Technische Universität Wien

Diplomarbeit

XML Databases for Augmented Reality

ausgeführt am
Institut für Softwaretechnik und Interaktive Systeme
der Technischen Universität Wien

unter Anleitung von
Ao.Univ.Prof. DI. Dr. Dieter Schmalstieg
und DI. Dr. Gerhard Reitmayr
als verantwortlich mitwirkenden Assistenten

durch
Werner Frieb
Treustraße 57
1200 Wien

Wien, 02. Oktober 2004

Contents

Abstract

The application of XML, an upcoming storage and communication format, requires an appropriate organisational system, if a certain quantity of document entities is exceeded. Generally, this can be best accomplished by employing a database system, which is targeted towards this technology, namely an XML Database.

This master thesis implements such a system for the Interactive Media Systems Group at Vienna University of Technology. Starting from the choice of three candidate databases, we have selected the one, which is best complying with the requirements of the Institute. According to the Client/Server architecture, we have installed and set up a central database server as a common storage system. The client applications accessing this database are part of *Studierstube*, an Augmented Reality system, which is the current research project of the group. In order to seamlessly integrate the XML database technology into *Studierstube* applications, we have designed and implemented a programming interface, which provides access to the database using XML query languages like XPath and XQuery. Finally, the implementation of the system was tested by means of a sample application, which processes XML documents stored in the database.

Kurzfassung

Der Einsatz von XML, dem standardisierten Dokument- und Kommunikationsformat, erfordert ein geeignetes Organisationssystem, wenn eine große Anzahl von Dokumenten verwaltet werden muss. Datenbanken eignen sich im allgemeinen am besten, um diese Aufgabe zu bewerkstelligen. Im Kontext von XML ist dies am effizientesten durch den Einsatz einer XML Datenbank zu erreichen, die direkte Unterstützung für diese Technologie bietet.

Diese Diplomarbeit implementiert ein derartiges Datenbanksystem für die Interactive Media System Group der Technischen Universität Wien. Ausgehend von einer Auswahl von drei Kandidat-Datenbanken, haben wir diejenige ausgesucht, die am besten den Anforderungen des Instituts entspricht. Gemäß Client/Server-Architektur, wurde von uns ein zentraler Datenbankserver installiert, der als gemeinsam genutzter Speicher dient. Die Client-Anwendungen, die auf diesen Server zugreifen, sind Teil des Augmented Reality Systems *Studierstube*, dem aktuellen Forschungsprojekt der Gruppe. Um die XML Datenbank-Technologie in *Studierstube* Anwendungen zu integrieren, haben wir eine Programmierschnittstelle entworfen und implementiert, die den Zugriff auf die Datenbank mit Hilfe der XML Abfragesprachen XPath und XQuery ermöglicht. Abschließend haben wir die Implementierung des Systems mit Hilfe einer Beispielanwendung getestet, die XML Dokumente verarbeitet, welche in der Datenbank gespeichert sind.

Acknowledgements

First of all, I would like to thank my parents, who made it possible for me to start studying computer sciences. Hopefully, now I can give something back and my mother has a reason to be a bit proud of her son.

A big "Thank You" also goes to my professor DI Dr. techn. Dieter Schmalstieg, who offered me this project and always provided me with resources and people in no time, thereby avoiding unnecessary delays in the progress of my work.

Furthermore, I would like to thank Dr. techn. Gerhard Reitmayr for supporting me with ideas and suggestions that helped me designing the software and to solve a lot of arising implementation problems. And, in particular, for his patience and understanding with my situation, when I was indignant and impatient.

Special thanks and two big kisses for Magistra Elisabeth Lahner-Altmann and Gudrun Wakolbinger, for proofreading my work and helping me to get rid of many errors.

And, last but not least, a "Thank You" to all of my friends, who supported me mentally during this work, especially in hard times. Special thanks go to Matthias Kramer, who encouraged me to proceed with my work in the most difficult phase.

Chapter 1

Introduction

XML was seen as a miracle drug for the software industry, when it was introduced and standardized by the W3C consortium in 1998. Rumors and wild speculations were spread. It was said that this technology would revolutionize and completely change computer technology, especially the Internet. Now, a few years later, these thoughts have turned out to be a hype. The revolution did not take place and COBOL based computer systems are still used by several banks. Instead, XML is going to be established as a standard in a slower, but quite steady way. The advantages of a common readable data format are increasingly outweighing the fears of management staff to fail at employing a new technology.

The idea of describing data with the help of a meta language is by no means new. XML is originating from SGML, the Standard Generalized Markup Language, which was conceived in the 1960s-1970s and standardized by the ISO organization in 1986. So, one can ask, why did it take so long for a good idea to be employed by a bigger community? SGML is, compared to XML, much more customizable and thus, more expensive to implement. Furthermore, at that time computer memory was much more expensive than it is today. It was seen as a waste of resources and money to use several bytes of computer memory in order to store a single byte of information, as it is common with XML.

The worldwide proliferation of PC systems and the resulting deterioration on prices for computer memory enabled the distribution of this technology. Strictly speaking, the additional bytes needed by XML to store data are not really wasted, but provide information about the structure of the data. This way, it is possible to write generic applications, which can process many kinds of different document formats. This advantage can not be achieved when using a proprietary binary data format. Nowadays, the ever growing pool of Open Source Software offers a vast amount of freely available applications and utilities, which support the processing of XML data.

