by the directly connected **XOBE** service node. In the latter case the request is additionally passed to all known other **XOBE** service nodes of the corresponding cluster. Global requests are more time consuming than local ones, on the other hand the results of global requests might be a super set.

In general requests contain an identification ID, a timestamp and hop entries to detect circles and for routing decisions. Every request a client sends results in a response indicating if the requested operation failed or could be successfully performed. In the following subsections some details about the most important operations and their parameters and possible results are given.

6.3 Retrieving Objects

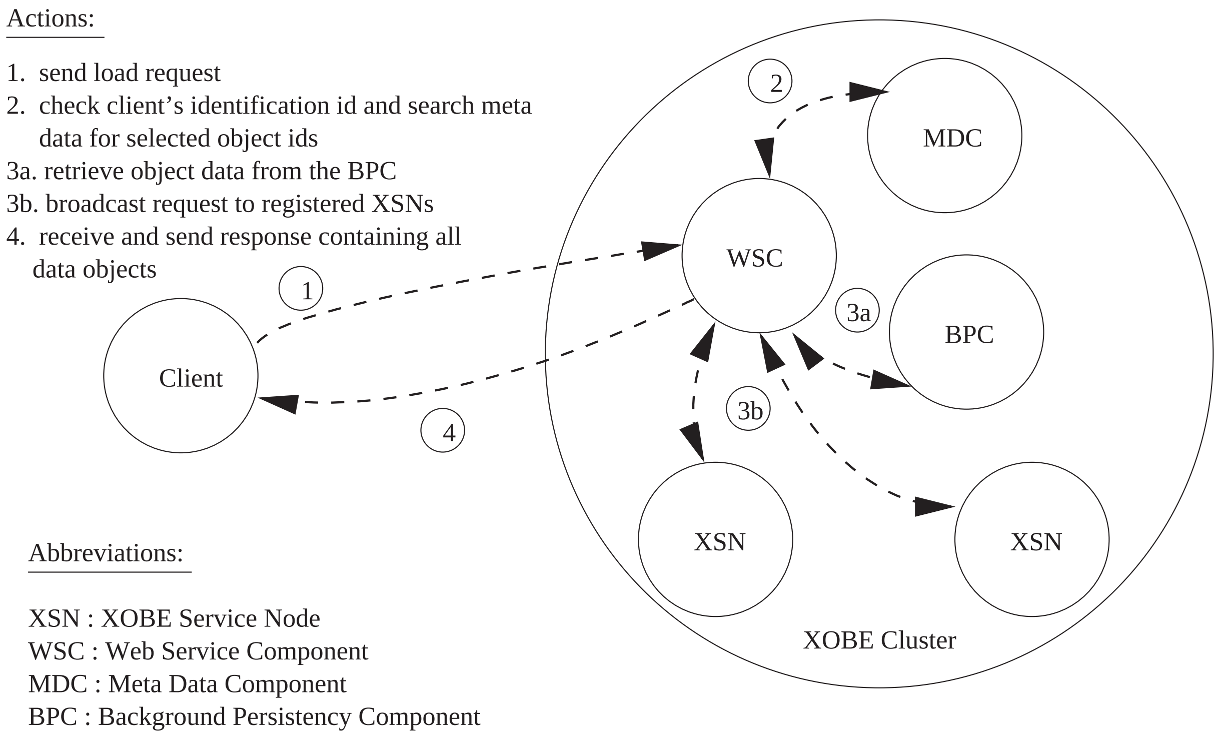
To retrieve persistent objects it is necessary to define a selection language . The selection language used in case of the **XOBE**$_{\mathrm{DBPL}}$ web service provides an object-oriented selection of data objects as well as meta data objects. The language's structure is similar to a predicate logic. Consequently, any other selection language, i.e. XPath, can be translated or mapped to this selection language.

The **XOBE**$_{\mathrm{DBPL}}$ web service receives a load request for either a single object or a set of objects. A load request consists of the client's session ID and a selection expression. The web service component checks the validity of the session ID and continues to evaluate the selection expression. A selection expression may for example consist of an object or type ID. With the help of the corresponding meta data the web service tests if the requested objects are available locally. Otherwise and in case of the global variant the request is broadcast to **XOBE** service nodes located in the corresponding **XOBE** cluster. If an ID is registered to a **XOBE** service node's meta data component a shallow copy of this object is loaded from the background persistency component. Finally, the **XOBE**$_{\mathrm{DBPL}}$ web service sends those shallow objects via SOAP as response to the client. The client can either be a **XOBE** service node or a **XOBE**$_{\mathrm{DBPL}}$ local server. The loading process is illustrated in figure 6.6. If no object or type is selected by the load request a no objects found response is sent back to the client.

6.4 Storing Objects

The store operation offered by the web service enables to store new objects as well as to update existing objects . The **XOBE**$_{\mathrm{DBPL}}$ web service client sends the data object, having to be kept persistently in connection with an identification ID . An identification ID can either be a session ID, which is sufficient for simple transactions, or a transaction ID, which is needed to perform complex transactions. Simple transactions are those that consist of a single operation. After checking the identification ID the web service component has to decide where to store or respectively update the object data. In case the data object is stored for the first time a suitable heuristic might be that the **XOBE** service node contacts its background persistency component and stores the data locally. Otherwise the object data is stored on the corresponding original web service's **XOBE** node. A modified strategy could move object data from one **XOBE** node to another if it turns out that this host is more suitable, e.g. its directly connected **XOBE**$_{\mathrm{DBPL}}$ local server works more frequently with this object data. The storing process described so far can be seen in figure 6.7. A client application can only use the web-service-based persistency layer if it has started a session. The directly connected **XOBE** node stores the corresponding session information within its meta data component, e.g. which object IDs are loaded. An object update can then be identified assuming that a corresponding load operation needs to be registered before. Please notice that up to now it is assumed that the **XOBE** service node or more precisely its meta data component already knows the object's type and its structure. In

Figure 6.6: Loading objects



Actions:

1. send load request
2. check client's identification id and search meta data for selected object ids

3a. retrieve object data from the BPC
3b. broadcast request to registered XSNs

4. receive and send response containing all data objects

Abbreviations:

XSN : XOBE Service Node
WSC : Web Service Component
MDC : Meta Data Component
BPC : Background Persistency Component

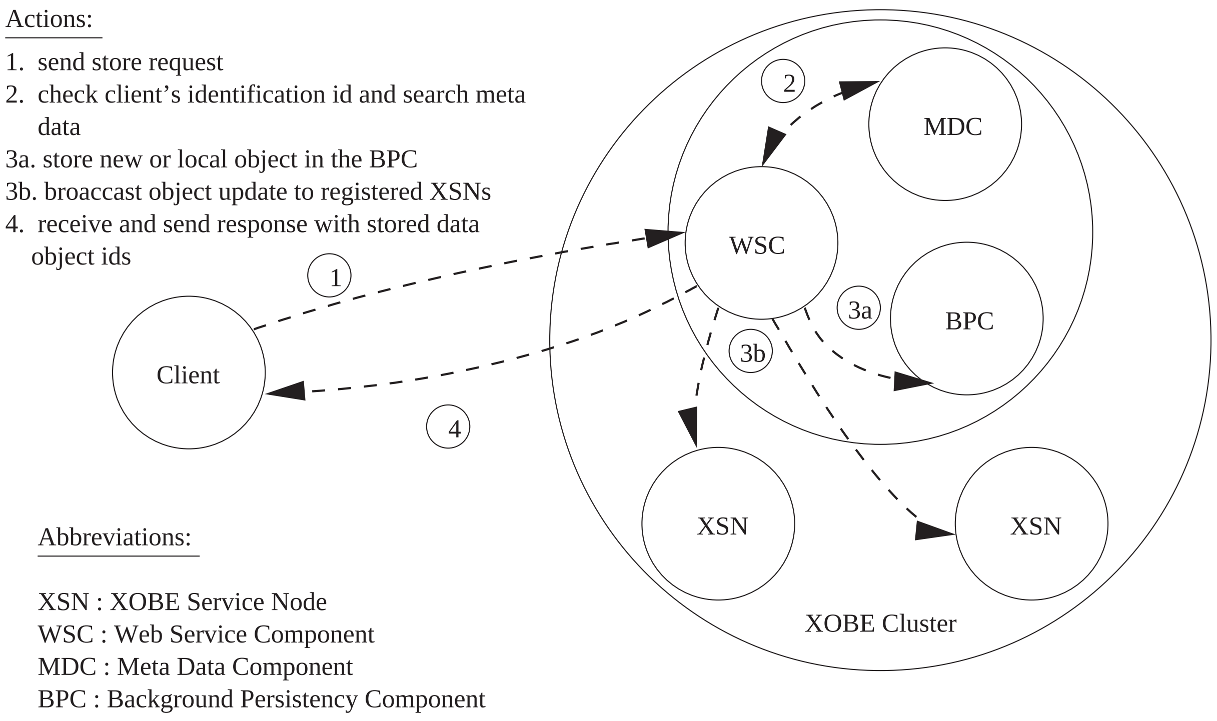
this case the **XOBE**$_{\text{DBPL}}$ client receives a store done response. If the requested store operation fails, an error response will be sent. For example in context of an unknown type it will be an unknown type response. If the object's type is unknown a second operation is needed which is described in the next section.

6.4.1 Registering Types

So far we have assumed that the web service which is going to store a new object already knows how to do this, e.g. that it knows the structure or more precisely the type. Hence, if an object of a new type is stored the web service's response asks for the corresponding type descriptor. In this case the web service's method called `registerType` can be used by the client to answer . Once again the client can either be another **XOBE** service node or a **XOBE**$_{\text{DBPL}}$ local server instance.

This storing process can be seen in figure 6.8.

Figure 6.7: Storing a data object with known type in single transaction mode


Actions:
1. send store request
2. check client's identification id and search meta data
3a. store new or local object in the BPC
3b. broaccast object update to registered XSNs
4. receive and send response with stored data object ids
Client
1
2
MDC
WSC
3a
BPC
3b
4
XSN
XSN
XOBE Cluster
Abbreviations:
XSN : XOBE Service Node
WSC : Web Service Component
MDC : Meta Data Component
BPC : Background Persistency Component

6.5 Removing Objects

The delete operation removes data from a **XOBE** service node . Analogously to the load operation objects are selected by a selection expression. After the successful processing of a delete operation none of the selected objects are available any longer. In this case the answer is a delete done response.

The **XOBE**$_{\text{DBPL}}$ web service client requests an object deletion by sending an identification ID and a selection expression. After checking the identification ID the web service component evaluates the selected set of object IDs analogously to the load case. With this information the web service component forwards the delete request to its background persistency component or respectively broadcasts it to the corresponding **XOBE** service nodes, which are part of the same cluster. If the objects attributes are marked as aggregated they are not influenced by this deletion process. Otherwise, if marked as composed they are deleted recursively. Moreover, objects which are requested to be deleted are not deleted physically at the same instance, instead, this happens after a specific time interval. The mechanism can be compared with a garbage collector in the Java programming language.

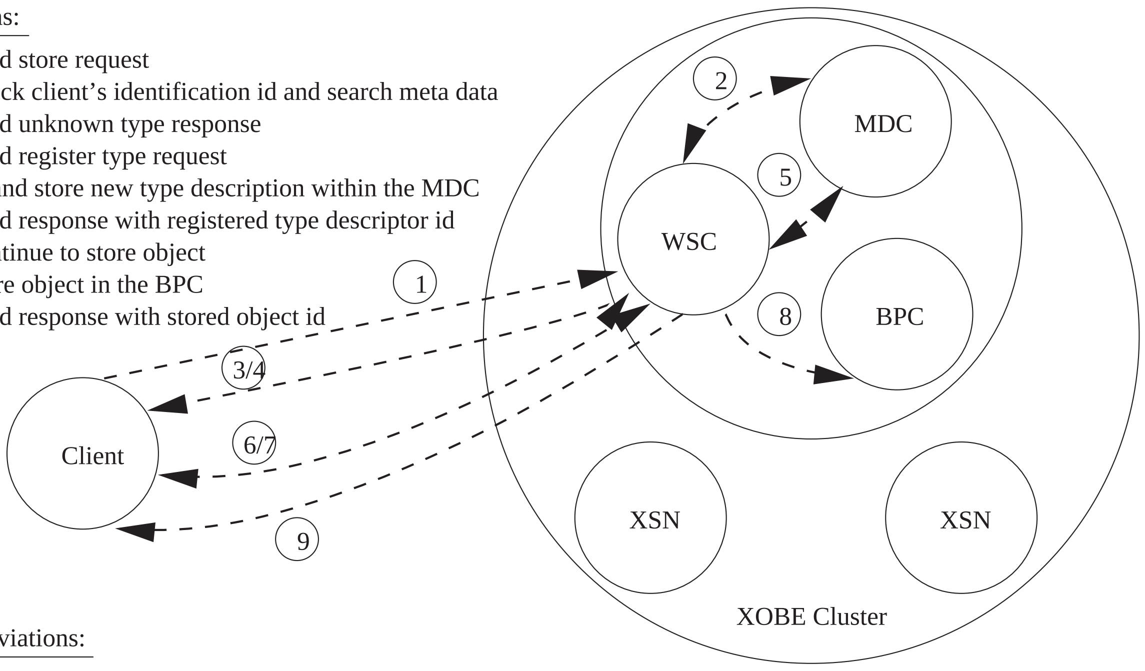
The delete process is shown in figure 6.9.

Besides selection expressions, data objects and class descriptors, successful communication

Figure 6.8: Storing a new data object with unknown type in single transaction mode

Actions:

1. send store request
2. check client's identification id and search meta data
3. send unknown type response
4. send register type request
5. 2. and store new type description within the MDC
6. send response with registered type descriptor id
7. continue to store object
8. store object in the BPC
9. send response with stored object id



Abbreviations:

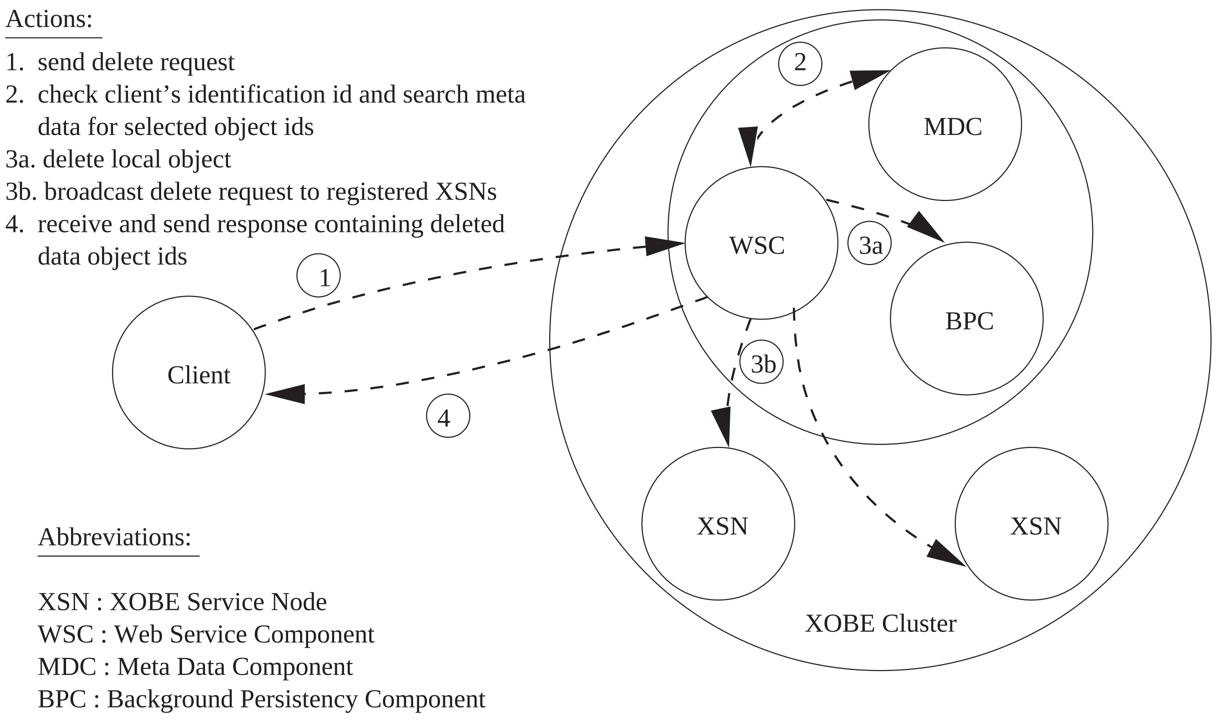
XSN : XOBE Service Node
WSC : Web Service Component
MDC : Meta Data Component
BPC : Background Persistency Component

with the **XOBE**$_{\text{DBPL}}$ web service requires session IDs to identify clients and transaction IDs to guarantee consistent access of distributed objects. These IDs and their semantics are described in the following subsection.

6.6 Session and Transactions

Session IDs as well as transaction IDs can be requested from the **XOBE**$_{\text{DBPL}}$ web service. IDs in the whole **XOBE**$_{\text{DBPL}}$ runtime environment are generated in a decentralized manner by using a hashing algorithm. For example the input value to calculate an object ID consists of its creation time, its type and client information. Accordingly, the input value for a session ID consists of the client's IP address and a time stamp. Hence, the generated IDs are statistically unambiguous. The collision probability is smaller than $1{:}2^{64}$. Nevertheless, if an object gets an ID which is already in use and is stored for the first time there are two possible cases. The first case is that the object has got another type than the already stored object. In this case an error

Figure 6.9: Deleting a data object


Actions:
1. send delete request
2. check client's identification id and search meta data for selected object ids
3a. delete local object
3b. broadcast delete request to registered XSNs
4. receive and send response containing deleted data object ids
Abbreviations:
XSN : XOBE Service Node
WSC : Web Service Component
MDC : Meta Data Component
BPC : Background Persistency Component
Client
1
2
3a
3b
4
WSC
MDC
BPC
XSN
XSN
XOBE Cluster

is thrown on client-side and the persistency layer remains unaffected. The second case is that the object and the already stored object are of the same type. In this case the second and latest object is interpreted as an object update. If the client is allowed to write the previous object it is overwritten by the new one. Otherwise an error is risen on client-side and the persistency layer remains unaffected. Please notice that this approach for generating object IDs in distributed environments is commonly used and other approaches, e.g. based on a centralized ID generator, will suffer from enormous performance problems and can only be considered theoretically.

A client requests a session ID by passing client specific data. After checking this data, the server sends a session ID with an initial lifetime, which is rather short. Besides, the web service generates its own customer ID to identify the client. Further requests from the same client extend the lifetime of the corresponding session ID. A client can finish any session by sending the ID back to the **XOBE**$_{\text{DBPL}}$ web service.

After receiving a valid session ID the client can perform load operations and store and delete operations in single transaction mode. To perform modifying operations like store and delete in complex transaction mode a transaction ID is needed in addition. Transaction IDs are given to the client on the basis of its session ID.

As mentioned before, every single modifying operation which is requested with a session

ID is handled and executed as a single short transaction internally. Complex transactions which consist of more than one modifying operation are requested along with a transaction ID. These complex transactions are executed as distributed transactions by the web-service-based persistency layer. As described in chapter 5 **XOBE**$_{\text{DBPL}}$ uses the two-phase commit protocol in conjunction with the distributed, timestamp-based optimistic concurrency control algorithm [75] . In this section only specific details concerning the realization with the web-service-based persistency layer are discussed. A more general description is alread given in section 5.3.1. The transactions are timestamped according to the Lamport protocol [48]. Transactions are initialized by a registered client application during its session. The **XOBE** web service component which is directly connected to this client application acts as the master transaction manager. The client application sends subsequent load and store operations along with the transaction ID. If such an operation cannot be handled by the master node itself it is passed to the other **XOBE** web services. Each operation which cannot be handled by the master node has to be registered. Finally, the corresponding response is received by the master again. The master checks which web service instance processed the operation. This information is part of the response message. The master stores this web service instance using its meta data component as a subtransaction manager for this transaction. In case the response is not received by the master in time the corresponding transaction has to be aborted. If the client commits the transaction the master starts the two-phase commit protocol as already described. A prepare to commit is sent using the `endTransaction` method in conjunction with a prepare-to-commit request. Contrary, a commit is sent in conjunction with a commit request. Otherwise, the master sends an abort message to each of the registered subtransaction managers for this transaction.

Nested transactions are not supported by the web-service-based persistency layer so far. For more information on transactions in **XOBE**$_{\text{DBPL}}$ please have a look at chapter 5. More details about the **XOBE**$_{\text{DBPL}}$ web service can be found in [73] and [65].

Chapter 7

Architecture and Implementation

This section contains the most important details about the architecture and implementation of the **XOBE**$_{\text{DBPL}}$ preprocessor, the program transformation process as well as about the runtime environment.

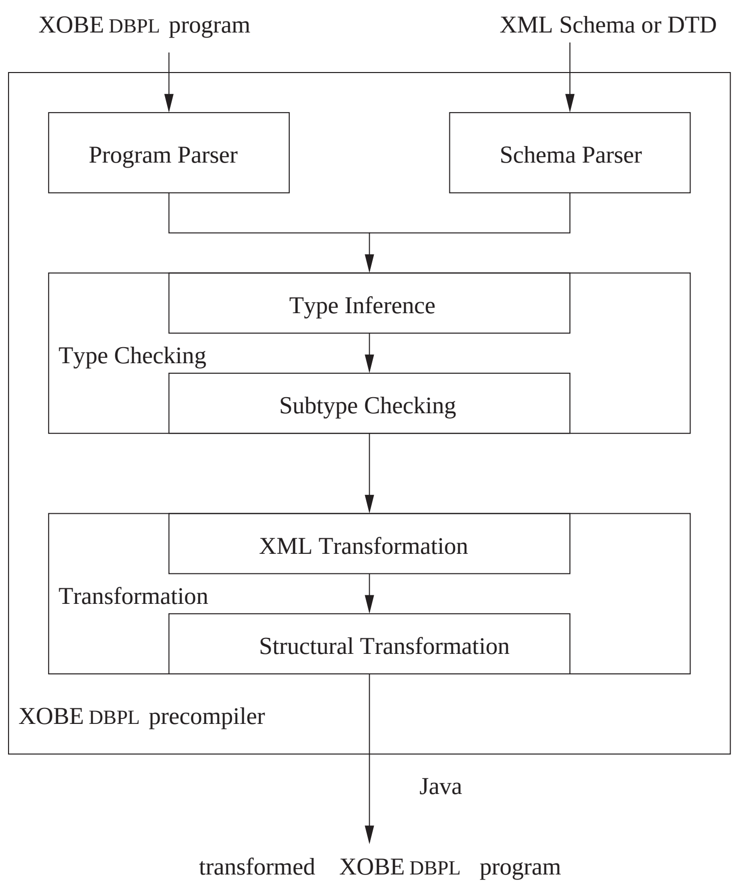
7.1 Preprocessor

The architecture of the **XOBE**$_{\text{DBPL}}$ preprocessor is shown in figure 7.1. In our implementation we use the Java compiler compiler JavaCC [37] to generate the **XOBE**$_{\text{DBPL}}$ program parser. Additionally we use the Xerces XML parser [5] to recognize the declared schemas. In the figure denoted by `Schema Parser` types are formalized by regular hedge expressions. The internal representation of processed **XOBE**$_{\text{DBPL}}$ programs is done via the Java tree builder JTB [30]. After parsing the input program and its required XML schemas the preprocessor infers the types of XML constituents in the program. These resulting types are used for the subsequent subtype checking. The algorithm is explained in [42] in detail. Finally, after the type checking process the input program is transformed into pure Java. The first step transforms XML specific program parts into DOM objects and corresponding method invocations. The second step performs the structural transformations into a framework of Java classes and interfaces. The framework depends on the underlying transient/persistent variant as described in section 4.2. The output of the whole preprocessor are the transformed **XOBE**$_{\text{DBPL}}$ programs that can be executed by any standard Java Virtual Machine (JVM). In section 7.2 the most important implementation details concerning the transformation process of a **XOBE**$_{\text{DBPL}}$ program are introduced.

7.2 Transformation

In this section we will use the same notation to define transformations as in [42]. A transformation instruction is described by a rule of the form $[[s]]^p_r \Rightarrow t$. S stands for an expression of

Figure 7.1: Architecture of the **XOBE**$_{\text{DBPL}}$ preprocessor



the source language and t for the desired translated expression in the target language. The parameter annotation p stands for values necessary for the further transformation process. These values are passed. Contrastly the resulting annotation r is set after the transformation is applied and therefore is available in succeeding transformation steps.
The transformation process is twofold. During the first step all XML specific program parts, e.g. XML object constructors, XPath, flwor and update expressions, that are extensions to the Java programming language are translated. Finally the second step performs structural transformations to realize persistency and transactions in **XOBE**$_{\text{DBPL}}$ as it is described in section 4.2 and chapter 5.

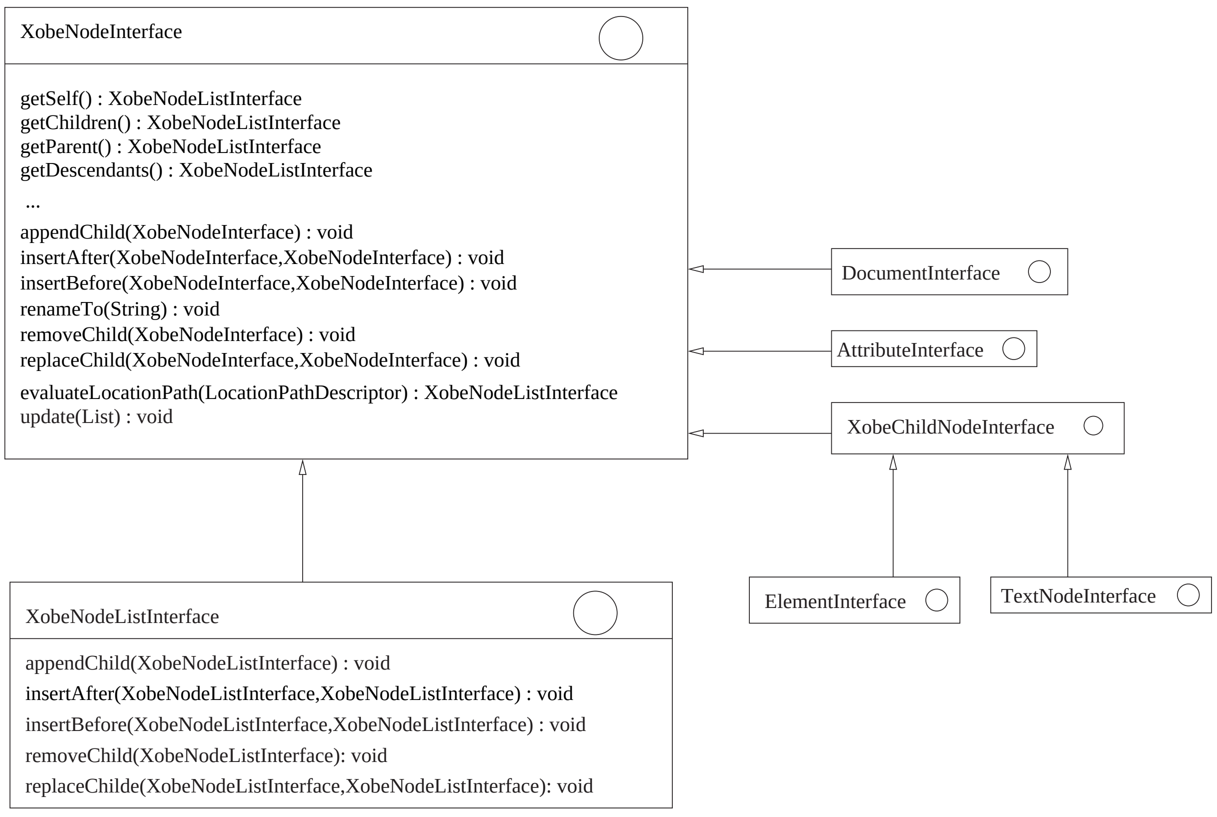
7.2.1 XML Transformation

The translation of XML object constructors and XPath expressions is already described in [42]. While the implementation of XML object constructors nearly remain unchanged in **XOBE**$_{\text{DBPL}}$ the implementation in the case of XPath changes entirely. In addition update and flwor expressions have to be taken into account.

XML Object Constructors

The translation of XML object constructors does not change except that **XOBE**$_{\text{DBPL}}$ uses a slightly modified DOM variant to represent XML objects. The node interface is called `XobeNodeInterface` and lists are represented by the `XobeNodeListInterface` interface. The interfaces can be seen in figure 7.2. The more specialized interfaces, e.g. `DocumentInterface`, `ElementInterface`, extend the `XobeNodeInterface` interface as it is known from the DOM. The `XobeNodeInterface`

Figure 7.2: The basic interfaces `XobeNodeInterface` and `XobeNodeListInterface` for the XML transformation process


XobeNodeInterface
getSelf() : XobeNodeListInterface
getChildren() : XobeNodeListInterface
getParent() : XobeNodeListInterface
getDescendants() : XobeNodeListInterface
...
appendChild(XobeNodeInterface) : void
insertAfter(XobeNodeInterface,XobeNodeInterface) : void
insertBefore(XobeNodeInterface,XobeNodeInterface) : void
renameTo(String) : void
removeChild(XobeNodeInterface) : void
replaceChild(XobeNodeInterface,XobeNodeInterface) : void
evaluateLocationPath(LocationPathDescriptor) : XobeNodeListInterface
update(List) : void
DocumentInterface
AttributeInterface
XobeChildNodeInterface
ElementInterface
TextNodeInterface
XobeNodeListInterface
appendChild(XobeNodeListInterface) : void
insertAfter(XobeNodeListInterface,XobeNodeListInterface) : void
insertBefore(XobeNodeListInterface,XobeNodeListInterface) : void
removeChild(XobeNodeListInterface): void
replaceChilde(XobeNodeListInterface,XobeNodeListInterface): void

as well as the `XobeNodeListInterface` provide methods for all XPath axis and basic update operations, e.g. append, insert after, insert before, rename, remove and replace. Furthermore, two important methods are offered, e.g. `evaluateLocationPath` and `update`. While `evaluateLocationPath` accepts a `LocationPathDescriptor` as a parameter and is used to query XML objects using translated location paths, `update` accepts an `UpdateDescriptor` as parameter and is used to update XML objects using translated update expressions.

XPath expressions

XPath expressions in **XOBE**$_{\text{DBPL}}$ are always executed upon a context variable . The variable references an already translated XML object. The remaining location path is translated into an equivalent location path descriptor object. Finally the `evaluateLocationPath` method described previously is invoked upon the context variable and the location path descriptor object is passed as parameter. This is different to the transformation of XPath expressions in [42]. There each location path step is translated into a single method invocation upon the context object. Since now XML objects may be persistent and remote, the former transformation process can lead to inefficiencies, because each method invocation upon the XML object might be serialized and then executed. The new XPath expression transformation first constructs a local XPath descriptor object and finally invokes the evaluate method one-time. The location path desriptor object is sent and executed where the context object is located. The interface to represent a location path descriptor is shown in listing 7.1.

Listing 7.1: The `LocationPathDescriptor` interface

```
public interface LocationPathDescriptor{
 public String getAxis();
        public void setAxis(String axis);
        public String getNodeTest();
        public void setNodeTest(String nodeTest);
        public List getPredicates();
        public void addPredicate(Predicate predicate);
        public LocationPathDescriptor getNext();
        public void setNext(LocationPathDescriptor next);
}
```

A location path consists of an arbitrary number of steps. Thus, the descriptor interface contains a reference to the next step. Each step consists of an axis name, node test and an arbitrary number of predicates, implied by the corresponding getters and setters. An implementation of the evaluate method defined by the `XobeNodeInterface` interface can be seen in listing 7.2.

Listing 7.2: Implementation of the `evaluateLocationPath` method

```
public XobeNodeListInterface evaluateLocationPath(
 LocationPathDescriptor path){
```

```
LocationPathDescriptor next = path;
//result of the location path
XobeNodeListInterface result = new XobeNodeList(this);
do{
 XobeNodeListInterface resultTemp = new XobeNodeList();
 //analyze next step of location path...
 String axisName = next.getAxisName();
 String nodeTest = next.getNodeTest();
 List predicates = next.getPredicates();
 next = path.getNext();
 //apply axis...
 if(axisName.equals("child")){
  result = result.getChildNodes();
 }else if(axisName.equals("descendant")){
  result = result.getDescendant();
 }
 ...//all remaining axis
 //apply node test
 result = result.nodeTest(nodeTest);
 //if predicates are given, evaluate predicates...
 if(predicates.size()>0){
  Iterator iter = predicates.iterator();
  while(iter.hasNext()){
   Predicate predicate =
    (Predicate)iter.next();
   for(int i=0; i<result.size(); i++){
    //filter each node of the result
    if(predicate.evaluateFor(result.get(i))){
     resultTemp.add(result.get(i));
    }else{
     resultTemp.remove(result.get(i));
    }
   }//end for
   result = resultTemp;
  }//end while
 }//predicates present
}while(next!=null);
return result;
}
```

Now we can define the transformation rule in the case of an XPath expression in **XOBE**$_{\text{DBPL}}$ 7.2.1.

Definition 7.2.1 The transformation of an XPath expression with the context XML object variable V and the location steps S_i with $i \in \{1, ..., n\}$ is defined by:

- an XPath expression with explicit context variable V

$$[[V/S_1/.../S_n]]_r \quad \Rightarrow \quad \begin{array}{l} \text{LocationPathDescriptor p = null;} \\ \{ \\ \text{LocationPathDescriptor t =} \\ \text{new LocationPathDescriptorImpl("self","}*\text{");} \\ \text{p = t;} \\ [[S_1]]^t \\ . \\ . \\ . \\ [[S_n]]^t \\ \} \\ \text{XobeNodeListInterface r = V.evaluateLocationPath(p);} \end{array}$$

- an XPath expression with implicit context variable as may be used within predicates and has to be used within update operations

$$[[/S_1/.../S_n]]_p \quad \Rightarrow \quad \begin{array}{l} \text{LocationPathDescriptor p = null;} \\ \{ \\ \text{LocationPathDescriptor t =} \\ \text{new LocationPathDescriptorImpl("self","}*\text{");} \\ \text{p = t;} \\ [[S_1]]^t \\ . \\ . \\ . \\ [[S_n]]^t \\ \} \end{array}$$

R references the resulting node list and `LocationPathDescriptorImpl` denotes an implementation of the interface `LocationPathDescriptor`. □

An XPath expression within a **XOBE**$_{\text{DBPL}}$ program is transformed into several Java statements. In the case of XPath expressions with a given context variable V, the result of an XPath expression is a list of XML objects. These objects are put into the result node list r. In principal the remaining location steps are translated into a location path descriptor object p. A temporary descriptor object t is initialized with a step selecting the self node and is assigned to p. The result list is gained by evaluating the descriptor object p upon the given context variable V. Each location step is transformed separately and adds information to the temporary descriptor object t and thus to the descriptor object p as well. In contrast to XPath expressions with a context variable V, XPath expressions within predicates or updates may start with a step. Semantically this means that the path expression is evaluated upon the selected node list of this predicate's step or the update target variable respectively. Then the transformation rule generates only the path descriptor that is later used in the predicate's evaluation.

Definition 7.2.2 The transformation of a location step consisting of the axis name A, the node test N and the predicates P_i with $i \in \{1, ..., n\}$ and the corresponding descriptor object t is defined by:

$$[[A :: N[P_1]...[P_n]]]^t \quad \Rightarrow \quad \begin{array}{l} \text{t.setNext(new LocationPathDescriptorImpl(A,N));} \\ \text{t = t.getNext();} \\ \{ \\ [[\quad [P_1] \quad]]_{p_1} \\ \text{t.addPredicate(}p_1\text{);} \\ . \\ . \\ . \\ [[\quad [P_n] \quad]]_{p_n} \\ \text{t.addPredicate(}p_n\text{);} \\ \} \end{array}$$

with `LocationPathDescriptorImpl` denoting an implementation of the interface `LocationPathDescriptor`. □

Each location step is transformed and appended to the corresponding location path descriptor object t. An example XPath expression without predicates is translated in 7.2.1 with the rules given so far.

Example 7.2.1 This example selects all person elements of the $auctionSite$ XML object variable with the help of an XPath expression. The resulting node list is assigned to the variable ps.

```
xml<person*> ps = $auctionSite/descendant::person$;
```

According to the defined XPath transformation rules the following pure Java code is generated.

```
LocationPathDescriptor p = null;
{
 LocationPathDescriptor t =
  new LocationPathDescriptorImpl("self","*");
  p = t;
  t.setNext(new LocationPathDescriptorImpl("descendant","person"));
  t = t.getNext();
}
XobeNodeListInterface ps = auctionSite.evaluateLocationPath(p);
```

Variables p and t reference location path descriptors. While p points to the initial location path step, t points to the last location path step. The result ps is translated into a **XOBE** node list that is gained by executing the path descriptor upon the context variable $auctionSite$.

While axis names and node tests can be taken as parameters without further translations, existing predicates need to be transformed by the following rule. The main idea in the context of a predicate transformation is equivalent to XPath expressions themselves. The predicate is translated into a descriptor object of type `Predicate`. In **XOBE**$_{\text{DBPL}}$ predicates consist of a predicate boolean expression that may contain e.g. conditional-or, conditional-and and relational or XPath expressions again. XPath expressions are translated according to the already defined rules. Conditional-or, conditional-and and relational expressions are represented by objects of type `CondOrPredicateExpr`, `CondAndPredicateExpr` and `RelPredicateExpr` respectively. All these types are subtypes of the `Predicate`.