As with many new technologies, universities and research labs are the first ones employing them, like the Interactive Media Systems Group at Vienna University of Technology did with XML. Meanwhile, a large number of different XML documents have been accumulated as single files stored on workstation computers of staff members. Thus, the file system is not an adequate storage medium any more, and another solution is needed to manage the data. Since databases have proven to be a good technology to store and query data in a scalable manner, the idea was born to employ an XML database as a replacement for the file system.

At the time of writing, XML Databases are a relatively new technology. The experiences in their robustness and usefulness are quite limited, compared to those with Relational Databases. Thus, it was not even clear, whether they are matured enough to be utilized for a project like that. Therefore, this thesis can also be seen as an experiment to check out the current state of XML Database technology. And, to anticipate it, it turned out to be an experiment with a successful outcome.

Reading the problem statement of this thesis, one may think that this is an easy task, which can be accomplished in no time - as the author admittedly was, when he started his work. Due to the youth of this technology several problems needed to be solved, which came across us mainly during the implementation phase. Nevertheless, the author does not regret having started this project, since he learned much about the upcoming XML technology. Hopefully, this work is a contribution to support the development of the *Studierstube* system and will serve as a base for further projects.

Chapter 2

Related work

2.1 XML

2.1.1 Introduction

XML is, like many other names in the information technology, an abbreviation and stands for the term E**x**tensible **M**arkup **L**anguage. It is a text based meta language for the definition of computer languages, which describe the syntax and structure of data. As a markup language, the syntax of XML is based on tags and attributes, and thus, looks quite similar to HTML. But, in contrast to the fixed language constructs of HTML, XML allows to define your own tags and attributes. In this way, it is possible to create your own XML language.

Originating from SGML, the **S**tandard **G**eneralized **M**arkup **L**anguage, XML was standardized in its first version by the World Wide Web Consortium in 1998. Since then it has achieved a great acceptance in the worldwide computer industry and is supported by many companies, including the big ones like Microsoft, Sun and IBM.

XML was originally intended as a universal data format in order to facilitate the exchange of information between applications. But, due to its popularity, it has entered many different fields of the information technology [6]. Until today, a number of languages have been defined, which are based on XML. Among them there are standardized languages like XHTML for World Wide Web applications, WML for WAP-phones, MATHML for mathematical expressions and XMLRPC for interprocess communication. In the course of the introduction of XML Databases, XML has even begun to be a partial replacement for Relational Databases.

Employing XML for storing data has many advantages over the usage of a proprietary binary data format. XML is platform independent, readable by humans, supports localization and is based on an international standard. Moreover, the common syntax of XML languages enables its users to share a wide range of tools, applications and related technologies like editors, parsers and XML processors. This way, projects utilizing XML can benefit from a big common pool of software applications and thus save time and money.

The following sections give an overview of the language features and related technologies like XSLT, XPath and XML Schema. Since a comprehensive description of XML is far out of the scope of this work, we will only discuss the basic features and refer to literature when needed.

2.1.2 The building blocks of XML

XML is a standardized meta language - or also meta document format - which is purely based on Unicode text. It is used to describe elements and structures of documents. The W3C XML standard defines a set of rules, which specifies the building blocks and syntax of a well-formed XML document. As the term "markup language" already reveals, the most characteristic entities of XML are marks, which are called tags.

A tag is a text, which is enclosed by angle brackets (<>). Tags are used to label and structure the content of an XML document. When defining a new document format, which is based on XML, one has to specify his own set of tags according to the type of data he wants to handle. The W3C standard only specifies the syntax rules for these tags, but does not say anything about their meaning. This is left to the creator of this new document format.

Additional to these tags the W3C standard specifies further entities, which can be used to form a document. The following sections give an overview of these entities, which an XML document can, or respectively, has to be composed of.

XML Declaration

Each well-formed XML document has to start with a declaration, which specifies the XML version and the character encoding used in the document, the so-called XML Declaration. It has to be the foremost item of a document. The following example specifies to use XML version 1.0 and character encoding UTF-8:

```
<?xml version="1.0" encoding="UTF-8"?>
```

This declaration tells an application, which processes the document, how to interpret the remainder of the XML document. Although the W3C standard defines it as obligatory, it is often omitted in practice.

Tags

Tags are used to define the logical structure of an XML document. A tag is a text, which is enclosed by angle brackets (<>). There are always pairs of tags: A start tag and an end tag, which enclose a part of a document and thus give it a certain meaning. The name of the end tag is the name of the start tag preceded by a "/". Such a pair of tags is also referred to as "XML element". The following example shows an XML element, which marks a text as a caption:

```
<caption> The building blocks of XML </caption>
```

Start and an end tags without any text between them are called "empty elements". Since it is quite usual to use empty elements, the W3C standard defines an abbreviated form for that. Instead of writing:

```
<caption></caption>
```

one can also write

```
<caption/>
```

Normally, an XML document will contain more than one XML element. There are two ways to align multiple elements. The first way is to align them serially, like in the following example, which represents a list of two captions.

```
<caption> The building blocks of XML </caption>
<caption> Using namespaces </caption>
```

The second way is by nesting them. This enables to structure a document hierarchically. The following example shows how previous captions can be defined as part of an article element.

```
<article>
    <caption> The building blocks of XML </caption>
    <caption> Using namespaces </caption>
</article>
```

There is one exception, where it is not allowed to align XML elements serially. This is the root tag of the document. The W3C standard specifies that a well-formed XML document must have only one topmost tag and all other tags have to be part of it.

Attributes

Beside the way to store data between a pair of tags, it is also possible to store data in form of attributes. An attribute is a named parameter, which can be attached to a start tag. Each start tag can own an arbitrary number of attributes. This is accomplished by adding the name of the attribute, followed by a "=" sign and the value of the attribute enclosed in apostrophes. The following example shows a list of two articles, where the title of the article and the name of the author are stored in attributes.

```
<article title="XML and Databases" author="Ronald Bourret">
    This paper gives a high-level overview of how
    to use XML with databases. It describes how...
</article>
<article title="eXist: An Open Source Native XML Database"
author="Wolfgang Meier">
    eXist is an Open Source effort to develop
    a native XML database system, which ...
</article>
```

Basically it does not matter, whether to use a pair of tags or an attribute to store a value, as long as it can be expressed as a single line string. The biggest difference between tags and attributes is, that attributes cannot be further structured like tags. Attributes are well suited to store atomic values, whereas tags are used when storing structured or big sized data.

CData

Some characters, like angle brackets, commas and apostrophes have a special meaning in XML. They cannot be used directly in regular text sections. Though there is a way to include them by using special expressions, sometimes a text must be kept in its original form. This is where the CData section comes into play. It allows to store arbitrary text in an XML document. The following example shows how you can use this language element to include special characters in an article.

```
<article>
    <![CDATA[Here can come any text,
    including special characters
    like <,> and &]
    ]>
</article>
```

A CData section is always enclosed by the strings "<![CDATA[" and "]]>".
Refer to [7] for a more comprehensive description of the CData element.

Processing instruction

The processing instruction (PI) is a convenient way to include additional information for applications, which process the XML document. A PI starts with "<?" followed by the name of the application. It can hold any number of attributes and is ended by the string "?>". The following example shows a PI, which defines two debugging attributes.

```
<?myapp debug="yes" output="print"?>
```

Comment

The comment tag is used to include annotations in an XML document, which are not part of the document content. Everything between the strings "<!–" and "–>" is treated as comment. The following example shows an annotation containing the name of the author.

```
<!-- Created by Tom Jones -->
```

2.1.3 Using namespaces

When working with XML, it happens, that one wants to use the same name for two different XML tags. For example, think of the tag "address", which could be used to hold a persons home address as well as the network address of a computer. Using the same name for different purposes, would lead to a name conflict, because we would not be able to distinguish between these tags. This name conflict can be resolved by the use of namespaces, which can be specified in any start tag. The following example shows how aforementioned "address" tags can be used side by side in the same XML document.

```
<?xml version="1.0" encoding="UTF-8"?>

<form xmlns:person="http://www.mydomain.com/xml/person"
xmlns:computer="http://www.mydomain.com/xml/computer">

    <person:address>New York, Broadway 173</person:address>
    <computer:address>192.168.0.10</computer:address>

</form>
```

The root tag "form" contains the declaration of two namespaces, "person" and "computer", which start with "xmlns" followed by ":" and a shortcut for the namespace, called namespace prefix. The string, which is assigned to these attributes, is used to identify the namespace and is usually a worldwide unique URI (Universal Resource Identifier). The namespace prefix is then utilized in the remaining document to refer to these declarations. It is written preceding to the tag, followed by ":" and the name of the tag. This way, the two "address" tags can be distinguished from each other. See [8] for a comprehensive description of XML namespaces.

2.1.4 Selecting data with XPath

XPath is a standardized addressing scheme for selecting a node or a set of nodes from XML data. The selected nodes can then be further processed by an application or by other languages like XSLT or XQuery. In fact, XPath is specified as part of XSLT, the W3C standard for XML transformation, which is described later in this chapter.

XPath allows to specify a location path, which defines the context of the nodes to find, and a condition the selected nodes must satisfy. These path expressions look very much like directory paths, when working with a computer file system. But XPath path expressions support several relationships, which are called axis. For example there is a child, parent, sibling or ancestor axis.

XPath also defines a library of standard functions for working with strings, numbers and Boolean expressions. These functions can be called in the conditional part of an XPath statement in order to specify further properties of the selected nodes.

Assume a library, which stores scientific articles in form of XML documents and that the following XML example represents a section of this library:

```
<library>
    <article title="XML and Databases" author="Ronald Bourret">
        This paper gives a high-level overview of
        how to use XML with databases. It describes how...
    </article>
    <article title="eXist: An Open Source Native XML Database"
             author="Wolfgang Meier">
        eXist is an Open Source effort to develop
        a native XML database system, which ...
    </article>
    ...
</library>
```

Assume further, that we want to select all articles written by Ronald Bourret from the library. Using XPath, the statement doing this, would look like the following line.

```
/library/article[@author="Ronald Bourret"]
```

The first part of the statement, the path expression "/library/article", selects all articles of the library. The subsequent conditional part, which is always enclosed in brackets, then reduces the selected set to those articles, which contain an author attribute with the value "Ronald Bourret".

For further information concerning XPath, see W3Schools XPath Tutorial [9] and [10] for a comprehensive set of practical examples explaining the various ways of selecting data with XPath.