Definition 7.2.3 The transformation of a predicate with the predicate expression PE is defined by:

$$[[\quad [PE] \quad]]_p \quad \Rightarrow \quad \begin{array}{l} [[PE]]_p \\ \text{Predicate predicate = p;} \end{array}$$

where p denotes the resulting predicate descriptor. □

Since the different kinds of predicate expressions are transformed equivalently, we will focus on the transformation rules for a conditional-and expression as well as for a relational expression that contains the equality operator.

Definition 7.2.4 The transformation for a conditional-and expression with the left hand side expression E_l and the right hand side expression E_r is defined by:

$$[[E_l \quad \&\& \quad E_r]]_{(\text{new CondAndPredicateExpr}(e_l,e_r))} \quad \Rightarrow \quad \begin{array}{l} [[E_l]]_{e_l} \\ \\ [[E_r]]_{e_r} \end{array}$$

where `CondAndPredicateExpr` denotes a class that represents a conditional-and expression. □

The transformation is applied recursively to the left and right hand side expression. The result of this transformation rule is the new conditional-and expression using the results of the recursive transformation processes as parameters.

Definition 7.2.5 The transformation for a comparison relation consisting of the left hand side expression E_l and the right hand side expression E_r is defined by:

$$[[E_l \quad == \quad E_r]]_{(\text{new RelPredicateExpr}(e_l,e_r,\text{"=="}))} \quad \Rightarrow \quad \begin{array}{c}[[E_l]]_{e_l}\\ \\ [[E_r]]_{e_r}\end{array}$$

where `RelPredicateExpr` denotes a class that represents a relational expression. □

Again the transformation rule is applied recursively to the right and left hand side. The result is a new object representing the comparison relational expression with the results of the recursive transformation processes as parameters. The example 7.2.2 applies the transformation rules including predicate rules given so far.

Example 7.2.2 This example adds a predicate to the XPath expression used in example 7.2.1. Now, only a single person with a given ID is selected and inserted into the resulting node list ps.

```
xml<person*> ps = $auctionSite/descendant::person[/@id = "p_00001"]$;
```

According to the defined XPath transformation rules the following pure Java code is generated.

```
LocationPathDescriptor p = null;
{
 LocationPathDescriptor t =
  new LocationPathDescriptorImpl("self","*");
 p = t;
 t = t.next(new LocationPathDescriptorImpl("descendant","person"));
 //new code corresponding to the predicate
 {
  //transformation of "/@id"
  LocationPathDescriptor p2 = null;
  {
   LocationPathDescriptor t2 =
    new LocationPathDescriptorImpl("self","*");
   p2 = t2;
   t2 = t2.next(new LocationPathDescriptorImpl("attribute","id"));
  }
  Predicate predicate =
    new RelPredicateExpr(new PrimaryPredicateExpr(p2),
                         new PrimaryPredicateExpr("p_00001"),
                         "==");
   t.addPredicate(predicate);
  }
  //end of predicate related code
}
XobeNodeListInterface ps = auctionSite.evaluateLocationPath(p);
```

The beginning of the generated code that is new and corresponds to the predicate is marked by a comment. A predicate in **XOBE**$_{\text{DBPL}}$ may contain location paths with implicit context variable, e.g. $@id$. The location path is transformed recursively into an inner location path descriptor object that is used to construct the predicate descriptor. Notice that in this case the predicate descriptor is a relational predicate expression instance. Finally, the predicate descriptor is added to the current or last location path step t.

Next we will go on to introduce transformation rules in the case of complex queries and updates.

Flwor Expressions

XFlwor expressions containing a return clause are called flwor expressions. The transformation of flwor expressions is based on the already defined rules for path expressions and XML objects. The definition can be seen in 7.2.6.

Definition 7.2.6 Without loss of generality the following transformation is given for flwor expressions containing at least one for and one let clause. The transformation for a flwor expression, consisting of the for clauses F_i with $i \in \{1, ..., n\}$, the let clauses L_j with $j \in \{1, ..., m\}$, the where clause W, the order by clause C and the return clause R, is defined by:

- $j < i$ and $1 \leq k < i$ and $j \leq t \leq m$:

$[[$		
.		.
.		.
.		.
L_j		$[[L_j]]_{l_j}^{(f\_1\_i,l_1,\ldots,f\_k\_i,l_{j-1})}$
.		.
.		.
.		.
F_i		$[[F_i]]_{f_i}^{(f\_1\_i,l_1,\ldots,f\_i-1\_i,l_t)}$
		for(int i_i=0; i_i< f_i.size();i_i++){
		XobeNodeListInterface f_i_i = new XobeNodeList(f_i.get(i_i));
.		.
.		.
.	$\Rightarrow$	.
W		$[[W]]^{(f\_1\_i,l_1,\ldots,f\_n\_i,l_m)}$
		XobeNodeListInterface r = new XobeNodeList();
O		$[[R]]^{(r,f\_1\_i,l_1,\ldots,f\_n\_i,l_m)}$
		.
		.
		.
		}//end for i_i
		.
		.
		.
R		$[[O]]^r$
$]]_r$		

If $j = 1$ the pre-evaluated let and for variables L_j do not exist.

- $i < j$ and $1 \leq t < j$ and $i \leq k \leq n$

$$
\begin{array}{lcl}
[[& & \\
\vdots & & \vdots \\
F_i & & [[F_i]]_{f_i}^{(f\_1\_i,l_1,\dots,f\_i-1\_i,l_t)} \\
 & & \text{for(int i\_i=0; i\_i}< f_i\text{.size();i\_i++)\{} \\
 & & \text{XobeNodeListInterface f\_i\_i = new XobeNodeList(}f_i\text{.get(i\_i));} \\
\vdots & & \vdots \\
L_j & & [[L_j]]_{l_j}^{(f\_1\_i,l_1,\dots,f\_k\_i,l_{j-1})} \\
\vdots & \Rightarrow & \vdots \\
W & & [[W]]^{(f\_1\_i,l_1,\dots,f\_n\_i,l_m)} \\
 & & \text{XobeNodeListInterface r = new XobeNodeList();} \\
O & & [[R]]^{(r,f\_1\_i,l_1,\dots,f\_n\_i,l_m)} \\
 & & \vdots \\
 & & \text{\}//end for i\_i} \\
 & & \vdots \\
R & & [[O]]^r \\
]]_r & &
\end{array}
$$

If $i = 1$ the pre-evaluated let and for variables F_i do not exist.

□

The result of a flwor expression is the node list r. Let and for clauses define new local variables f_i and l_j. While let variables are taken as a whole, for variables cause iteration processes. These iteration processes are translated into for-loops. Where clauses act as filters upon let and for variables. The result clause constructs the overall result of the flwor expression r and finally the order by clause kind of sorts the result list.
The isolated let and for clause transformations are equally defined as can be seen in the definitions 7.2.7 and 7.2.8.

Definition 7.2.7 The transformation of a let clause with the variable i and the path expression p is defined by:

$$[[\text{LET i := p }]]_i^{(fl_1,...,fl_n)} \quad \Rightarrow \quad [[p]]_i^{(fl_1,...,fl_n)}$$

□

fl stands for either a for or let variable. The different binding behaviour of a for and a let variable is preserved by the flwor rule as mentioned before.

Definition 7.2.8 The transformation of a for clause with the variable i and the path expression p is defined by:

$$[[\text{FOR i IN p}]]_i^{(fl_1,...,fl_n)} \quad \Rightarrow \quad [[p]]_i^{(fl_1,...fl_n)}$$

□

For and let transformation rules simply rely on those for path expressions 7.2.1. The transformation rule for a where clause is not given here since it would go beyond the scope of this work. Moreover, where clauses are transformed analogously to XPath predicates. The transformation rule for return clauses is given in definition 7.2.9. Once again this rule is based on path expressions as well as XML object transformation rules.

Definition 7.2.9 The transformation of a return clause with path expressions, XML objects or variables pv_i with $i \in \{1, ..., n\}$ and predefined for or let variables fl_j with $j \in \{1, ..., m\}$ is defined by:

$$[[RETURN \quad pv_1, ..., pv_n]]^{(r,fl_1,...,fl_m)} \quad \Rightarrow \quad \begin{array}{l} [[pv_1]]_{pvr_1}^{(fl_1,...,fl_m)} \\ \text{r.add}(pvr_1); \\ . \\ . \\ . \\ [[pv_n]]_{pvr_n}^{(fl_1,...,fl_m)} \\ \text{r.add}(pvr_n); \end{array}$$

□

Each component of a return clause is transformed first and then added to the result list r. r is instantiated before and passed as reference. Finally an order by clause is transformed according to the rule defined in 7.2.10.

Definition 7.2.10 The transformation of an order by clause with the order specification $spec$ is defined by:

$$[[\text{ORDER BY spec}]]^r \quad \Rightarrow \quad \text{r.orderBy("spec");}$$

□

The resulting node list r is ordered with respect to the specification $spec$. An example demonstrating the transformation rules for flwor expressions is given in 7.2.3.

Example 7.2.3 This example transforms the flwor expression of listing 3.2 that returns names and quantities of sellers referenced by open auction elements.

```
$FOR i IN auctionSite//person
 LET j := auctionSite//open_auction[/seller/@id=i/@id]
 RETURN i/name j/quantity$;
```

According to the transformation rules for flwor expressions the following pure Java code is generated:

```
//for clause transformation
LocationPathDescriptor p = null;
{
 LocationPathDescriptor t =
  new LocationPathDescriptorImpl("self","*");
 p = t;
 t = t.next(
  new LocationPathDescriptorImpl("descendant","person")
           );
}
XobeNodeListInterface i =
 auctionSite.evaluateLocationPath(p);
for(int i_0=0; i_0<i.size(); i_0++){
 XobeNodeListInterface i_i_0 =
  new XobeNodeList(i.get(i_0));
 //let clause transformation
 LocationPathDescriptor p2 = null;
 {
  LocationPathDescriptor t2 =
   new LocationPathDescriptorImpl("self","*");
  p2 = t2;
  t2 = t2.next(
   new LocationPathDescriptorImpl("descendant","open_auction")
             );
  //predicate transformation
  {
   LocationPathDescriptor p3 = null;
   {
      LocationPathDescriptor t3 =
        new LocationPathDescriptorImpl("self","*");
      p3 = t3;
      t3 = t3.next(
        new LocationPathDescriptorImpl("child","seller")
                  );
      t3 = t3.next(
        new LocationPathDescriptorImpl("attribute","id")
                  );
    }
    LocationPathDescriptor p4 = null;
    {
      LocationPathDescriptor t4 =
        new LocationPathDescriptorImpl("self","*");
      p4 = t4;
      t4 = t4.next(
        new LocationPathDescriptorImpl("attribute","id")
                  );
    }
    XobeNodeListInterface i4 = i.evaluateLocationPath(p4);
  }
  Predicate p = new RelPredicateExpr(
    new PrimaryPredicateExpr(p3),
    new PrimaryPredicateExpr(i4),
    "==");

  t2.addPredicate(p);
}
XobeNodeListInterface j =
auctionSite.evaluateLocationPath(p2);

//return clause transformation
//flwor expression result
XobeNodeListInterface r = new XobeNodeList();
//transformation of first component
LocationPathDescriptor p5 = null;
{
  LocationPathDescriptor t5 =
    new LocationPathDescriptorImpl("self","*");
  p5 = t5;
  t5 = t5.next(
    new LocationPathDescriptorImpl("child","name")
              );
```

```
72  }
73  XobeNodeListInterface i5 =
74   i.evaluateLocationPath(p5);
75
76  //first transformed component
77  //is added to result
78  r.add(i5);
79
80  //transformation of the second component
81  LocationPathDescriptor p6 = null;
82  {
83   LocationPathDescriptor t6 =
84    new LocationPathDescriptorImpl("self","*");
85   p6 = t6;
86   t6 = t6.next(
87    new LocationPathDescriptorImpl("child","quantity")
88              );
89  }
90  XobeNodeListInterface j6 =
91   j.evaluateLocationPath(p6);
92
93  //second transformed component
94  //is added to result
95  r.add(j6);
96 }
```

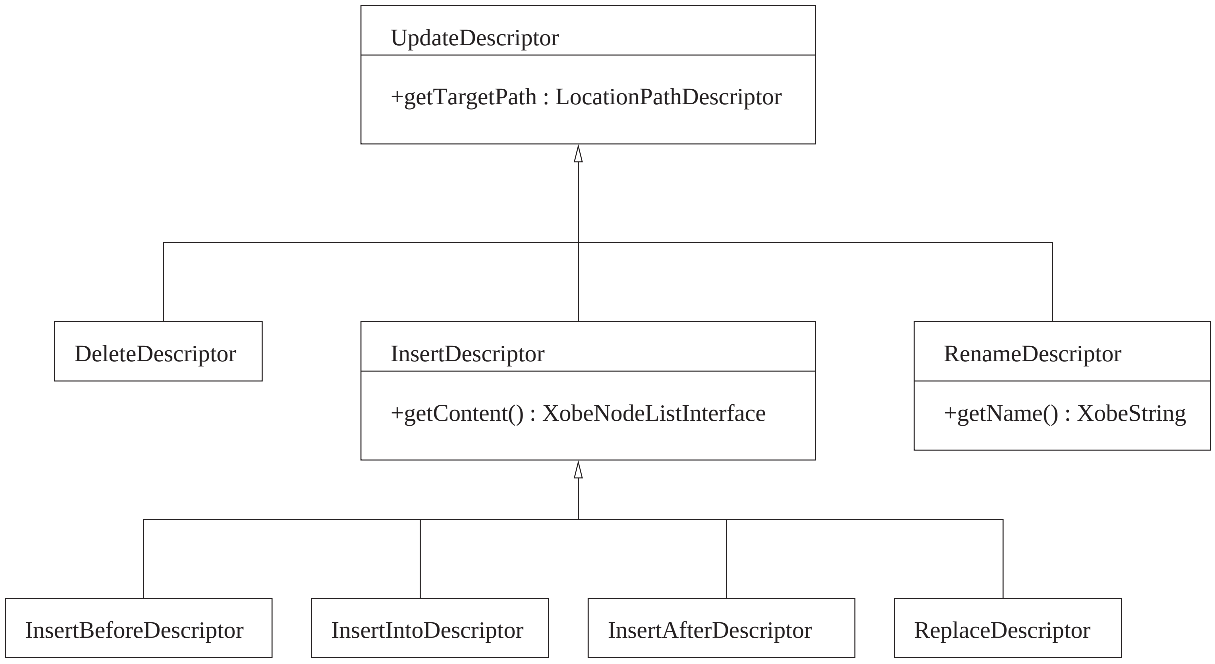
The generated code that corresponds to the new flwor transformation rules really starts in line 60. In line 62 the node list, that collects all resulting tuples of the whole flwor expression, is generated. In line 78 and 95 the transformed components representing the path expressions $i/name$ and $j/quantity$ are added to the result.

Update Expressions

Update expressions are xFLWOR expressions containing an update clause. The rules for a let, for and where clause are the same as defined in the previous section. For this reason new rules for the update clause itself are needed. Similar to XPath expressions and predicates basic update operations are translated into update descriptors. The type hierarchy that is used by **XOBE**$_{\text{DBPL}}$ can be seen in figure 7.3. Each kind of basic update operation is represented by a class. If the update clause consists of several basic update operations the corresponding descriptor objects are put in a list. As can be seen in figure 7.2 the interface `XobeNodeInterface` declares an update method that accepts a list of update descriptor objects as a parameter. An implementation of this method is presented in listing 7.3.

Listing 7.3: An implementation of the `XobeNodeInterface` update method

Figure 7.3: The update descriptor type hierarchy



```
public void update(List updates){
 XobeNodeListInterface xnl = new XobeNodeList(this);
 Iterator iter = updates.iterator();
 while(iter.hasNext()){
  //get next update operation
  UpdateDescriptor update = (UpdateDescriptor)iter.next();
  //evaluate corresponding target nodes
  XobeNodeListInterface xnli =
   xnl.evaluateLocationPath(update.getTargetPath());
  if(update instanceof DeleteDescriptor){
   xnl.getParent().removeChild(xnl);
  }else if(update instanceof InsertIntoDescriptor){
   XobeNodeListInterface content =
   ((InsertIntoDescriptor)update).getContent();
   xnl.appendChild(content);
  }
  //remaining update operations
  //are treated analogously
  ...
 }
}
```

In contrast to the transformation rule 7.2.6 for a flwor expression, the transformation rule for an update expression covers an update clause instead of an order by and a return clause. The rule is given in definition 7.2.11.

Definition 7.2.11 Without loss of generality the following transformation is given for flwu expressions containing at least one for and one let clause. The transformation of an update expression consisting of the for F_i and let L_j clauses, with $i \in \{1, ..., n\}$ and $j \in \{1, ..., m\}$, the where clause W and an update clause U is defined by:

- $j < i$ and $1 \leq k < i$ and $j \leq t \leq m$

$$
\begin{array}{lcl}
[[& & \\
. & & . \\
. & & . \\
. & & . \\
L_j & & [[L_j]]_{l_j}^{(f\_1\_i,l_1,...,f\_k\_i,l_{j-1})} \\
. & & . \\
. & & . \\
. & & . \\
F_i & & [[F_i]]_{f_i}^{(f\_1\_i,l_1,...,f\_i-1\_i,l_t)} \\
 & & \text{for(int i\_i=0;i\_i<}\ f_i\text{.size();i\_i++)}\{ \\
 & & \text{XobeNodeListInterface f\_i\_i = new XobeNodeList(}f_i\text{.get(i));} \\
. & & . \\
. & & . \\
. & \Rightarrow & . \\
W & & [[W]]^{(f\_1\_i,l_1,...,f\_n\_i,l_m)} \\
U & & [[U]]^{(f\_1\_i,l_1,...,f\_n\_i,l_m)} \\
 & & . \\
 & & . \\
 & & . \\
 & & \}\text{//end for i\_i} \\
 & & . \\
 & & . \\
 & & . \\
]] & &
\end{array}
$$

If $j = 1$ the pre-evaluated let and for variables L_j do not exist.