2.1.5 Applying transformations using XSLT

XSLT, which stands for the term E**x**tensible **S**tylesheet **L**anguage **T**ransformations, is a programming language, especially developed for the transformation of XML data. An XSLT document, also called stylesheet, contains rules and statements, which define how to transform the data. Using this language an XML document can be transformed to any text format, including HTML and XML. Since XSLT is defined using XML, an XSLT program is also an XML document - it follows the syntactical rules of XML, too.

The basic principle of how an XSLT stylesheet works, is to select certain parts of an XML document and apply a transformation to them. It works a bit like a pattern matching mechanism, where the patterns are specified by XPath expressions (see previous section). The XPath expression selects the parts of the documents, which should be transformed in a certain way. The transformation is then defined by special XSLT statements, which access the data stored in the XML fragment and copy, transform or rearrange them in a new way. The transformation is applied for each XML fragment matching the XPath expression. In general, an XSLT stylesheet contains multiple pattern matching/transformation statements, which are executed successively.

Since an XSLT document is just plain text, a special application is needed to interpret the included statements and apply them to an XML document. Such an application is called XSLT processor. There are a number of freely available XSLT processors, which can be downloaded from the Internet, like Xalan from the Apache project, or Saxon, which was written by Michael Kay.

In the following example we will compile a list of all articles contained in the library document of the previous section. The resulting document should contain the article's title and the name of the author separated by a comma.

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
    <xsl:output method="text"/>

    <xsl:template match="/library/article">
        <xsl:value-of select = "@title"/>
        <xsl:text>, </xsl:text>
        <xsl:value-of select = "@author"/>
    </xsl:template>

</xsl:stylesheet>
```

Since an XSLT stylesheet is an XML document, the first line has to contain the XML declaration. The second line defines the root element "xsl:stylesheet" of the stylesheet and declares the namespace of the XSLT language. The subsequent "xsl:output" statement then sets the output method to plain text. The XSLT output statement supports the methods text, html and xml. The fourth line defines a template element, the pattern matching part of the stylesheet, which selects all articles from the source document by passing the XPath expression "/library/article" in the attribute "match". Within the template element we extract the title and author attributes from the current XML fragment and add a comma between them. The template element is called for each article of the source document and adds the selected attributes to the output list. The result of applying the stylesheet to the source document is then a list, separated by comma, like the following:

```
XML and Databases, Ronald Bourret
eXist - An Open Source Native XML Database, Wolfgang Meier ...
```

In order to learn more about the XSLT language, we recommend to read the XSLT tutorial by W3Schools [11].

2.1.6 Querying documents using XQuery

In addition to the languages XPath and XSLT, the W3C Consortium is currently working on a draft of a new XML query language, called XQuery. Though some details may still be a subject of change, the specification of this language already seems to have reached its final phase, at the time of writing. The current version has already been implemented in a number of projects, including the XML Databases *Tamino* and *eXist*.

XQuery is a combination of the features of XPath and XSLT and, to some extent, can be seen as a replacement for these two. XQuery syntax has some similarities to the syntax of SQL [7] and is therefore often referred to as the SQL for XML. One big advantage to XSLT is - due to its non-XML syntax - its improved readability for humans and thus it is much easier to learn.

The XML Query language has more than one syntax. In order to store XQuery statements in an XML document using a syntax that is compliant with the requirements of XML, the W3C Consortium has specified another syntax, called XQueryX. Since there is a bijective relation between XQuery and XQueryX, they can be exchanged for each other according to the current application.

Another advantage to the combination of XPath/XSLT is the improved support for datatypes in XQuery. The datatypes of XQuery are based on those of XML Schema (see next section), which provides a really extensive range. Moreover, it even allows to extend these, in order to define and customize own types. Hereby, the introduction of XQuery also eliminated the biggest deficiency of XML query languages, which was the poor support of datatypes.

Though there is a common sense that update queries are an important part of a query language, due to time reasons they have not been included in the first version of XQuery. So, utilizing a W3C compliant implementation of XQuery, we still have to use alternative ways to update documents, like XUpdate. Other vendors, like Software AG, the maker of *Tamino*, have forgone to be compliant with the standard and released an extended version of XQuery, which supports update operations for inserting, updating and deleting data. Since Software AG is a member of the W3C XML working group, we can expect that similar update operations will be introduced in one of the future versions of XQuery.

The following example demonstrates how to use XQuery in order to implement the task of the previous section. We presented an XSLT stylesheet, which extracts the title and the author from a list of articles stored in an XML document. In this example, we use the "For-Let-Where-Return" clause (spelled **FLOWER**) of XQuery to implement this.

```
FOR $a IN input()/library/article
WHERE $a/@author="Ronald Bourret"
RETURN <result>
    $a/@title, $a/@author
</result>
```

The FOR statement specifies the working set of the query, which in our example is the list of all articles contained in the XML document. Furthermore in this statement, the items of the set are bound to the variable "$a". The WHERE clause then reduces this set to the articles written by Ronald Bourret by testing the "author" attribute. Finally, the RETURN clause is called for each matching instance of the set and thus returns the list of articles. As one can see, compared to the XSLT code of the previous example, the solution using XQuery is rather short and much more readable.

If you want to learn more about this forthcoming standard, we recommend to read [12] and to have a look at [13] and [14].

2.1.7 Defining languages using XML Schema

The XML specification only gives a general description of the building blocks of a well-formed XML document. But it does not include a method to define rules for a specific, user defined XML language. Therefore, the W3C Consortium standardized a special language, which accomplishes this, namely XML Schema.

XML Schema, which is itself an XML language, provides ways to specify the valid elements and structures of an XML document. The fact, that XML Schema is formally expressed in XML, allows applications to process schemes in the same way as any XML document. This enables them to share a common code base and to save a lot of work and time. Actually, many of the XML parsers support the use of XML Schema, thus providing a way to automatically check whether a document follows certain rules, or not.

The specification of the sophisticated mechanisms of XML schema [15] is quite extensive and a comprehensive explanation of them fills whole books. Thus, we can only give an overview of the most important features and refer to further literature for interested readers. As usual in this chapter, we will also give a short example, which showcases the syntax of this language.

Since the basic building blocks of XML are tags and attributes, XML Schema provides means to specify, which of these entities are allowed in a certain context. In addition to defining datatypes for element content and attribute values, XML Schema includes mechanisms to define sequences of elements in a certain order, to restrict the number of allowed occurrences of an element, to define default values and to declare an element as optional, just to name the most important. Structures can be built by nesting elements and declaring arrays and lists of elements.

One of the biggest advantages, that comes along with the introduction of XML Schema, is the improved support for datatypes in XML. As mentioned in the section about XQuery, XML Schema provides an extensive range of datatypes we can choose from. Among them are numeric types like byte, int and float - string types like string, token, ID - date and time types - and types for storing binary data likes hexBinary or base64Binary. This enriches the capabilities of defining XML languages a lot and at the same time reduces the chances of error-prone type conversions, which would be necessary otherwise. Furthermore, by utilizing this type system, one is able to exactly specify the content of a valid XML document.

Another powerful feature of XML Schema is the possibility to declare user defined types and elements. In the manner of object oriented languages, XML Schema allows to derive new types and elements from existing ones. Depending on the kind of inheritance used, which can be restriction or extension, the derived types support a reduced or extended range of allowed elements and values. Several predefined types of XML Schema are defined by utilizing this inheritance scheme, thus establishing a hierarchy analog to class hierarchies of object oriented programming languages. The technique of creating new types by derivation is one of the key features of XML Schema, since it has already proven its usefulness in many software projects for years.

In the following example we define an XML Schema for the article element, which we used in the previous sections.

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http//www.w3.org/2001/XMLSchema">

<xs:element name="article">
    <xs:complexType>
        <xs:attribute name="title" type="xs:string"/>
        <xs:attribute name="author" type="xs:string"/>
    </xs:complexType>
</xs:element>

</xs:schema>
```

Above XML Schema starts with the XML declaration followed by the root element ”xs:schema”, which is always the topmost element. Since the XML Schema specification defines a namespace, we need to declare it here, too. Next comes the definition of the article element, which is of complex type, because it owns two attributes. The attributes for the title and the author of the article are then declared as child elements of the ”xs:complexType” tag using the ”xs:attribute” element. Finally, the datatype of both attributes is specified as simple string, because they should be able to store any text.

For further reading we recommend to study the tutorials at [16] and [17] and the W3C specification at [15].

2.1.8 Working with XML documents

Until now, we have seen how XML documents are structured and discussed related technologies to query, transform and describe XML data. This section deals with the question how one can work with XML data within programming languages.

In the course of the introduction of XML, a number of tools and libraries have been created to facilitate the processing of XML data within software applications, each of them embarking on a different strategy. All of these libraries have one thing in common: They serve as an interface between an application and XML data. Meanwhile, two of these interfaces have evolved into a standard, namely DOM and SAX.

DOM

DOM, the **D**ocument **O**bject **M**odel, standardized by the W3C Consortium [18], defines a set of classes and methods to access the elements of an XML document. When using DOM to process an XML document, a parser reads in the XML text and creates a tree of objects, which represents the elements contained in the document. The advantage of this method is, that the document is read in as a whole, thus enabling an application to randomly access and manipulate all of the elements. Whereas the disadvantage is given by high memory requirements and the delay resulting from the parsing process, which an application has to wait for until it can access the document content.

SAX

Alternatively to the DOM Model there exists another "de facto" standard for accessing XML data. SAX, the **S**imple **A**PI for **X**ML [19], uses an event oriented strategy to pass the content of an XML document to an application. When the parser reads in the document, it generates events for each XML entity found in the document. An application can then process these events, by deriving a new class from a special interface class and overwriting its virtual callback methods. This way, an application can selectively process certain document entities and leave out the remaining parts. Compared to DOM, this method needs much less memory, but, as disadvantage, it requires to store and manage context information if needed.

Other techniques

Additionally to DOM and SAX, there are further techniques to deal with XML documents. These will not be discussed here. For the sake of completeness, we have compiled the following list. See the literature references in order to read more about alternative ways to DOM and SAX.

- XML Data Binding - Bindings of XML data to programming language structures [23]
- JAXB - Java Architecture for XML Binding [25]
- JAXP - The Java API for XML Processing [20]
- JDOM - The Java Document Object Model [21]
- DOM4J - Document Object Model for Java [22]

2.2 XML Databases

2.2.1 Introduction

The conventional way to manage XML documents is to store them as files in a directory structure on a computer drive. This approach follows a natural way of keeping data as documents, since XML is a document format and documents are usually kept in files. So, for instance, applications that need to manage structured configuration data, can do this conveniently by utilizing XML as storage medium in form of files.