- $i < j$ and $1 \leq t < j$ and $i \leq k \leq n$

$$
\left[\left[
\begin{array}{lll}
. & & . \\
. & & . \\
. & & . \\
F_i & & [[F_i]]_{f_i}^{(f\_1\_i,l_1,...,f\_i-1\_i,l_t)} \\
 & & \text{for(int i\_i=0;i\_i< } f_i\text{.size();i\_i++)\{} \\
 & & \text{XobeNodeListInterface f\_i\_i = new XobeNodeList(}f_i\text{.get(i));} \\
. & & . \\
. & & . \\
. & & . \\
. & & . \\
L_j & & [[L_j]]_{l_j}^{(f\_1\_i,l_1,...,f\_k\_i,l_{j-1})} \\
. & & . \\
. & \Rightarrow & . \\
W & & [[W]]^{(f\_1\_i,l_1,...,f\_n\_i,l_m)} \\
U & & [[U]]^{(f\_1\_i,l_1,...,f\_n\_i,l_m)} \\
 & & . \\
 & & . \\
 & & . \\
 & & \text{\}//end for i\_i} \\
 & & . \\
 & & . \\
 & & .
\end{array}
\right]\right]
$$

If $i = 1$ the pre-evaluated let and for variables F_i do not exist.

□

The only rule that has not been defined yet is the transformation rule for an update clause and is defined in 7.2.12.

Definition 7.2.12 The transformation of an update clause with the context variable i and the basic update operations U_i with $i \in \{1, ..., n\}$ is defined by:

$$
[[UPDATE \quad i \quad U_1, ...U_n]]^{(fl_1,...,fl_n)} \quad \Rightarrow \quad
\begin{array}{l}
\text{List d = new LinkedList();} \\
[[U_1]]^{(d,fl_1,...,fl_n)} \\
. \\
. \\
. \\
[[U_n]]^{(d,fl_1,...,fl_n)} \\
\text{i.update(d);}
\end{array}
$$

□

The list d is passed to each basic update transformation. Each basic update transformation process adds the corresponding descriptor object to this list. Finally the context variable i is updated by invoking the update method and passing the update descriptor list d. Basic update operations are transformed according to the rules given in the following definition 7.2.13.

Definition 7.2.13 The transformation of the basic update operations with the target path expression P and the content C is defined by:

- delete operation

$[[DELETE \quad P]]^{(d,fl_1,\ldots,fl_n)} \Rightarrow$ $[[P]]_t^{fl_1,\ldots,fl_n}$
UpdateDescriptor u = new DeleteDescriptor(t);
d.add(u);

- simple insert operation

$[[INSERT \quad C]]^{(d,fl_1,\ldots,fl_n)} \Rightarrow$ $[[C]]_c^{fl_1,\ldots,fl_n}$
UpdateDescriptor u = new InsertDescriptor(c);
d.add(u);

- insert into operation

$[[INSERT \quad C \quad INTO \quad P]]^{(d,fl_1,\ldots,fl_n)} \Rightarrow$ $[[C]]_c^{fl_1,\ldots,fl_n}$
$[[P]]_t^{fl_1,\ldots,fl_n}$
UpdateDescriptor u =
new InsertIntoDescriptor(c,t);
d.add(u);

- insert before operation

$[[INSERT \quad C \quad BEFORE \quad P]]^{(d,fl_1,\ldots,fl_n)} \Rightarrow$ $[[C]]_c^{fl_1,\ldots,fl_n}$
$[[P]]_t^{fl_1,\ldots,fl_n}$
UpdateDescriptor u =
new InsertBeforeDescriptor(c,t);
d.add(u);

- insert after operation

$$[[INSERT \quad C \quad AFTER \quad P]]^{(d,fl_1,\ldots,fl_n)} \Rightarrow$$

$[[C]]_c^{fl_1,\ldots,fl_n}$
$[[P]]_t^{fl_1,\ldots,fl_n}$
UpdateDescriptor u =
new InsertAfterDescriptor(c,t);
d.add(u);

- replace operation

$$[[REPLACE \quad P \quad WITH \quad C]]^{(d,fl_1,\ldots,fl_n)} \Rightarrow$$

$[[C]]_c^{fl_1,\ldots,fl_n}$
$[[P]]_t^{fl_1,\ldots,fl_n}$
UpdateDescriptor u =
new ReplaceDescriptor(c,t);
d.add(u);

- rename operation

$$[[RENAME \quad P \quad TO \quad m]]^{(d,fl_1,\ldots,fl_n)} \Rightarrow$$

$[[P]]_t^{fl_1,\ldots,fl_n}$
UpdateDescriptor u =
new RenameDescriptor(t,m);
d.add(u);

Semantically path expressions selecting target nodes of an update operation always have an implicit context variable and start with a slash **'/'**. □

The above rules are based on transformation rules for XML objects and XPath expressions. Each rule adds its corresponding update descriptor object to the list d that is passed by the update clause rule. In context of an update operation the XPath expression with an implicit context variable, e.g. the target variable of the update operation, selecting the target nodes is only transformed into the corresponding descriptor object p. An example demonstrating the transformation rules for update expressions is given in 7.2.4.

Example 7.2.4 This example update operation deletes all closed auction elements of the auction site.

```
$LET i := auctionSite
 UPDATE i DELETE //closed_auction$;
```

Applying the transformation rules for update expressions yields the following Java code.

```
//let clause transformation
LocationPathDescriptor p = null;
{
 LocationPathDescriptor t =
  new LocationPathDescriptorImpl("self","*");
 p = t;
}
XobeNodeListInterface i =
 auctionSite.evaluateLocationPath(p);

//update clause transformation
List d = new LinkedList();

//delete operation transformation
//target path transformation
LocationPathDescriptor p2 = null;
{
 LocationPathDescriptor t2 =
  new LocationPathDescriptorImpl("self","*");
 p2 = t2;
 t2 = t2.next(
  new LocationPathDescriptorImpl("descendant","closed_auction")
            );
}

UpdateDescriptor u = new DeleteDescriptor(p2);

d.add(u);

i.update(d);
```

7.2.2 Transaction Transformation

A transaction statement in **XOBE**$_{\text{DBPL}}$ is transformed according to the rule given in definition 7.2.14.

Definition 7.2.14 The transformation of a transaction statement with a name list consisting of the variables $p_1, ...p_n$ and a transaction block B is defined as:

[[**transaction**(p_1,...,p_n){B}]] $\Rightarrow$

```
TransactionID id =
LocalServer.getInstance().beginTransaction();
try{
synchronized(p1){
.
.
.
synchronized(pn){
p1.setTransactionID(id);
...
pn.setTransactionID(id);
[[B]]
}
.
.
.
}
}catch(TransactionAbortException e){
throw e;
}catch(Exception e){
LocalServer.getInstance().abortTransaction(id);
throw new TransactionAbortExcetion();
}
LocalServer.getInstance().endTransaction(id);
```

□

The above rule to transform a transaction statement into pure Java code starts a transaction by invoking the corresponding method on the local server instance of the **XOBE**$_{\text{DBPL}}$ runtime environment. The local server is used as an adapter between **XOBE**$_{\text{DBPL}}$ programs and the web service based persistency layer. The local server class provides the same methods as the **XOBE**$_{\text{DBPL}}$ web service, but with different, mostly simpler argument and response types. The purpose of the local server is to hide communication details with the concrete persistency layer implementation. The resulting transaction ID is set within a synchronized environment for each persistent object being part of the transactional name list. After setting the transaction IDs the transformed transaction block is inserted. Finally, the transaction is committed by calling the `endTransaction` method upon the local server instance. Within transactions a `TransactionAbortException` might be thrown by the local server. In this case the corresponding transactional operations are rolled back. The rollback is done by the local server or more precisely by the **XOBE**$_{\text{DBPL}}$ web service persistency layer. A server-side caused transaction abort is caught by the first catch clause. Moreover, a transaction can also be aborted by the client. If an exception occurs within the beginning and ending of a transaction, it is caught by the second catch clause. In this case the transaction has to be aborted invoking the corresponding method upon the local server and a `TransactionAbortException` is thrown explicitely.

7.2.3 Structural Transformation

Details about the concepts of structural transformation can be found in 4.2. Similarily to the transformation rules defined for XML program parts it is possible to define a set of transformation rules for the structural transformation process. Nethertheless, this would go far beyond the scope of this text and would yield nothing really new. Details about the realization concepts are given in chapter 5.

7.3 Runtime Environment

Figure 7.4 illustrates the **XOBE**$_{\text{DBPL}}$ runtime environment. As described in section 4.2 persistency, distribution and transactions are based on a special web service. The concepts of the **XOBE**$_{\text{DBPL}}$ web service as well as its architecture are described in detail in chapter 6. At the moment **XOBE**$_{\text{DBPL}}$ web services are implemented in .NET [54]. Transformed persistent classes communicate with a **XOBE**$_{\text{DBPL}}$ local server instance. Local servers use local persistency mechanisms like databases for local data. If data is shared they connect to a suitable given **XOBE**$_{\text{DBPL}}$ web service instance. The communication between the web service and the **XOBE**$_{\text{DBPL}}$ local server is automatically achieved with the help of the `WSDL2Java` tool that is part of the Apache Axis project [2]. Transformed transient classes do not communicate with local servers. Transient classes use persistent classes and vice versa. At the moment we use Postgres [66] as the relational database for performance reasons, but this is transparent to the programmer. Object-relational mapping is done automatically and independent of type. Further implementation details can be found in section 7.4. The figure also indicates that an arbitrary number of clients can access arbitrary web service instances. Moreover, there may be an arbitrary number of **XOBE**$_{\text{DBPL}}$ web service instances. The persistent data managed by the web service is passed as SOAP messages. Since these SOAP messages envelope data composed according to an intermediate language introduced in chapter 6, the clients need not be implemented in Java at all.

7.4 OR Mapping Techniques

In this section we will give basic issues of mapping objects into relational tables. As mentioned previously background databases in **XOBE**$_{\text{DBPL}}$ are mostly relational due to performance reasons. The techniques that are described here and used in **XOBE**$_{\text{DBPL}}$ are inspired by the proposals made in [1].

7.4.1 Mapping Member Variables

Member variables of a class are mapped to zero or more columns in a relational database. A member variable is mapped to zero columns if it is transient. It is mapped to one column if it has got a basic type including `String`, e.g. boolean, int, float. Furthermore, because some member variables of a class are themselves class types they are mapped to several columns of several tables. In **XOBE**$_{\text{DBPL}}$ each persistent object is identified by a global, non-business data, unique ID of type `String`. If a member variable is complex only its ID is mapped to a column. The ID references the object or respectively another row of the appropriate table. If this row does not already exist the corresponding member variable object must be mapped recursively. Object IDs are used as keys and foreign keys in the resulting relations.

Bulk types

Bulk types include lists and arrays. Since a list or array is also an object in Java a bulk typed attribute of an object is also mapped to its object ID value of the corresponding table. For each class and each of its bulk typed attributes a corresponding table is created. The table consists of three columns, the first column references the list's object ID, the second the object contained in this list instance and finally the third keeps the object's position in this list instance.

7.4.2 Mapping Classes

Classes map to tables. In [1] three different approaches are proposed and discussed in reference to inheritance that causes the most significant problems in every object-relational mapping approach. The first approach uses one table for an entire class hierarchy that will frequently lead to altering this table if a new class appears in the hierarchy. The second uses one table per concrete class. When modifying a class its table must be modified and all the tables of any of its subclasses. The third approach that is also used in **XOBE**$_{\text{DBPL}}$ uses one table per class. The main advantage of this approach is that it conforms to object-oriented concepts the best, e.g. polymorphism. One disadvantage might be that there are many tables in the database and objects of a subtype are read and written using multiple joins.

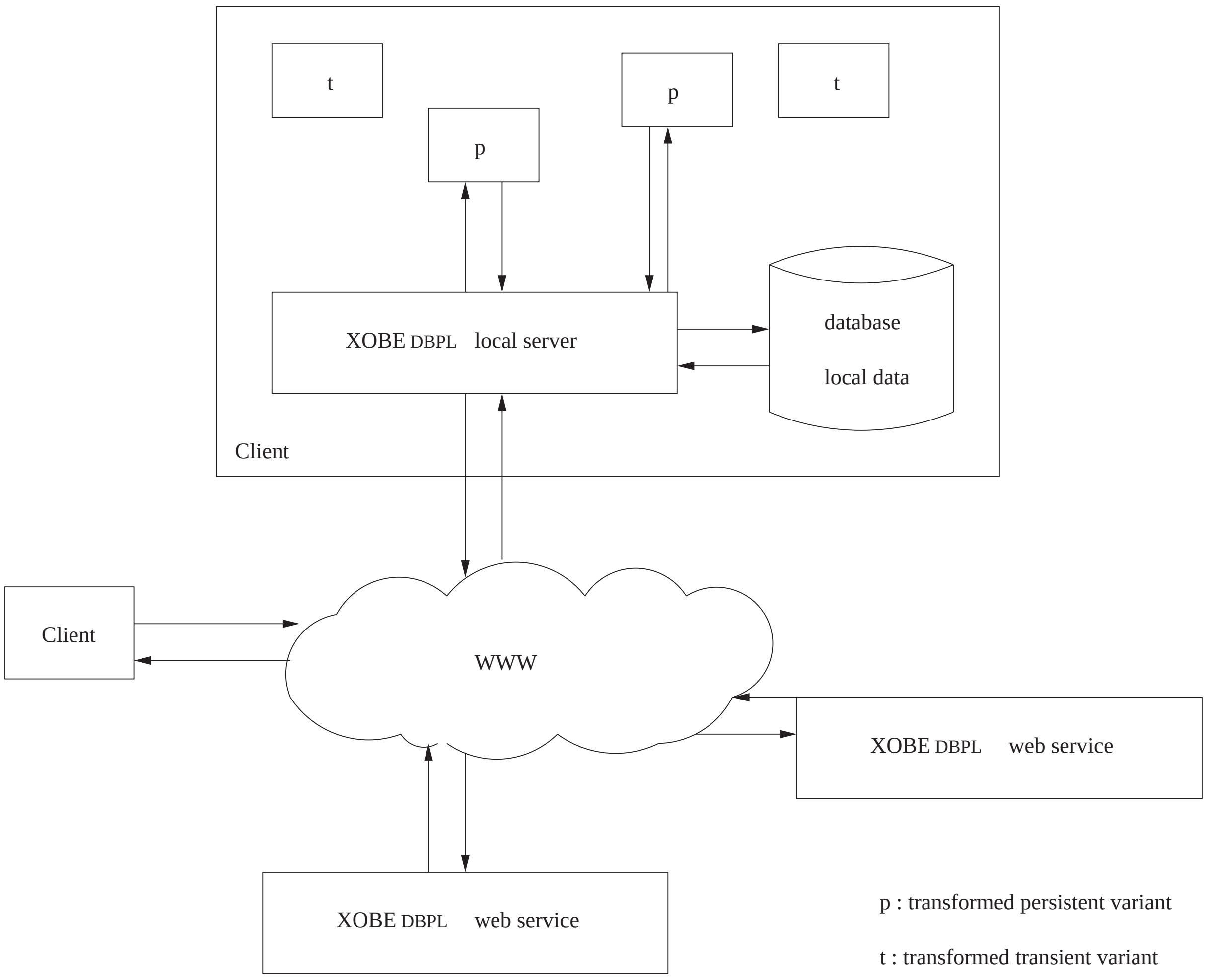
7.4.3 Relationships

There are two types of relationships that an object can be involved in, association and aggregation. Aggregation means that anything which occurs to the whole in the database needs to be performed upon the parts. By contrast, in the case of an association the parts are independent and remain unchanged. In **XOBE**$_{\text{DBPL}}$ database objects might only be associated with other database objects. Persistent, non database objects are always aggregated to one or more database objects. For example if a database object is deleted all its constituent objects are deleted as well. Relationships in relational databases are maintained through the use of *foreign keys* and *keys*. The mapping process used in **XOBE**$_{\text{DBPL}}$ does not use keys or foreign keys since unique IDs and references are generated outside of the database and mapped automatically as described before. If keys and foreign keys were to be declared or defined the database would have to check the corresponding constraints. Checking needs time and slows down update operations. Since the database mapping process is entirely transparent to the user such constraints are not neccessary.

7.5 Performance Aspects

Performance data concerning the precompiler's type checking mechanism can be found in [42]. Since in **XOBE**$_{\text{DBPL}}$ only a new set of type inference rules for updates that rely on those for XPath are introduced and nothing is changed in principal, the former results are still meaningful. The program transformation process can be estimated by $O(n)$ where n denotes the program size. Linear runtime can be expected since each node of the program tree that represents the input **XOBE**$_{\text{DBPL}}$ program is only processed once and is translated in a constant number of Java code lines. The latter is reflected by the transformation rules that are given in section 7.2. Moreover, the performance of the **XOBE**$_{\text{DBPL}}$ precompiler is not that significant, because each **XOBE**$_{\text{DBPL}}$ program has to be translated once, but the result can be executed arbitrarily often. Thus, the runtime behaviour of the generated Java classes is crucial for statements about the performance of **XOBE**$_{\text{DBPL}}$.
In chapter 9 experiments are performed. The first tests focussing on XML update support in **XOBE**$_{\text{DBPL}}$, as well as in related approaches, are made and evaluated. The second test concentrates on the example application and **XOBE**$_{\text{DBPL}}$'s overall functionality including XML integration and persistency.

Figure 7.4: The **XOBE**$_{\text{DBPL}}$ runtime environment



Chapter 8

XOBE$_{\text{DBPL}}$ as Semantic Extension

In the previous chapters the concepts of **XOBE**$_{\text{DBPL}}$ and their realization were introduced. In particular the integration of statically type checked XML and type independent transparent persistency are realized by extending the syntax of the chosen source programming language. Consequently, a complicated preprocessor is needed, which translates the extended Java syntax into pure Java again. More importantly, a **XOBE**$_{\text{DBPL}}$ programmer cannot use existing development tools like Eclipse [90], since there is no **XOBE**$_{\text{DBPL}}$ editor for example. Another problem in the context of the actual **XOBE**$_{\text{DBPL}}$ implementation is its dependency on the actual Java syntax. When the new Java version 1.5 was introduced a new grammar, parser and new analyser for **XOBE**$_{\text{DBPL}}$ had to be implemented. In the future this might happen again. A third problem is that the **XOBE**$_{\text{DBPL}}$ program analyser and code generator work on Java files ending with `.java` which are rather high level. Types of methods, objects etc. have to be found and determined first. The syntax is diverse and offers a lot of operators and possibilities. Consequently, it is very difficult to write and implement the **XOBE**$_{\text{DBPL}}$ analyser and code transformator.

From this point of view the question arises if it is necessary to extend the existing object-oriented programming language (Java) syntactically anyway. Or the other way round, could it be sufficient to realize all **XOBE**$_{\text{DBPL}}$ concepts introduced so far by merely extending Java semantically. If this could be achieved, all problems mentioned above do not occur anymore.

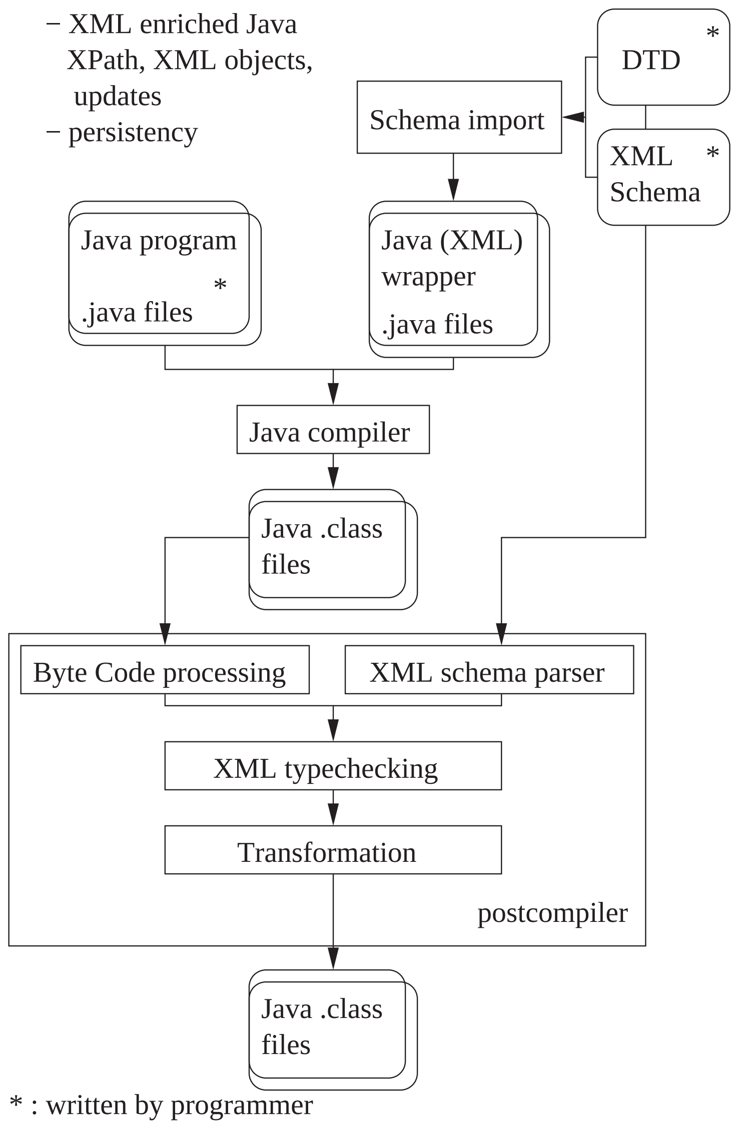
Indeed it is possible to keep the **XOBE**$_{\text{DBPL}}$ concepts without changing Java's syntax. The following sections will explain this for each concept. There is only one limitation regarding static type checking of XML list types. This limitation is discussed in section 8.1.3.

The main realization idea is to use annotations introduced in Java 1.5 [85] and to analyse and manipulate the Java byte code by a **XOBE**$_{\text{DBPL}}$ postprocessor based on an existing Java framework called **B**yte **C**ode **E**ngineering **L**ibrary (BCEL) [3] . A short introduction is given as well in section 2.1.

It is important to notice that all concepts and in particular the static type checking algorithm and type inference rules and the communication with the **XOBE**$_{\text{DBPL}}$ persistency layer remain unchanged.

Before going into the details, the overall architecture of the **XOBE**$_{\text{DBPL}}$ postprocessor and related tools can be seen in figure 8.1. This architecture enables to realize the XML integration concepts. The `Schema import` makes DTD and XML Schema instances available to the Java programmer by generating corresponding wrapper classes. The programmer can then write Java programs using XML objects, XPath, updates and persistency aspects. The `Java compiler` produces the associated set of `.class` files which in turn are used as input to the `postcompiler`. Besides this, the postcompiler also takes the DTD and XML Schema instances as input. After `Byte Code processing` and `XML schema parser` the `XML typechecking` component statically checks the programs. Finally the `Transformation` component transforms the original into a new set of Java class files, which solely generate and work with valid XML and may be persistency capable. According to the architecture in figure 8.1 the first section introduces the XML integration including XML schema import, XPath queries and updates. The second section explains the realization of persistency concepts using annotations. Further details can also be found in [70].

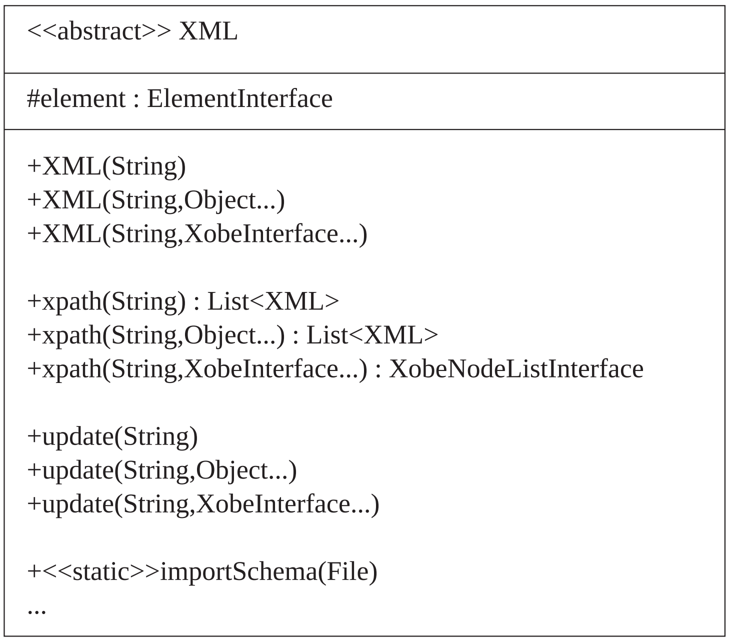
Figure 8.1: XML integration concepts



8.1 XML Integration

The main idea regarding the integration of XML is that all XML objects are castable to a common XML supertype. Every XML element type has got a corresponding Java type. In contrast to other approaches discussed in section 2.9 XML classes are only provided to offer Java constructor invocation to instantiate XML instances. Solely the supertype offers basic functionality and no default code. It is implemented as an abstract Java class named `XML`. The corresponding class diagram is shown in figure 8.2.

Figure 8.2: The XML super class


<<abstract>> XML
#element : ElementInterface
+XML(String)
+XML(String,Object...)
+XML(String,XobeInterface...)
+xpath(String) : List<XML>
+xpath(String,Object...) : List<XML>
+xpath(String,XobeInterface...) : XobeNodeListInterface
+update(String)
+update(String,Object...)
+update(String,XobeInterface...)
+<<static>>importSchema(File)
...

The `XML` class uses the `varargs` functionality introduced in Java 1.5 which allows multiple arguments to be passed as parameters to methods. It requires the simple `...` notation for the method to accept the argument list. Besides constructors an `xpath` method is offered for queries and an `update` method for updates, respectively. It is important to notice that there are three variants for each method. The programmer is supposed to use the variant containing the single string parameter signature. The second variant is used during the type checking process at compile time, while the third one is executed at runtime. Moreover, the first two versions only contain default bodies. Only the third one, which is really executed, contains the true code and therefore the needed functionality. The second important idea behind the XML super class is to provide the user the possibility to formulate XML element constructors, XPath queries and updates as before. This means that this can be done in a high-level and transparent syntax already known from **XOBE**$_{\text{DBPL}}$. But **XOBE**$_{\text{DBPL}}$'s main advantage is checking XML constituents at compile time. Therefore, the strings must be transformed with the help of BCEL into parameters of the second method version. After the type checking process is done, these method invocations are once again transformed into invocations of the third variant. At runtime both realization approaches, the one extending Java's syntax and this one, are equivalent.

In the following sections XML constructors, XPath queries and updates are introduced. This time they are realized without syntax extensions but still are statically type checked according to the corresponding schema. Last, the persistency and transaction realization using annotations is introduced.

8.1.1 XML Schema Import

An XML schema is made available by importing it. Importing means that for each declared element type a corresponding wrapper class is generated. For this purpose the `XML` super class offers a static method called `importSchema`. The method accepts a file containing the desired XML schema. The schema can either be a DTD or XML Schema instance. The method's code can be seen in listing 8.1.

Listing 8.1: The method to import XML schemas

```
1 public static void importSchema(File file){
2  HedgeGrammar grammar = new HedgeGrammar(file);
3  Map environment = grammar.getEnvironment();
4  Iterator iter = environment.values().iterator();
5  while(iter.hasNext()){
6   //generate a wrapper class for
7   //every element type
8  }
9 }
```

Concepts behind XML schemas remain unchanged. In line 2 the XML schema is transformed into its hedge grammar representation. Then a while-loop in line 5-8 iterates over all declared XML element types. Within the loop a wrapper class for each element is generated. The wrapper class is derived from the `XML` super class and is named according to its fully qualified name. The schema information is preserved by adding a `Schema` annotation. Its `uri` attribute references the corresponding XML schema file.

Let's take the `auction` DTD for an example. The generated `site` class `site` is shown in listing 8.2.

Listing 8.2: The generated wrapper class for the XML element `site`

```
...
@Schema (uri="auction.dtd")
public class site extends XML {
 public site(String xml){
  super(xml);
 }

 public site(String xml,Object... variables){
  super(xml,variables);
 }
}
```

After importing an XML schema, the programmer can start to invoke XML constructors, XPath queries and updates.

8.1.2 XML Objects

An XML object of a certain element type can be instantiated by invoking the single string parameter constructor. The string parameter will be checked syntactically and semantically at compile time against the XML schema that is referenced by its XML class annotation.

Construction

Let's examine an example given in listing 8.3. The example shows how to create an XML element `person`.

Listing 8.3: **XOBE**$_{DBPL}$ Java method `createPerson`

```
 1 person createPerson(String name){
 2       person p;
 3       ...
 4       ...
 5
 6       //new person is created
 7       p = new person("<person␣@id='"+generateAnID()+"'>"+
 8                      "<name>"+name+"</name>"+
 9                      "<emailaddress>an␣email<emailaddress>"+
10                    "</person>");
11
12       return p;
13 }
```

Compared to the old version the **XOBE**$_{DBPL}$ constructor is passed as a string parameter in lines 7-10. It is important to notice that except XML element contents or attribute values the constructor string must consist of constants. Otherwise, a static error is thrown, since it would be impossible to check its correctness at compile time. The fact that XML constructor strings are statically checked and validated against an XML schema implicates a semantically enhanced Java.

Variables and in particular previously defined other XML objects, or arbitrary expressions may be put as element content or attribute value within the constant string. In the given example two variable parts namely the result of the `generateAnID` method and the passed `name` parameter are contained. Examining the `person` constructor from a syntactic point of view, it is pure Java. What is new is that this constructor will be treated differently regarding its semantics. The string must conform to an XML element, it must also be valid according to the declared XML schema and non-constant parts of the string are treated as XML variables and their types are checked too. Checking happens at compile time and is explained in the next subsection.

Static Type Checking

Every part of the XML constructor string must be constant except for the XML variables. Therefore it is easy to transform the constructor invocation into the second constructor variant, which requires a string and an object array. The new string parameter contains the constant string parts, while the object array contains the variable parts. The order of their appearance in the original string is preserved. The transformation process happens at compile time. The version for the example given in listing 8.3 can be seen in listing 8.4.

Listing 8.4: **XOBE**$_{\text{DBPL}}$ Java method `createPerson` already transformed for the static type checking process

```
 1 person createPerson(String name){
 2       person p;
 3       ...
 4       ...
 5
 6       //new person is created
 7       p = new person( ”<person␣@id=’{}’>”+
 8                      ”<name>{}</name>”+
 9                      ”<emailaddress>an␣email<emailaddress>”+
10                     ”</person>”,
11                     new Object[]{generateAnID(),name}
12                    );
13
14       return p;
15 }
```

Places of variable parts within the constant string remain marked with the help of a pair of brackets ({}). In the above example there are two pairs of brackets within the constant string part, one for the `generateID` method in line 7 and one for the name in line 8. This order is preserved among the objects being part of the second array parameter in line 11. After this transformation static type checking can be done.

The byte code representing the constructor invocation in listing 8.4 is shown below in listing 8.5. The byte code is later executed by a stack machine. Please notice that high-level type information is still available. While a `new` instruction creates a new object the `anewarray` instruction creates a new array of given size and type. `Ldc` pushes a constant and `iconst` pushes an integer constant. An `aload` instruction pushes a local object variable. An `aastore` stores a value to a given object array's position. Finally `invokevirtual` and `invokespecial` invoke a method or a constructor respectively upon an object with given parameters.

Listing 8.5: Byte code representation of the `person` constructor invocation

```
 6: new person
 ...
10: ldc (<person @id='{}'><name>{}</name>
          <email>an email</email></person>)
12: iconst 2
13: anewarray java.lang.Object
...
17: iconst 0
18: aload this
19: invokevirtual generateAnID() : String
22: aastore
...
24: iconst 1
25: aload name : String
26: aastore
27: invokespecial person(String,Object[])
```

At 6: a new person object is generated and finally instantiated by invoking the constructor with the signature `person(String,Object[])` at 27:. The constant XML constructor with the placeholders loaded at 10: is passed as string parameter. An object array of size 2 is generated by 12: and 13:. Its first component at 17: contains the result of the `generateAnID` method invocation as can be seen at 18:,19: and 22:. Finally the object array's second position at 24: contains the local variable `name` at 25: and 26:. The object array is passed as second parameter to the `person` constructor. The information within the byte code is sufficient for statically type checking if this `person` XML constructor invocation generates a valid XML `person` element. The type checking process starts by parsing the constant string and continues inferring its type. Each time the type inference process detects a pair of brackets the corresponding type of the variable part is looked up, which is the declared type of a variable or a method's return type. In the given example the first pair of brackets is detected as id attribute value. According to the information extracted from the byte code the corresponding attribute value type can be inferred as string, since this is the return type of the `generateAnID` method. The second variable part which defines the `name` element's content type is looked up as string, too, since this is the type of the local variable `name`. It is finally checked if the whole inferred type is a valid subtype of the constructor's XML schema type. In the given example the inferred type in regular hedge type notation is **person**[**@id**[string];**name**[string];**email**[string]]. Thus, the XML element person constructor is checked successfully.

Please note that in contrast to the syntactically enhanced **XOBE**$_{\text{DBPL}}$ version it is no longer necessary to check if the left type of the assignment in line 7 is a valid super type of its right hand part. This is now checked by the Java compiler. Instead it is only necessary to check if the XML element constructed by the parameters is a valid subtype of the constructor's XML type. XML constructors can be easily recognized with the help of BCEL. After parsing the string and getting the XML element syntax tree, the well known type inference rules and type checking algorithm can be applied without modification. Please notice that all variable parts being part of the object array have got well-defined types which are easily available using BCEL and the Java Virtual Machine's constant pool.

After the type checking process a second compile time transformation process occurs. All specific XML types are mapped to the general XML DOM-like model that is also used in the syntactically enhanced **XOBE**$_{\text{DBPL}}$ version. The super interface for all **XOBE**$_{\text{DBPL}}$ objects and lists is named `XobeInterface`. Consequently, at runtime every XML constructor invokes the third constructor of the XML super class listed in figure 8.2. The corresponding code is given in listing 8.6.

Listing 8.6: The XML constructor invoked at runtime

```
 1 public XML(String xml, XobeInterface... variables){
 2  XobeParser parser = new XobeParser(new StringReader(xml));
 3  XMLElement element = null;
 4  try{
 5   element = parser.XMLElement();
 6   XMLElementTrans trans;
 7   //transform the XML element...
 8   this.element = trans.getElement();
 9  }catch(ParseException e){
10    ...
11  }
12 }
```

In line 2 the **XOBE**$_{\text{DBPL}}$ parser parses the constant XML string which constructs the XML object. Since the string is already checked at compile time, this operation is guaranteed to be safe. The parser produces the corresponding XML element syntax tree in line 5. A second transformation process in line 6-8 transforms the runtime XML object. The runtime XML object referenced by the `element` attribute in line 8 is a DOM like object and uses the same classes as before in **XOBE**$_{\text{DBPL}}$.

8.1.3 Queries in XPath

Since all XML classes are derived from the superclass `XML` shown in figure 8.2, XPath queries are designed as method invocations upon XML objects. The `xpath` method, requiring one string parameter, is provided for this purpose. Once again, the programmer formulates an XPath query as a constant string. Analogously to an XML constructor invocation XML variables might be included and referenced. In contrast to the syntactically enhanced **XOBE**$_{\text{DBPL}}$ version the context node is implicitly given. It is the XML object the method is invoked upon.

Invocation

An example retrieving all person elements with a given id is given in listing 8.7.

Listing 8.7: An XPath query

```
//List<person> plist = (List<person>)
//auctionSite.xpath(
// "//person[@id='{"+p_id + "}']");
List<XML> plist =
  auctionSite.xpath(
    "//person[@id='{"+p_id+"}']");
```

The result of an XPath expression is always a list of XML objects. Once again the query string must consist of constant string parts except attribute values or element contents. Otherwise a static error is thrown. Like in case of XML constructors the single string `xpath` variant is transformed into the two parameter version. The string is split into its constant and variable constituents and type checking is performed analogously. In the case of XPath queries, types are often limited to the general generic list type `List<XML>`. Unfortunately, it is not possible to cast generic list types, as implied by the comment in listing 8.7. If we assume the following lines of code in listing 8.8 with `plist` as defined and instantiated in listing 8.7, static type checking will raise a warning in line 2. According to the auction DTD a people element contains a sequence of `person` elements. Nevertheless, lines 4-6 will succeed because the true inferred type is still available and can be taken into account.

Listing 8.8: XPath result types

```
1 people p =
2   new people("<people>{"+plist+"}</people>");
3
4 p = new people("<people>{"+
5   auctionSite.xpath(
6     "//person[@id='{"+p_id+"}']")
7                   +"</people>");
```

The above limitation is caused by the Java language itself. We hope that in the future generic lists will be castable as one might expect. In general XPath types are more complex than Java types and have to be mapped to the generic XML list type anyway.

Static Type Checking

XPath result types, in this new semantically enhanced **XOBE**$_{\text{DBPL}}$ version, are more limited than in the previous version. In the previous version it is possible to specify among others a choice type, for example `xml<(person|name)* >`. This type can only be expressed as `List<XML>` which is the super type of the original version. Since Java does not know such kind of result types, it is not available for the programmer anymore. On the one side this is a true limitation and using the supertype version might result in false negative warnings by the static type checking process. On the other side these cases are rather unimportant in practice and it is more important to use pure Java. Moreover, if an XPath expression resulting in such a choice type is directly included inside an XML object constructor invocation for example, the type is inferred as choice type and treated as such. An example is given above in listing 8.8.

To prepare XPath invocations for the type checking process the string parameter is separated analogously to the XML object constructor into two parameters. The string parameter contains all constant parts and labels for the variable parts. The object array contains all variable references in the same order as they appear within the former string. In case of the given XPath example the transformation process results in lines of code shown in listing 8.9.