Due to the popularity of XML, the amount of data that is organized using XML has enormously increased in the past few years. The more data a system has to manage, the harder it is to keep it consistent. Handling a large number of XML documents in form of single files and using the directory structure to organize them, can become very impractical, soon. And, if the number goes into the thousands, this can become an impossible mission and an appropriate storage system, usually a database, is needed.

Apart from managing large amounts of data, there are several more reasons that speak for employing a database in a project. So, for instance, by utilizing indexes, a database usually provides a better performance than a file system. Furthermore, a database is capable of managing concurrent access of multiple users, offers data integrity functions, transactions, triggers and a lot more. In regard to XML, such a database system is called XML Database.

The term "XML Database" subsumes all available database systems, which are capable of storing and retrieving XML documents. This can be accomplished either by utilizing conventional Relational Databases or by databases, which are especially designed for the storage of XML documents. Ronald Bourret [27] distinguishes between the following three types of XML Databases.

- Relational databases (supporting XML)
- XML-enabled databases
- Native XML databases

The key to choose the right one for a particular application lies in the structure of the data, which needs to be processed. Thus, we have to have a look on the different forms of XML data first. The subsequent sections then discuss the different properties and features of these database types.

2.2.2 Data-centric vs. Document-centric XML data

According to their structure, XML documents can be roughly divided into two categories: Data-centric XML documents and document-centric XML documents [27]. Though it is not always clear to which category a document should be added to, it makes sense to distinguish between these two categories. It helps to choose an appropriate storage technology.

Data-centric documents

Data-centric documents are characterized by a fairly regular structure with fine-grained data (small independent units of data at element and attribute level) and with little or no mixed content (XML tags mixed with text). These are documents, which have a structure similar to that of a relational database table and could possibly originate from it. The following XML document showcases an example for a data-centric document, which stores patient data of a hospital.

```
<?xml version="1.0" encoding="UTF-8"?>

<patients>
    <patient>
        <firstname>Otmar</firstname>
        <lastname>Bauer</lastname>
        <assuranceid>1234-56678</assuranceid>
        <street>123 Main street</street>
        <city>Vienna</city>
        <country>Austria</country>
    </patient>
    <patient>
        <firstname>Fred</firstname>
        <lastname>Mayer</lastname>
        <assuranceid>432-83474</assuranceid>
        <street>124 Haydn street</street>
        <city>Eisenstadt</city>
        <country>Austria</country>
    </patient>
    ...
</patients>
```

Document-centric documents

The second category, document-centric documents, are characterized by a less regular or irregular structure with larger grained data and a lots of mixed content. These are documents, which are usually designed for human consumption and typically do not fit well into a relational database scheme. For example, to stay within the hospital domain, it could be a diagnostic description of a doctor like the following.

```
<?xml version="1.0" encoding="UTF-8"?>

<diagnosis>
    The patient <firstname>Otmar</firstname>
    <lastname>Bauer</lastname> was admitted to the
    <ward>surgery</ward>,because of a
    <admission_diagnose>broken arm</admission_diagnose>.
</diagnosis>
```

Though both categories of XML documents, data-centric and document-centric, follow the XML specification, there is a big difference between them regarding the needed storage technology. The following sections discuss the different techniques with regard to the structure of the documents they can handle.

2.2.3 Relational Databases

This type of XML Database uses a table-based mapping to store particular XML documents in a Relational Database. It models XML documents as a single table or as a set of tables. The model is directly derived from the schema of the document, complex elements are mapped to tables and simple elements and attributes are mapped to columns. Since there is a direct structural relationship between the database tables and the XML document, it cannot be used for any XML documents that do not match the format.

The table-based mapping is most suitable to handle data-centric documents, when transferring data between two relational databases.

2.2.4 XML-enabled databases

XML-enabled databases use an object-relational mapping. Therefore, XML documents are modeled as a tree of objects, which are specific to the data in the documents. The model is then mapped to Relational Databases using object-relational mapping techniques or SQL 3 object views. What means, that classes are mapped to tables, scalar properties are mapped to columns and object-valued properties are mapped to primary key/foreign key values.

This technique can handle data-centric and document-centric XML documents, though it does not support mixed content well.

2.2.5 Native XML Databases

Native XML Databases are designed especially to store XML documents. Their internal logical model is based on XML, but is not a question of the physical storage model. The physical backend of this database type can still be a Relational Database, like the first version of *eXist* showed.

One important characteristic of Native XML Databases is, that they preserve important properties of XML documents (like element order) and XML specific elements (like processing instructions and comments). Which means that, what one puts in, will also come out. A further characteristic is the support for XML query languages like XPath or XQuery. Naturally, Native XML Databases should also offer common database features like transactions and multi user access.

Native XML Databases are best used, when the data is semi-structured. The structure varies and a mapping to a Relational Database is difficult to achieve. For instance, object oriented tree structures, which make use of derivation, would need an additional table for each derived class. Native XML Databases can handle both categories of XML documents, data-centric as well as document-centric.

Finally, depending on the database architecture they use, Native XML Databases can be further divided into text-based and model-based databases. Text-based databases store XML documents as text, thus storing an identical copy of the data, which is put in. This gives text-based XML Databases a tremendous speed advantage, when retrieving entire documents or document fragments, compared to the solutions mentioned. Though they also can encounter performance problems, when retrieving data, which is different from the predefined hierarchy.

Model-based databases build an internal object model of the XML document and store this model. Some databases store the model in Relational Databases, others in object-oriented databases or they store the model as persistent DOM, like the recent version of *eXist*. They have the same performance characteristics like text-based databases. Whether they are faster or slower than text-based systems is not clear. Future applications will show which approach is the better one.

2.3 Database interfaces

2.3.1 Introduction

In order to benefit from the long running experience of approved database interfaces, this section investigates the design of several programming interfaces for databases. All of these interfaces are implemented using an object oriented language, preferably C++, because our implementation language is C++, too. Thereby, the focus of our attention is directed mainly to class collaboration and derivation. We will analyze how these class libraries work and try to pick out the approaches, which fit our needs best.

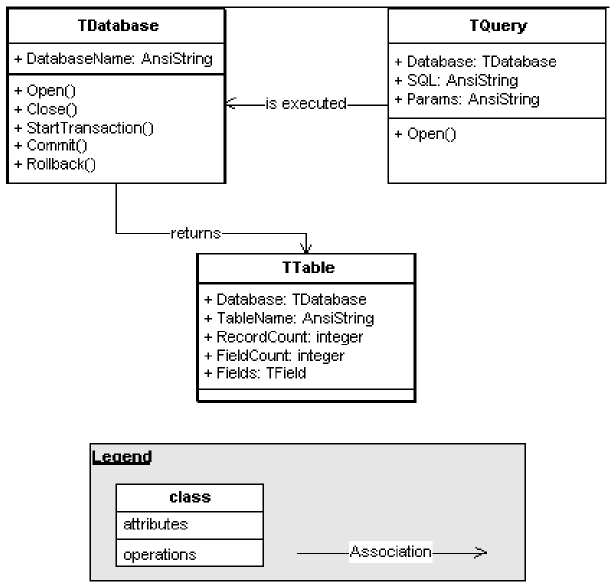


Figure 2.1: Collaboration diagram of VCL database classes

Due to the fact that there have not been many XML Database API's released until now - only two we know of - we will also have a look at two SQL Database API's. Basically, they follow the same approach as their XML counterpart. They provide classes to query databases and have to handle the resulting data in some way. In principle, they only differ in the syntax of query scripts and in the type of query results. Therefore, analyzing the design of this interfaces makes sense, too.

2.3.2 Borland VCL

Borland VCL [29], the **V**isual **C**omponent **L**ibrary, is a class library originating from *Borland Delphi* (Object Pascal), which has been made available for C++. This library includes a set of classes, which are designed for database access, mainly for Relational Databases, which support SQL. The collaboration diagram in figure 2.1 on page 27 shows the most important classes of this library, which are involved in a database query.

TDatabase represents a physical database. It supports operations to connect (*Open()*) and disconnect (*Close()*) to a particular database and methods for transactions (*StartTransaction()*, *Commit()*, *Rollback()*). A database query is represented by the class *TQuery*, which holds a SQL script and additional connection parameters. When calling the *Open()* method of *TQuery*, the SQL query is executed against a database, which is passed in the member variable "Database". In consequence, *TDatabase* runs the query and stores the result in an instance of *TTable*. The query result can then

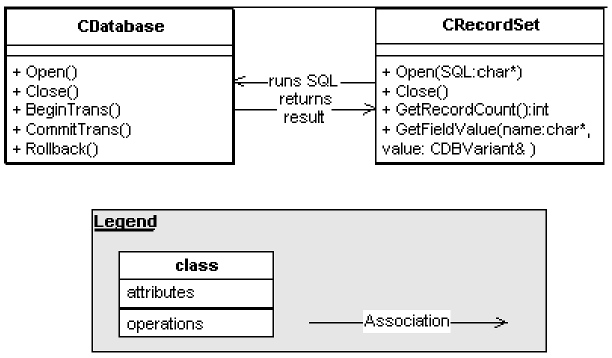


Figure 2.2: Collaboration diagram of MFC ODBC database classes

be accessed in the "Fields" member variable of *TTable*.

Though there are more classes, which can be utilized for other tasks like managing sessions or running batch jobs, the previous paragraph describes the basic scenario for querying databases in VCL. This quite simple, but efficient and easy to use approach is worth to be considered in our design.

2.3.3 Microsoft MFC

MFC, the **M**icrosoft **F**oundation **C**lasses [28], is a C++ class library for developing MS-Windows based applications. It supports database operations by providing different sets of classes for different types of databases. Figure 2.2 on page 28 depicts a collaboration diagram of the classes for ODBC database access.

The approach used by the classes in figure 2.2 is quite similar to that of Borland's VCL, though with different class names. There is a *CDatabase* class, which supports operations for connection and transaction and a class *CRecordSet* for storing the result set of database queries. The main difference to the VCL approach is, that there is no separate class for queries. SQL queries are executed by calling the Open() method of *CRecordSet*, which thereby covers both tasks, querying and storing result sets. As long as there is only a single query parameter to keep track of - here in form of an SQL string - this approach seems to be sufficient. Whereas the VCL solution seems to be better, when a query supports several parameters, because in this way they can be reused more easily in another context.