Listing 8.9: The XPath example transformed for the type checking process

```
...
List<XML> plist =
 auctionSite.xpath("//person[/@id='{}']",new Object[]{p_id});
if(plist.size()>0)return null;
...
```

After parsing the string and getting the XPath syntax tree the well known type inference rules and type checking algorithm can be applied without further modification.

At runtime the method is invoked upon an XML object. The variables are then guaranteed to be of type `XobeInterface` due to the XML and persistency transformation process known from the syntactically enhanced **XOBE**$_{\text{DBPL}}$ version. Hence, the following `xpath` method of the `XML` super class is invoked. Its code can be seen in listing 8.10.

Listing 8.10: XPath method invoked at runtime

```
public XobeNodeListInterface xpath(String xpath,
                                  XobeInterface... variables){
 XobeParser parser = new XobeParser(new StringReader(xpath));
 XPathExpression path = null;
 try{
  path = parser.XPathExpression();
  XPathTrans trans;
  //transform the XPath expression...
  LocationPathDescriptor descriptor = trans.getPath();
  return this.element.evaluateLocationPath(descriptor);
11   }catch(ParseException e){
12     ...
13   }
14   return null;
15 }
```

At runtime an xpath method returns a `XobeNodeListInterface` type. This is also due to transformation steps happened at compile time. In line 2 the XPath query is parsed. The operation is safe, since the constant string is checked at compile time. The XPath expression syntax tree is extracted and transformed into an XPath location path descriptor in lines 5-8. XPath descriptors are introduced and explained in chapter 7. Finally the XPath query is executed upon the context XML element in line 9. The result is returned.

8.1.4 Updates

The last section concerning the XML integration focuses on XML updates. Updates do not have a return value. Analogously to XPath queries updates are realized by method invocations upon XML objects. Once again, the update expression is passed as string parameter which is checked statically at compile time. Therefore the update must be formulated as a constant string and only contents supposed to be inserted or replaced, for example, can be passed as variable parts.

Invocation

Let's formulate the update example from listing 3.3 chapter 3 in semantically enhanced pure Java syntax. The result of this **XOBE**$_{DBPL}$ version can be seen in listing 8.11.

Listing 8.11: The update example in case of the semantically enhanced **XOBE**$_{DBPL}$ version

```
synchronized int bid(String p_id, int incr, String a_id){
 //calculate new current
 List<XML> cur =
  auctionSite.xpath("//open_auction[/@id='{"+a_id+"}']/current");
     current new_current = new current(
   "<current>{"+
    (Integer.parseInt(cur.get(0))+incr)+
   "}</current>"
  );

     //create new bidder
     bidder bid = new bidder("<bidder>"+
                    "<date>{"+getDate()+"}</date>"+
                    "<time>{"+getTime()+"}</time>"+
                    "<personref␣person={"+p_id+"}/>"+
16                          "<increase>{"+incr+"}</increase>"+
17                        "</bidder>");
18
19   //update auction
20   auctionSite.update("LET␣i:=␣//open_auction[/@id={a_id}]"+
21                          "UPDATE␣i␣INSERT␣{"+bid+"}␣BEFORE␣/current,"+
22                          "REPLACE␣/current␣WITH␣{"+new_current+"}");
23   return new_current;
24 }
```

The first part of the example including lines 1-19 shows an XML object constructor and XPath expressions as described before. This is only supposed to give an impression of a slightly more complex example. The new update method invocation appears in line 20-22. The string is a one to one transformation of the former **XOBE**$_{\text{DBPL}}$ example. Like in case of xpath expressions the context element object is implicitly defined by the object the update method is invoked upon. In the given example it is the auction site object. Consequently, the path expression defining the local variable `i` can be formulated as a relative location path.

Static Type Checking

To perform static type checking in case of update expressions it is required to transform the single parameter invocation into a two parameter invocation. The first parameter contains once again the static parts of the update string and labels where variables are inserted. The second parameter is the object array containing the variable parts of the former string. In case of the example update expression the invocation results in the one presented in listing 8.12.

Listing 8.12: The update example ready to perform type checking

```
//update auction
auctionSite.update("LET␣i:=␣//open_auction[/@id={}]"+
                       "UPDATE␣i␣INSERT␣{}␣BEFORE␣/current,"+
                       "REPLACE␣/current␣WITH␣{}",
                       new Object[]{a_id,bid,new_current});
```

For static type checking the constant string parameter is parsed. The validity of this update expression can be detected by determining the corresponding variable types and method signatures. Type inference rules and the type checking algorithm remain unchanged.

At runtime an update method is invoked upon an XML object and the variable parameters of the array are then of the type `XobeInterface`. This happens due to the XML and persistency transformation step. The code of the runtime update method is shown in listing 8.13.

Listing 8.13: The update method invoked at runtime

```
public void update(String update,
                       XobeInterface... variables){
  XobeParser parser =
 4    new XobeParser(new StringReader(update));
 5   try{
 6    UpdateClause uclause = parser.UpdateClause();
 7    UpdateClauseTrans trans;
 8    //transform the update...
 9    Update updates = trans.getUpdate();
10    for(UpdateDescriptor u : updates.getUpdates()){
11     this.element.update(u);
12    }
13   }catch(ParseException e){
14    ...
15   }
16 }
```

The update is parsed in lines 3-4 and the corresponding syntax tree is extracted in line 6. The parsing process is safe, since the string is checked at compile time. Then this tree is transformed into a list of update descriptors in the lines 7-9. Update descriptors are explained and introduced in chapter 7. Finally, each basic update operation is invoked consecutively upon the XML object. The update execution happens in lines 10-12.

8.2 Persistency

The persistency concept of **XOBE**$_{DBPL}$ is realized using annotations. Instead of extending Java's syntax with the additional keyword **database** a persistent class is now declared by adding a specific class annotation.

8.2.1 Declaration

A persistent environment is now signed with the **Persistent** annotation tag. The database declaration of the syntactically enhanced **XOBE**$_{DBPL}$ variant is therefore no longer needed. If we want to declare the `AuctionSite` class as persistent environment as it is done in example 4.1.1 previously the version using an annotation looks like the one shown in listing 8.14.

Listing 8.14: Declaring the persistent `AuctionSite` environment

```
@Persistent
public class AuctionSite{
 ...
}
```

Invoking an available constructor will cause the generation of a persistent object of type `AuctionSite`. The creation of persistent objects remains the same. The retrieval of existing persistent objects still happens by a special type of XPath expressions. For this purpose the already introduced `XML` super class provides a static method `xpath` that can be invoked by passing the desired class object and an XPath expression as queries. This time the class object is implicitly taken as the context variable. The class object is of the type `java.lang.Class`. The rest is analogous as described in section 4.1.1. The example given in 4.1.3 is then translated into the following version shown in listing 8.15. Please notice that only the changed lines are given.

Listing 8.15: Retrieving of arbitrary persistent objects

```
List<XML> auctionSites =
 XML.xpath(AuctionSite.class,"[/description='{"+description+"}']");
```

The resulting list contains all auction site objects which are already persistently kept and which have got the specific description value.

If objects of classes that are marked as persistent are supposed to be destructible the programmer has to add a default `delete` method with an empty body. The method doesn't have a return type and takes no parameters. Moreover, the method is marked with a `Delete` annotation. An example can be seen in listing 8.16.

Listing 8.16: Declaring a delete method

```
@Delete
public void delete(){
}
```

A suitable body is added during the transformation process taking place at compile time. Consequently, the example demonstrating the different deletion types given in 4.1.4 remain semantically and syntactically unchanged.

8.2.2 Realization Concepts

The concepts behind persistency remain unchanged except that interface, transient and persistent variants are no longer generated as Java classes explicitly. Only their `.class` versions are generated using BCEL. Details about the realization concepts concerning persistency can be found in section 4.2.

8.3 Transactions

Transactions are realized using annotations as well. The **XOBE**$_{\text{DBPL}}$ version extends Java's syntax with a transaction statement. By contrast, the semantically enhanced version provides the possibility to label methods with a specific annotation. In **XOBE**$_{\text{DBPL}}$ flat, nested and distributed transactions are supported. The latter is out of the programmer's control. Distributed transactions solely depend on the underlying persistency layer.

8.3.1 Declaration

A method which is supposed to be executed as a transaction is marked with the **Transaction** annotation tag. In general the expression power of a transactional method is not weaker than that of a transaction statement. This is because every transaction statement may be separated into an extra method. The transaction example given in listing 5.1 chapter 5 is written analogously in the semantically enhanced **XOBE**$_{\text{DBPL}}$ variant in listing 8.17. The body illustrated by three dots is already given in listing 8.11.

Listing 8.17: A transaction example

```
@Transaction
synchronized int bid(String p_id,int incr,String a_id){
...
}
```

Besides flat transactions, it is also possible to realize nested transactions by annotations. Nested transactions are formulated if a transaction annotated method is invoked from within a transaction method. Distributed transactions are provided implicitly by the underlying persistency layer in **XOBE**$_{\text{DBPL}}$.

8.3.2 Realization Concepts

Realization concepts in context of transactions don't change compared to the syntactically enhanced **XOBE**$_{\text{DBPL}}$ version. More details can be found in section 5.2.1 and section 5.3.1.

Chapter 9

Experimental Results

In this chapter some experimental results using the **XOBE**$_{\text{DBPL}}$ prototypical implementation are introduced. The first section focuses on tests in the context of the XML integration and statically type checked update operations. The second section describes tests regarding the transparent and type independent integration of persistency concepts in **XOBE**$_{\text{DBPL}}$.

9.1 XML Updates

We tested the **XOBE**$_{\text{DBPL}}$ prototype implementation on four XML documents generated by the XMark project's `xmlgen` generator. In the context of XML update testing we used rather small scaling factors for `xmlgen` resulting in XML documents with sizes reaching from kilobytes to three megabytes. Moreover we defined four representative updates and measured times to validate and execute them. Validation and execution times of the **XOBE**$_{\text{DBPL}}$ prototype implementation were compared with three other approaches all referenced in sections 2.9 and 2.10 about corresponding related approaches. Validation of updates in the context of the candidates using DOM , e.g. Infonyte [36] and Xindice , is tested at runtime by parsing the whole document against the schema. Contrarily, **XOBE**$_{\text{DBPL}}$ and Xact are based on static validation, which is independent of the XML document's size. The four updates that are chosen here are defined as follows.

Update 1. deletes an existing person of the auction site by its id.

Update 2. deletes all closed auctions elements of the auction site.

Update 3. inserts a new person into the people element of the auction site.

Update 4. is an update operation consisting of a delete operation as defined in the first update and an insert as defined in update 3.

XMark scaling factor	**XOBE**$_{\text{DBPL}}$	Xact	DOM+Infonyte	Xindice
0.0	0.17/0.23	13.5/13.64	-/0.36	-/0.37
0.01	0.17/0.52	13.5/13.64	-/0.69	-/2.09
0.02	0.17/0.83	13.5/13.64	-/0.87	-/2.67
0.03	0.17/1.06	13.5/13.67	-/1.07	-/3.96

Table 9.1: Update 1. Deletion of an existing person by its id.

XMark scaling factor	**XOBE**$_{\text{DBPL}}$	Xact	DOM+Infonyte	Xindice
0.0	0.17/0.19	11.7/11.87	-/0.36	-/0.45
0.01	0.17/0.19	11.7/12.17	-/0.99	-/1.49
0.02	0.17/0.19	11.7/12.47	-/1.48	-/2.57
0.03	0.17/0.19	11.7/12.7	-/3.96	-/4.76

Table 9.2: Update 2. Deletion of all closed auction elements.

The four tables 9.1,9.2,9.3 and 9.4 present the results. The four rows represent the four different sized XML documents generated by the XMark generator. Each record consists of two numbers representing static validation time as well as the execution time of the valid update, measured in seconds respectively. Latter times do not include former times. A **'-'** for static validation time indicates that static validation is not supported. The leftmost column contains the scaling factors of the XMark project. A scaling factor of 1.0 produces an XML auction schema instance in the size of 100 MB, a scaling factor of 0.1 consequently leads to a 10 MB sized document and so forth. In particular the scaling factor 0.0 generates a minimum XML document according to the auction schema. As we limit the size of the largest XML document to 3 Megabytes all operations were able to take place in main memory. This was required because Xact does not contain a database connection, but is the only other available prototype supporting static validation and XML manipulation. Consequently, measured times are not influenced by times accessing hard disk. Moreover, times to load and/or store the documents are not included.
The experimental results show that **XOBE**$_{\text{DBPL}}$ is very well suited to replace existing approaches based on DOM and rather low-level APIs. In particular, the times needed to statically validate the four updates are very small and much better than Xact's. Even for the chosen, rather small XML documents, approaches like `Infonyte+DOM` and `Xindice` suffer from the time needed to check the validity of updates at runtime. And contrary to static validation times, these times will grow as the size of the documents grow.
We ran our tests using Sun's Java 1.4.1 virtual machine on a 2.0 GHz Pentium 4 with 768 MB RAM. Each test was run repeatedly to get average times.

XMark scaling factor	**XOBE**$_{\text{DBPL}}$	Xact	DOM+Infonyte	Xindice
0.0	0.17/0.19	13.34/13.54	-/0.35	-/0.47
0.01	0.17/0.23	13.34/14.24	-/0.95	-/1.32
0.02	0.17/0.24	13.34/14.94	-/1.31	-/2.37
0.03	0.17/0.25	13.34/15.14	-/1.66	-/3.96

Table 9.3: Update 3. Insert of a new person into the people element.

XMark scaling factor	**XOBE**$_{\text{DBPL}}$	Xact	DOM+Infonyte	Xindice
0.0	0.17/0.4	22.14/22.35	-/0.52	-/0.61
0.01	0.17/0.73	22.14/22.37	-/1.47	-/3.18
0.02	0.17/1.1	22.14/24.64	-/2.74	-/5.24
0.03	0.17/1.3	22.14/27.24	-/3.62	-/7.52

Table 9.4: Update 4. Update operations 1 and 3 are combined.

9.2 Distributed and Persistent Objects

This section summarizes the results we gained while testing the **XOBE**$_{\text{DBPL}}$ prototype implementation concerning persistency. The example application deals with the auction site scenario used throughout this work. To compare **XOBE**$_{\text{DBPL}}$ regarding persistency concepts and realization we chose the following three other candidates which are implemented in Java.

- The EJB 2.1 implementation used in JBoss [79, 39],
- Hibernate [32] and
- the JDO implementation JPOX [80, 41].

As introduced in section 2.12 EJB provides persistent Java objects which are managed by a server. If a client connects to a persistent object, operations, parameters etc. have to be serialized. How these objects are kept persistent is not limited by the specification. Moreover, EJBs are rather complicated to use and learn. In addition, neither version 2.1 nor 3.x support all object-oriented concepts regarding the persistent components. Thus, this example application cannot deal with inheritance. Contrastly, Hibernate offers a tool to ease the mapping from objects to relations. Hibernate is limited to using relational databases. The JDO specification is a persistence solution for Java objects on the client side. In general, JDO can be used in connection with arbitrary types of databases. In the case of a relational database the mapping has to be described by an XML file. All approaches require explicit code dealing with the corresponding persistency manager. JBoss, JDO and Hibernate used the relational database product HSQLDB [91] as storage for persistent objects. Moreover, the database server and the auction site program were executed on the same host. **XOBE**$_{\text{DBPL}}$ is transparent and does support all object-oriented concepts. Since the persistency layer is developed for distributed applications which are not solely limited to Java, persistent objects have to be serialized to XML (SOAP messages). The client auction site application ran on another host. Moreover, at the moment **XOBE**$_{\text{DBPL}}$ clients use the Axis [2] implementation to communicate with the persistency layer's web service interface. **XOBE**$_{\text{DBPL}}$ is the only candidate in this scenario using serialization. Thus, times to generate XML from objects and vice versa were not taken into account. Consequently, times measured for **XOBE**$_{\text{DBPL}}$ are better comparable to those of the other candidates.

The tests consist of several operation types, in particular inserting, deleting, searching, retrieving and a mixture of them. In particular the auction example application offers the following selective characteristic methods:

Insert Inserting a new person into the auction site. Thus, the person becomes persistent as well.

Deletion Removing a registered person from the auction site.

Retrieval Retrieving a specific person from the auction site and printing its personal data.

Mixture 1 Starting a new auction. First, an item has to be referenced or created as well. Then the new auction can be created and inserted into the auction site's open auction list. The operation consists either of two inserts or one insert and one retrieval.

Mixture 2 Bidding for an open auction. The corresponding open auction's last bid has to be found and then a new bid has to be created and inserted according to the desired increase. In addition, a person must be referenced by each bid. The whole operation consists of a search, an update and an insert.

Mixture 3 Closing of all expired open auctions. Since open auctions contain a closing date and time, it must repeatedly be checked if some open auctions have to be turned into closed auctions. The whole operation consists of a search and a retrieval, a deletion, a creation and an insert.

The auction site application offers the above operations and was implemented using each of the four candidates. Then the operations were executed using the four different implementations. The following criteria were checked:

Performance The execution time of the above operations.

Transparency in particular regarding database or persistency layer access.

Expressiveness in particular regarding the availability of object-oriented concepts like inheritance. In addition, the possibility to develop distributed applications including exchanges with applications written in other object-oriented programming languages was also evaluated.

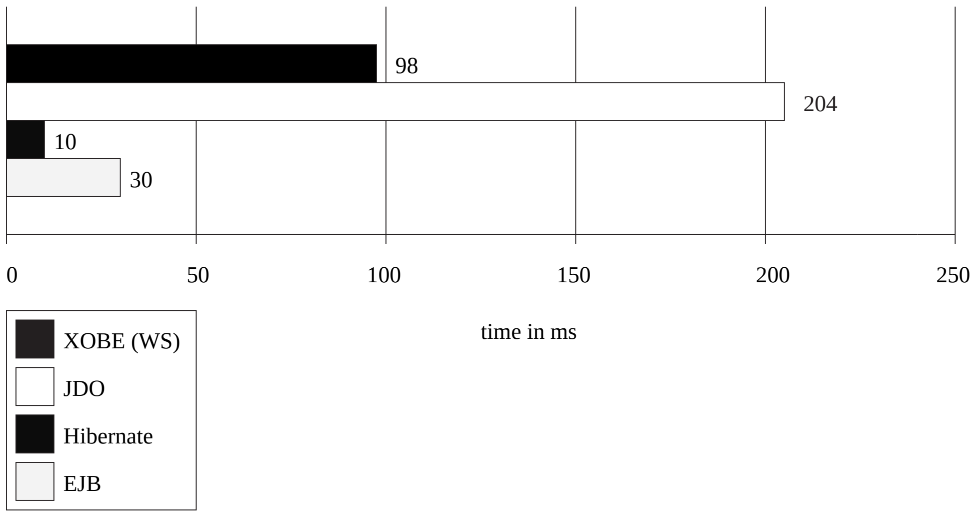
Usability concerning the ease of use, clarity, reusability and the retrieval of persistent objects.

The only test parameter was chosen to be the starting size of the auction site XML document reaching from 35 KB to 1 MB. We ran our tests using Sun's Java 1.4.2 virtual machine on a 2.0 GHz Pentium 4 with 768 MB RAM. Each test was repeated several times to get average times. The resulting time diagrams are given in figures 9.1, 9.2, 9.3, 9.4, 9.5 and 9.6.

Transparency, expressiveness and usability results concerning the four candidates are summarized by table 9.5.

After all the tests were run, **XOBE**$_{\text{DBPL}}$ has got the highest degree regarding transparency, expressiveness and usability. EJB suffers from complicated components and doesn't support all object-oriented features, in particular inheritance. JDO, Hibernate and EJB require explicit code to deal with transaction or persistency managers. In **XOBE**$_{\text{DBPL}}$ this is fully transparent to the programmer. The retrieval of persistent objects is done by certain kinds of query languages. **XOBE**$_{\text{DBPL}}$ provides type checked XPath queries. JDO, EJB and Hibernate offer SQL which is of course not type checked since queries are passed as simple strings. Moreover, Hibernate also offers HQL, which is similar to OQL [58], and example objects also called criteria queries. The latter are type checked and profit from the Java type system.

Figure 9.1: Inserting a new person

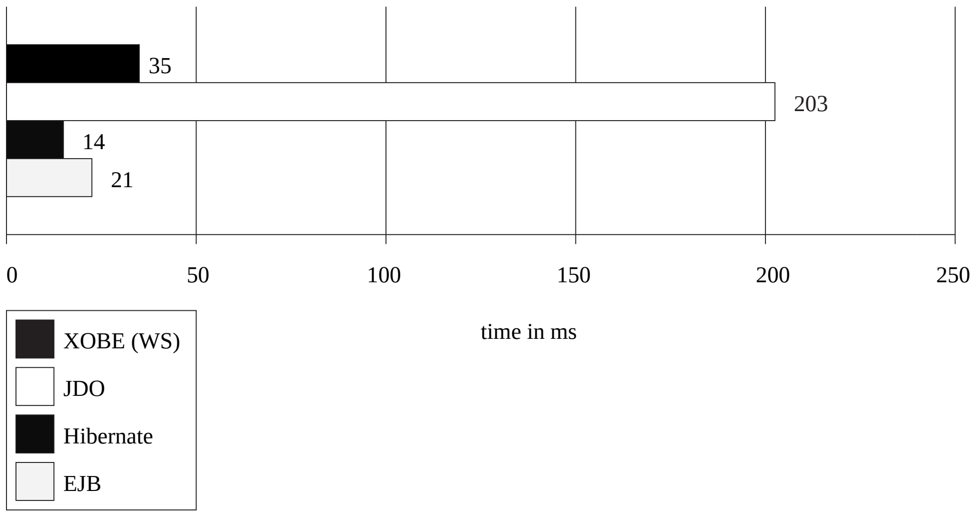


As a result one can say that each candidate has got its own usage scenario. Hibernate is probably best suited for non-distributed Java applications using relational databases as storage. For this rather simple but common scenario Hibernate provides straightforward support. In contrast to Hibernate JDO also supports other database paradigms, in particular object-oriented databases. EJB is a good solution for server-side persistent Java objects. EJB also offers support for distributed applications, but excludes distributed transactions. Moreover, EJB does not fully support object-oriented features, e.g. inheritance. Contrastly, **XOBE**$_{\text{DBPL}}$ is developed for distributed persistent objects and XML. **XOBE**$_{\text{DBPL}}$ is fully transparent and supports inheritance. In addition persistent and distributed objects can also be exchanged with other applications

		Hibernate	**JDO**	**EJB**	**XOBE**$_{\text{DBPL}}$
transparency	persistency mapping	$\oplus$	$\odot$	$\ominus$	$\oplus\oplus$
	transactions	$\odot$	$\odot$	$\oplus$	$\oplus\oplus$
	persistency layer access	$\odot$	$\odot$	$\odot$	$\oplus\oplus$
expressiveness	inheritance	$\oplus$	$\oplus$	$\oslash$/$\oplus$(EJB 3.x)	$\oplus\oplus$
	query languages	$\oplus$	$\odot$	$\odot$	$\oplus$
	distribution	$\oslash$	$\oslash$	$\oplus$	$\oplus\oplus$
	exchange	$\oslash$	$\oslash$	$\ominus$	$\oplus$
usability		$\oplus\oplus$	$\oplus$	$\ominus$	$\oplus$

Table 9.5: Transparency, expressiveness and usability results, with $\ominus\ominus$ very bad, $\ominus$ bad, $\odot$ ok, $\oplus$ well, $\oplus\oplus$ very well and $\oslash$ not supported at all

Figure 9.2: Removing a registered person



including non-Java applications.

It is important to notice that **XOBE**$_{\text{DBPL}}$'s performance can be significantly improved. In the following there are four main starting points. First, store operations sent to the persistency layer might be executed by threads, because the client program doesn't generally have to wait for an answer. Second, as mentioned before **XOBE**$_{\text{DBPL}}$ clients use the Axis implementation for web service communication. It comes out that Axis has rather got a weak performance. **XOBE**$_{\text{DBPL}}$ clients have only got to deal with the persistency layer web service. Therefore, it might be better to develop a specific web service client side which is only capable of sending and receive messages from the persistency layer's web service. Third, a more efficient consisteny model on client-side has to be found and implemented. A more efficient client-side consistency model can reduce communication with the persistency layer in general. Fourth and last, using XML and SOAP in particular causes rather large document or message sizes. This is due to the talkative nature of XML. There are several approaches dealing with the efficient binary compression of XML. One promising proposal is made by [107]. An outlook on future work and conclusions can also be found in chapter 10.

Figure 9.3: Retrieving and printing specific personal data

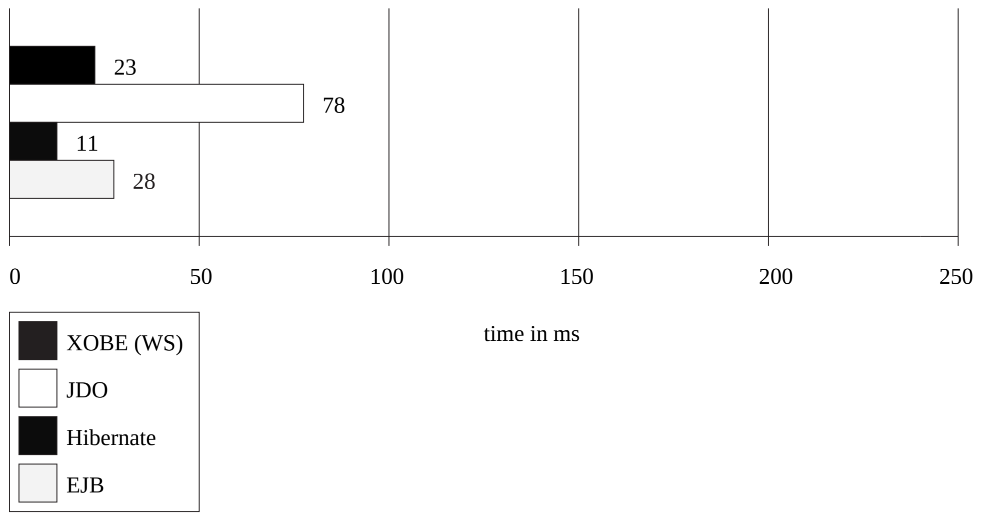


Figure 9.4: Starting a new auction

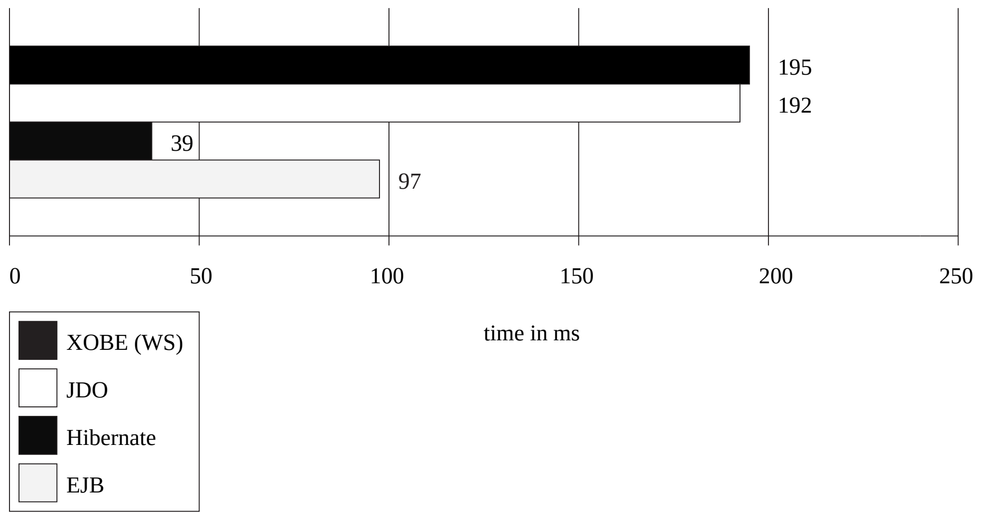


Figure 9.5: Bidding for an open auction

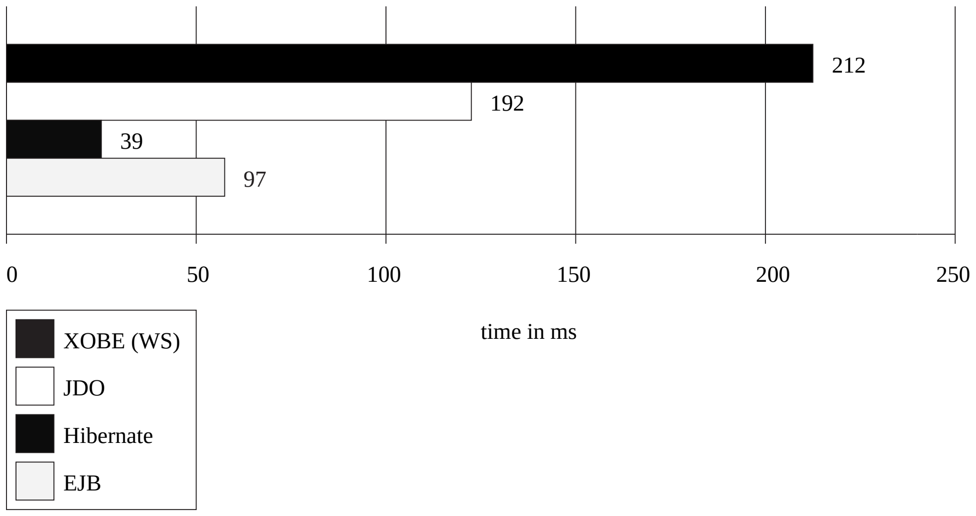
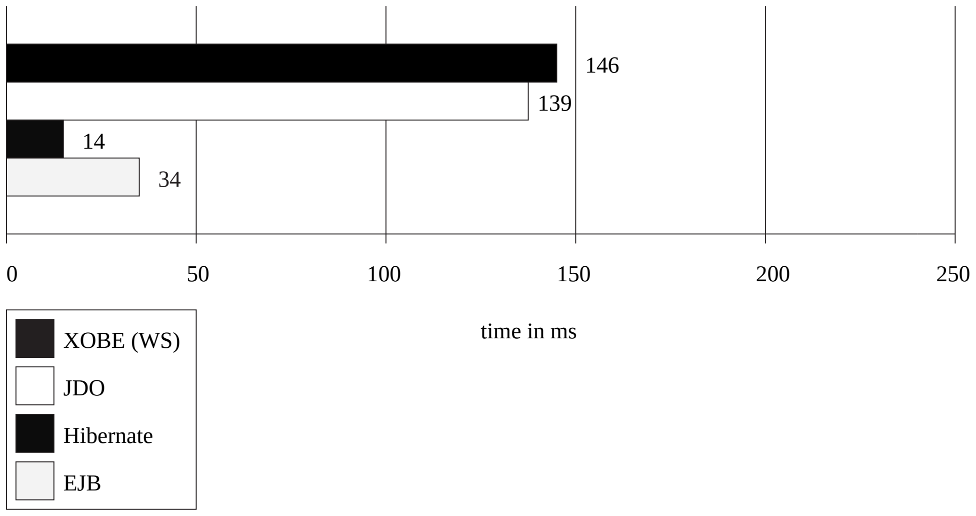


Figure 9.6: Closing of all expired auctions



Chapter 10

Conclusions and Future Work

This work introduces a database programming language for XML applications named **XOBE**$_{\text{DBPL}}$. As discussed in the introduction there is an increasing need for the seamless processing of XML in existing object-oriented programming languages. Moreover, persistency is a desired ability in programming languages as well. Both concepts should, if possible, be integrated on a high-level and transparently for the programmer. In **XOBE**$_{\text{DBPL}}$ the focus lies exactly on the integration of XML corresponding operations like queries and updates as well as persistency. For various reasons, of which two of them are popularity and availability, Java is chosen as the source object-oriented programming language. An introduction is given in chapter 1. Basics and related approaches in reference to XML and persistency can be found in chapter 2. **XOBE**$_{\text{DBPL}}$ is the successor project of **XOBE** which focuses on the development of web applications and integrates XML objects as well as simple XPath queries. XML constituents of a program are checked at compile time. In **XOBE**$_{\text{DBPL}}$ XML objects can now be manipulated ,too. XML updates and more complex queries are introduced and the type checking process is adapted. The extended XML integration in **XOBE**$_{\text{DBPL}}$ is introduced in chapter 3. The other main new topic in **XOBE**$_{\text{DBPL}}$ is the transparent integration of persistency. Persistency in **XOBE**$_{\text{DBPL}}$ is provided on a high-level, regardless of type and preserves object-oriented concepts like inheritance. The persistency concept in **XOBE**$_{\text{DBPL}}$ is explained in chapter 4. Chapter 5 introduces transactions which are closely connected to persistency. In **XOBE**$_{\text{DBPL}}$ persistency is realized using a persistency layer. The persistency layer is hidden from the programmer and is based on an approach dealing with web services. Thus, the persistency layer might also be used by both non-Java and Java applications. Persistent data is highly distributed and can be shared. Details are discussed in chapter 6. More details concerning the architecture and implementation are given in chapter 7. The original **XOBE**$_{\text{DBPL}}$ realization approach extends the Java programming language syntactically. Another solution is explained in chapter 8, which extends Java semantically without changing its syntax and the concepts for XML and persistency integration. Of course, it is more elegant to develop XML applications in **XOBE**$_{\text{DBPL}}$'s original variant, but existing development tools can't deal with the new syntax. Consequently **XOBE**$_{\text{DBPL}}$ specific plug-ins have to be developed and programmers need to learn a new syntax. The semantically enhanced **XOBE**$_{\text{DBPL}}$

realization approach circumvents these problems. In the future it is planned to develop both approaches in parallel.

Chapter 9 discusses some first experimental results concerning the integration of XML as well as the integration of persistency. In case of XML it is shown that the new type inference rules regarding updates are very efficient. Results in the context of persistency will be improved by applying more efficient XML (SOAP) processing tools and more efficient consistency models among others. As a result **XOBE**$_{\text{DBPL}}$, using a web service based persistency layer, is developed and suitable for distributed applications.

Future Work

Finally, there are some very interesting future topics to proceed and further develop **XOBE**$_{\text{DBPL}}$. Future work includes for example the efficient processing of XML and the aspect of keeping persistent data redundantly. Closely connected to redundancy is the management of efficient consistency. It is also desirable to deal with errors during transactions. Last but not least the persistency layer deals internally with relational databases. In the future it might be better to use so-called hybrid databases as well. Hybrid databases provide the possibility to store relational data as well as XML data natively.

XML Processing

As can be seen from the results in chapter 9 concerning persistent data it is important to develop the processing of XML and SOAP messages in general. The persistency layer based on web services and **XOBE**$_{\text{DBPL}}$ programs, exchanges data and messages via SOAP. SOAP is a specific XML language. On the one hand it should be analyzed if software approaches or new models can improve efficiency. On the other hand it is important to find out if hardware components capable of processing XML or SOAP in particular can improve performance. There are several approaches which compress XML documents and messages into a binary representation. Due to the talkative nature of XML there is a lot of overhead within XML and in particular SOAP messages. One promising approach is introduced in [107]. Another aspect references the representation of XML objects. XML objects in **XOBE**$_{\text{DBPL}}$ are represented using a DOM-like model at runtime. DOM significantly limits the size of these XML objects. Another model might be more suitable.

Redundancy

Up to now persistent objects in **XOBE**$_{\text{DBPL}}$ are stored without redundancy. In the future it might be suitable to introduce redundancy. Redundancy is important if components of the persistency layer fail and, just as important, it can improve performance. If persistent objects are supposed to be distributed and should be accessible by many applications, copies of these objects improve access times. In context of redundancy realization concepts concerning transactions and concurrency within the persistency layer need to be adapted as well.

Efficient Consistency

Closely connected to the redundancy is the consistency aspect. If objects or data are kept redundantly these copies, if updated, might become inconsistent. To minimize or respectively hide these effects from the applications, server-side consistency approaches are needed. Besides consistency in the context of redundant data, efficient consistency also plays an important role in the context of shared persistent data in general. As mentioned in section 4.2.1 and can be seen by analyzing the tests in section 9.2 there is an urgent need for a much more efficient client-side consistency realization concept. It is also planned to provide semi-automatic client-side consistency to the **XOBE**$_{\text{DBPL}}$ programmer. It should be possible to mark program points where persistent objects should be reloaded and respectively updated. The programmer must be able to define these points on a high-level. The usage of aspect-oriented programming [44] seems to be promising in this context.

Nested Transactions

Syntactically **XOBE**$_{\text{DBPL}}$ offers the possibility to formulate nested transactions. However, the distributed web-service-based persistency layer does not support these kind of transactions so far. Hence, future work may also include the implementation and support of nested transactions on the persistency layer level.

Errors and Transactions

Currently, it is assumed that transactions in **XOBE**$_{\text{DBPL}}$ are either committed successfully or aborted. The abortion of a transaction happens if one of the ACID properties can not be guaranteed any longer due to other concurrent transactions. Contrastly, there might be other reasons why a transaction fails or should be aborted. These reasons are errors which are possible since the persistency layer in **XOBE**$_{\text{DBPL}}$ is realized by a distributed network of arbitrary **XOBE** service nodes based on web services.

Applying Hybrid Databases

Finally, the persistency layer in **XOBE**$_{\text{DBPL}}$, although not limited to, uses relational databases internally. Relational databases are widely available and provide high performance. Nevertheless, a new paradigm of databases has been developed which are called hybrid databases. An example is SystemRX [8]. These new databases are able to store relational data as well as XML data natively. It would be very interesting to discover how these hybrid databases can be applied within the **XOBE**$_{\text{DBPL}}$ persistency layer.

Scalability

Another very interesting topic is scalability, regarding in particular the size of the underlying persistency layer and the number of clients. In the case of large scale persistency systems consistency and transaction models need to be analyzed and adapted in a suitable way. Scalability concerning the **XOBE**$_{\text{DBPL}}$ persistency service is definitely an ongoing research topic.

Index

Bibliography

[1] Scott W. Ambler. Mapping Objects To Relational Databases. `http://www.AmbySoft.com/mappingObjects.pdf`, October 2000.

[2] The Apache Axis Project. axis. `http://ws.apache.org/axis/index.html`, December 2004. Version 1.2.

[3] Apache Software Foundation. The Byte Code Engineering Library (BCEL). `http://xml.apache.org/BCEL/bcel-5.1/docs/index.html`, 2003. Version 5.1.

[4] The Apache XML Beans Project. Apache XML Beans. `http://xml.apache.org/xmlbeans/index.html`, 19. June 2003. Version 2.0.

[5] The Apache XML Project. Xerces Java Parser. `http://xml.apache.org/xerces2-j/`, 2005. Version 2.7.1.

[6] Malcolm P. Atkinson and O. Peter Buneman. Types and Persistence in Database Programming Languages. volume 19 of *ACM Computing Surveys*, pages 105–190. ACM, June 1987.

[7] M.P Atkinson and R. Morrison. Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3):319–401, 1995.

[8] Kevin Beyer, Roberta J. Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy Lohman, Bob Lyle, Fatma Özcan, Hamid Pirahesh, Normen Seemann, Tuong Truong, Bert Van der Linden, Brian Vickery, and Chun Zhang. System RX: One Part Relational, One Part XML. In *ACM Sigmod Conference 2005*, Baltimore, Maryland, USA, 14-16. June 2005.

[9] Borland. *XML Application Developer's Guide, JBuilder*. Borland Software Corporation, Scotts Valley, CA, 1997,2001. Version 5.

[10] Beatrice Bouchou and Mirian Halfeld Ferrari Alves. Updates and Incremental Validation of XML Documents. In *Proceedings of the 9th International Conference on Data Base Programming Languages(DBPL)*, Potsdam, Germany, 6-8. September 2003.

[11] Ronald Bourret. XML Data Binding Resources. web document, `http://www.rpbourret.com/xml/XMLDataBinding.htm`, 28. July 2002.

[12] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The bigwig project. In *ACM Transactions on Internet Technology*, volume 2(2), pages 79–114. ACM, 2002.

[13] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular tree and regular hedge languages over unranked alphabets: Version 1. Technical Report HKUST-TCSC-2001-05, Hong Kong University of Science & Technology, April 3 2001. Theoretical Computer Science Center.

[14] Manfred Broy and Otto Spaniol, editors. *Informatik und Kommunikationstechnik*. Springer-Verlag, Berlin Heidelberg New York, 1999.

[15] Michael J. Carey and Miron Livny. Distributed concurrency control performance: A study of algorithms, distribution, and replication. In *VLDB ’88: Proceedings of the 14th International Conference on Very Large Data Bases*, pages 13–25, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[16] Josephine Cheng and Jane Xu. Xml and db2. In *Proceedings of the 16th IEEE International Conference on Data Engineering (ICDE)*, pages 569–576. IEEE, 2000.

[17] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending java for high-level web service construction. In *ACM Transactions on Programming Languages and Systems*, volume 25(6), pages 814–875. ACM, 2003.

[18] Ecma International. EcmaScript Language Specification. `http://www.ecma-international.org/`, December 1999. Edition 3.0.

[19] Ecma International. EcmaScript for XML Specification. `http://www.ecma-international.org/`, June 2004. Edition 1.0.

[20] K.P. Eswaran, J.N Gray, R.A. Lorie, and I.L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. Communications of the ACM 19, 1976.

[21] ExoLab Group. Castor. ExoLab Group, `http://castor.exolab.org/`, 2001.

[22] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and TillWestmann. Anatomy of a native XML base management system. In *The VLDB Journal*, volume 11, pages 292–314, 2002.

[23] Daniela Florescu, Andreas Grünhagen, and Donald Kossmann. XL: An XML Programming Language for Web Service Specification and Composition. In *Proceedings of International World Wide Web Conference (WWW 2002), May 7-11, Honolulu, Hawaii, USA*, pages 65–76. ACM, 2002. ISBN 1-880672-20-0.

[24] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, 2002.

[25] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150:1–??, 2001.

[26] Vladimir Gapayev and Benjamin C. Pierce. Regular object types. In *ECOOP 2003, Lecture Notes in Computer Science 2743*, pages 151–175. Springer-Verlag, 2003.

[27] David Gelernter. Multiple tuple spaces in linda. In *PARLE (2)*, pages 20–27, 1989.

[28] C. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[29] J. Gray. Notes on database operating systems: An advanced course. In R. Bayer, R. Graham, and G. Seegmuller, editors, *Lecture Notes in Computer Science*, number 60, pages 393–481. Springer-Verlag, 1978.

[30] UCLA Compilers Group. Java Tree Builder JTB. `http://compilers.cs.ucla.edu/jtb/`, 2004. Version 1.3.2.

[31] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael Burke, Vivek Sarkar, and Rajesh Bordawekar. XJ: Integration of XML Processing into Java. *IBM Research Report RC23007 (W0311-138)*, November 18, 2003.

[32] Hibernate. Hibernate. URL: `http://www.hibernate.org/4.html`, 2005.

[33] Haruo Hosoya and Benjamin C. Pierce. Xduce: A statically typed xml processing language. In *ACM Transactions on Internet Technology*, volume 3(2), pages 117–148. ACM, 2003.

[34] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for xml. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada*, volume 35(9) of *SIGPLAN Notices*, pages 11–22. ACM, September 18-21 2000. ISBN 1-58113-202-6.

[35] IBM Corporation. IBM DB2 XML Extender. URL: `http://www-3.ibm.com/software/data/db2/extenders/xmlext/`.

[36] Infonyte GmbH. Infonyte DB. URL: `http://www.infonyte.com`, 2003.

[37] java.net. Java Compiler Compiler (JavaCC) – The Java Parser Generator. `http://javacc.dev.java.net/`, 2004. Version 4.0.

[38] java.net. HyperJAXB. URL: `http://hyperjaxb.dev.java.net/`, 2005.

[39] JBoss, Inc. JBoss. http://www.jboss.com/, 2006.

[40] JDOM Project. JDOM FAQ. `http://www.jdom.org/docs/faq.html`.

[41] JPOX. Java Persistent Objects(JPOX). `http://www.jpox.org/index.jsp`.

[42] Martin Kempa. *Programmierung von XML-basierten Anwendungen unter Berücksichtigung der Sprachbeschreibung*. PhD thesis, Institut für Informationssysteme, Universität zu Lübeck, 2003. Aka Verlag, Berlin, (in German).

[43] Martin Kempa and Volker Linnemann. Type Checking in XOBE. In Gerhard Weikum, Harald Schöning, and Erhard Rahm, editors, *Proceedings of Datenbanksysteme für Business, Technologie und Web (BTW), 10. GI-Fachtagung,*, volume P-26 of *Lecture Notes in Informatics*, pages 227–246. Gesellschaft für Informatik, 26.-28. Februar 2003.

[44] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[45] Sang-Kyun Kim, Myungcheol Lee, and Kyu-Chul Lee. Validation of XML Document Updates Based on XML Schema in XML Databases. volume 2736 of *Lecture Notes in Computer Science (LNCS)*, pages 98–108, Heidelberg, 2003. Springer-Verlag.

[46] Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

[47] H.T. Kung and J.T Robinson. On Optimistic Methods for Concurrency Control. volume 2 of *ACM Transactions on Database Systems 6*, pages 213–226. ACM, 1982.

[48] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[49] Yuri Leontiev, M. Tamer Ozsu, and Duane Szafron. On Type Systems for Object-Oriented Database Programming Languages. volume 34 of *ACM Computing Surveys*, pages 409–449. ACM, December 2002.

[50] Mengchi Liu, Li Lu, and Guoren Wang. A Declarative XML-R Update Language. volume 2831 of *Lecture Notes in Computer Science (LNCS)*, pages 506–519, Heidelberg, 2003. Springer-Verlag.

[51] F. Matthes, G. Schröder, and J.W. Schmidt. Tycoon: A Scalable and Interoperable Persistent Environment. Fully Integrated Data Environments, ESPRIT Basic Research Series, pages 365–381, Heidelberg, 2000. Springer-Verlag.

[52] Erik Meijer, Wolfram Schulte, and Gavin Biermann. Programming with Circles, Triangles and Rectangles. `http://www.cl.cam.ac.uk/~gmb/Papers/vanilla-xml2003.html`, 2003.

[53] Microsoft. DCOM: Distributed Component Object Model Technologies. Microsoft, `http://www.microsoft.com/com/default.mspx`, 2005.

[54] Microsoft Corporation. .NET Framework Developer's Guide. web document, `http://msdn.microsoft.com/library/default.asp`, 2001.

[55] Ravi Murthy and Sandeepan Banerjee. XML Schemas in Oracle XML DB. In *Proceedings of the 29th VLDB Conference, Berlin, Germany*, pages 1009–1018, 2003.

[56] OASIS. Introduction to UDDI: Important Features and Functional Concepts. whitepaper: `http://www.uddi.org/whitepapers.html/`, October 2004.

[57] OASIS. UDDI Executive Overview: Enabling Service-Oriented Architecture. whitepaper: `http://www.uddi.org/whitepapers.html/`, October 2004.

[58] Object Data Management Group (ODMG). The Object Query Language (OQL). http://www.odmg.org/, 2000.

[59] Object Management Group (OMG). Common Object Request Broker Architecture (CORBA). OMG CORBA, `http://www.corba.org/`, 2005.

[60] Oracle Corporation. *Oracle9i, Application Developer's Guide – XML, Release 1 (9.0.1)*. Redwood City, CA 94065, USA, June 2001. Shelley Higgins, Part Number A88894-01.

[61] Oracle Corporation. Oracle XML DB. URL: `http://otn.oracle.com/tech/xml/xmldb/index.html`, 2003.

[62] T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Upper Saddle River, NJ, 1999.

[63] Yannis Papakonstantinou and Victor Vianu. Incremental Validation of XML Documents. volume 2572 of *Lecture Notes in Computer Science (LNCS)*, pages 47–63, Heidelberg, 2003. Springer-Verlag.

[64] Eduardo Pelegrí-Llopart and Larry Cable. JavaServer Pages Specification, Version 1.1. Java Software, Sun Microsystems, `http://java.sun.com/products/jsp/download.html`, 30. November 1999.

[65] Dominik Pietzsch. Entwicklung eines Prototypen für einen verteilten Datenpersistenz-Service im Kontext von **XOBE**$_{\mathrm{DBPL}}$. Master's thesis, Institut für Informationssysteme, Universität zu Lübeck, 2005. (in German).

[66] PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org/`, 2005.

[67] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, and R. Busse. XMark: A Benchmark for XML Data Management. In *International Conference on Very Large Data Bases(VLDB'02)*, pages 974–985, Hong Kong, August 2002.

[68] J.W. Schmidt and F. Matthes. The DBPL Project: Advances in Modular Database Programming. volume 19 of *Information Systems*, pages 121–140, 1994.

[69] Harald Schöning. Tamino - A DBMS designed for XML. In *Proceedings of the 17th International Conference on Data Engineering*, pages 149–154, Heidelberg, Germany, April 2-6 2001. IEEE Computer Society.

[70] Henrike Schuhart, Beda C. Hammerschmidt, and Volker Linnemann. Integrating Statically Typechecked XML Data Technologies Into Pure Java. In *15th International Conference On Software Engineering and Data Engineering (SEDE-2006)*, July 6-8 2006.

[71] Henrike Schuhart and Volker Linnemann. Implementing A Database Programming Language For XML Applications. In Nuno Guimaraes and Pedro Isaias, editors, *International Conference Applied Computing(IADIS)*, volume 1, pages 153–161. International Association for Development of the Information society, 21.-25. Februar 2005.

[72] Henrike Schuhart and Volker Linnemann. Updates for Persistent XML Objects. In Gottfried Vossen, Frank Leymann, Peter Lockemann, and Wolffried Stucky, editors, *Proceedings of Datenbanksysteme für Business, Technologie und Web (BTW), 11. GI-Fachtagung,*, volume P-65 of *Lecture Notes in Informatics*, pages 245–264. Gesellschaft für Informatik, 2.-4. March 2005.

[73] Henrike Schuhart, Dominik Pietzsch, and Volker Linnemann. Developing a Web Service for Distributed Persistent Objects in the Context of an XML Database Programming Language. In *International Conference On the Move(OTM)*, volume 3670 of *LNCS*, pages 613–630. Springer Verlag, 21.10.-4.11. 2005.

[74] Henrike Schuhart, Dominik Pietzsch, and Volker Linnemann. Framework of the XOBE Database Programming Language. In *International Conference Applied Computing(IADIS)*, 21.-25. Februar 2005.

[75] Mukul K. Sinha, P. D. Nandikar, and S. L. Mehndiratta. Timestamp based certification schemes for transactions in distributed database systems. *SIGMOD Rec.*, 14(4):402–411, 1985.

[76] SourceForge.net. XDoclet - Attribute Oriented Programming. `http://xdoclet.sourceforge.net/xdoclet/`, 2005. Version 3.2.

[77] Hong Su, Bintou Kane, Victor Chen, Cuong Diep, De Ming Guan, Jennifer Look, and Elke Rundensteiner. A Leightweight XML Constraint Check and Update Framework. volume 2784 of *Lecture Notes in Computer Science (LNCS)*, pages 39–50, Heidelberg, 2003. Springer-Verlag.

[78] Dan Suciu. The xml type checking problem. *SIGMOD Rec.*, 31(1):89–96, 2002.

[79] Sun Developer Network. Enterprise JavaBeans Technology. Sun Developer Network, `http://java.sun.com/products/ejb/`, 2005.

[80] Sun Developer Network. Java Data Objects (JDO) . Sun Developer Network, `http://java.sun.com/products/jdo/`, 2005.

[81] Sun Developer Network. Java Remote Method Invocation (Java RMI) . Sun Developer Network, `http://java.sun.com/products/jdk/rmi/`, 2005.

[82] Sun Developer Network. Jini Network Technology - Specifications. Sun Developer Network, `http://java.sun.com/software/jini/specs/`, 2005.

[83] Sun Microsystems, Inc. The Java Virtual Machine Specification. `http://java.sun.com/j2se/jvm/docs/index.html`, 1999.

[84] Sun Microsystems, Inc. Java 2 Platform, Standard Edition, v 1.4.2, Documentation. `http://java.sun.com/j2se/1.4.2/docs/index.html`, 2003.

[85] Sun Microsystems, Inc. Java 2 Platform, Standard Edition, v 1.5.0, Documentation. `http://java.sun.com/j2se/1.5.0/docs/index.html`, 2004.

[86] Sun Microsystems, Inc. Java Architecture for XML Binding (JAXB). `http://java.sun.com/xml/jaxb/`, July 2005.

[87] Gargi M. Sur, Joachim Hammer, and Jerome Simeon. UpdateX - An XQuery-Based Language for Processing Updates in XML. In *International Workshop on Programming Language Technologies for XML(PLAN-X 2004)*, pages 40–53, January 2004.

[88] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, Inc., Upper Saddle River, NJ, 2002.

[89] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating XML. In *ACM Sigmod Conference 2001*, pages 413–424. ACM, 2001.

[90] The Eclipse Foundation. Eclipse. `http://www.eclipse.org/`, 2006. Version 3.1.

[91] The hsqldb Development Group. HSQL Database Engine (HSQL). http://www.hsqldb.org/, February 2006.

[92] The Organization for Advancement of Structured Information Standards (OASIS). RELAX NG Specification. `http://relaxng.org/spec-20011203.html`, December 2001.

[93] W3Consortium. Hypertext Transfer Protocol 1.1(HTTP). W3C Draft Standard, `http://www.w3.org/Protocols/Specs.html`, June 1999.

[94] W3Consortium. Updates for XQuery. Working Draft, unpublished, October 2002.

[95] W3Consortium. SOAP Version 1.2 Part 0: Primer. W3C Recommendation, `http://www.w3.org/TR/soap-12-part0/`, 24. June 2003.

[96] W3Consortium. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation, `http://www.w3.org/TR/soap-12-part1/`, 24. June 2003.

[97] W3Consortium. Document Object Model (DOM) Level 3 Core Specification. Recommendation, `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/`, 07. April 2004.

[98] W3Consortium. Extensible Markup Language (XML) 1.0 (Third Edition). Recommendation, `http://www.w3.org/TR/2004/REC-xml-20040204/`, 04. February 2004.

[99] W3Consortium. XML Path Language (XPath), Version 2.0. W3C Working Draft, `http://www.w3.org/TR/xpath20`, 04. April 2004.

[100] W3Consortium. XML Schema Part 0: Primer Second Edition. Recommendation, `http://www.w3.org/TR/xmlschema-0/`, 28. October 2004.

[101] W3Consortium. XML Schema Part 1: Structures Second Edition. Recommendation, `http://www.w3.org/TR/xmlschema-1/`, 28. October 2004.

[102] W3Consortium. XML Schema Part 2: Datatypes Second Edition. Recommendation, `http://www.w3.org/TR/xmlschema-2/`, 28. October 2004.

[103] W3Consortium. Web Service Description Language (WSDL) Version 2.0 Part 0: Primer. Working Draft, `http://www.w3.org/TR/wsdl20-primer/`, 10. May 2005.

[104] W3Consortium. Web Service Description Language (WSDL) Version 2.0 Part 1: Core Language. Working Draft, `http://www.w3.org/TR/wsdl20/`, 10. May 2005.

[105] W3Consortium. XQuery 1.0: An XML Query Language. Working Draft, `http://www.w3.org/TR/xquery/`, 04. April 2005.

[106] W3Consortium. XQuery Update Facility Requirements. W3C Working Draft, `http://www.w3.org/TR/2005/WD-xquery-update-requirements-20050603/id-update-functionality`, 3. June 2005.

[107] C. Werner, C. Buschmann, and S. Fischer. WSDL-Driven SOAP Compression. *International Journal of Web Services Research*, 2(1), 2005.

[108] A. R. Williamson. *Java Servlets by Example*. Manning Publications Co., Greenwich, 1999.

[109] XML Database Initiative(XML:DB). XUpdate. `http://xmldb-org.sourceforge.net/xupdate`, 2004.

[110] About SAX. `http://sax.sourceforge.net`.

[111] Kun Yue, Zhengchuan Xu, Zhimao Guo, and Aoying Zhou. Constraint Preserving XML Updating. volume 2642 of *Lecture Notes in Computer Science (LNCS)*, pages 47–58, Heidelberg, 2003. Springer-Verlag.