2.3.4 XML:DB API

The XML:DB API [30] is the standard programming interface for XML Databases. It is targeted towards all object oriented programming languages, but has only been implemented for Java so far. The class library is intended to be vendor neutral and

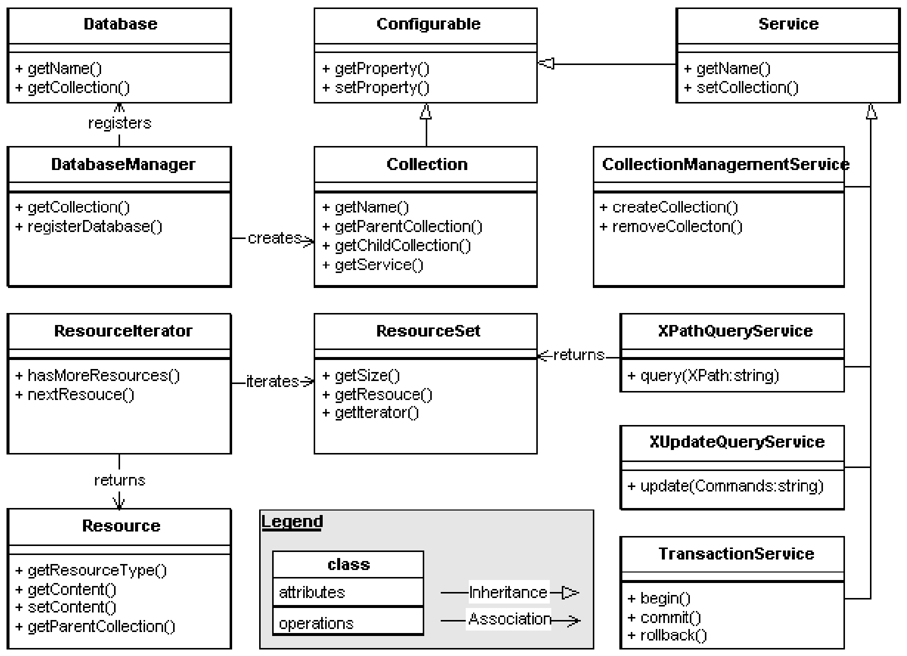


Figure 2.3: Collaboration diagram of XML:DB API database classes

thus has been designed to provide as much flexibility as possible. For this reason, it is also the most complex interface of those examined here.

We will only discuss the most important aspects of this class library here. For the interested readers we recommend to read the introduction written by Kimbro Staken [31] and to have a look at the API specification [30].

Figure 2.3 on page 29 shows a collaboration diagram of the main classes of this API. In the middle of this figure one finds the *Collection* class, which is the counterpart to the classes *TDatabase* and *CDatabase* of the previous sections. It represents an XML Database collection, which generally contains a set of XML documents and further sub collections. A collection is the data source of queries and the destination of updates, which are represented by the classes *XPathQueryService* and *XUpdateQuery-Service*. The resulting XML documents or document fragments of a query are stored in an instance of *ResourceSet*, whose items can be iterated by *ResourceIterator*. These items are stored in objects of the *XMLResource* and *BinaryResource* type (not depicted here because of the lack of space), which are derived from the abstract base class *Resource*.

A bit misleading is the name of the class *Database*, which is actually not an interface for an XML Database as one could suppose, but a vendor specific implementation of a database driver. These implementations are managed by the class *DatabaseManager*, which is also the starting point for an application using the API. *DatabaseManager* provides access to particular XML Databases, by providing instances of the class *Collection*.

In order to be modular and extensible, non-basic features are implemented as services by deriving new classes from the abstract base class *Service*. So, for instance, the collection management features (*CollectionManagementService*) and transactions (*TransactionService*) have been implemented this way.

In order to get a version of this API, which complies with the lowest level of the standard (core level 0), one has to support the basic framework and thus, has to implement most of the classes in figure 2.3. At the time of writing, even vendors like Software AG, the maker of the *Tamino* XML Database, have not completely implemented this standard API for their database product. Hopefully, future versions will.

2.3.5 XinCJ - Xindice C++ API

The *XinCJ* class library [32] is an attempt to implement a programming interface for the XML Database *Xindice*. It is a C++ Java Native Interface wrapper for *Xindice* implementing a subset of the XML:DB interface and thus allows to access the *Xindice* XML Database from C++. It tries to match the original Java class structure as closely as possible.

Unfortunately, the development of this class library has been suspended in an early state and currently it does not seem that it will ever be finished. Nevertheless we had a look at this implementation. We were once more disappointed to find it poorly documented and a bit quirky. Maybe the increasing demand for XML applications will motivate someone to continue this work at a later time.

Chapter 3

Problem Description

3.1 Introduction

This chapter specifies the conceptual formulation of this project, or in other words, describes the task, which was consigned to the author of this thesis. It defines the goal of the project by describing the basic features of the applications and the content of the documents, which should be included in the final result. (We have also included some of the details, that have only been verbally discussed later in the project.)

Studierstube is a scientific platform for contriving and developing Augmented Reality applications. Naturally, in the context of research, a large number of documents and a lot of resulting data has to be managed. Up to now, the collaborators of this platform solely utilized the file system of their workstations to persistently store and organize the accruing data. As one can imagine, at a certain point, this form of organization will come across its limits. When a certain amount of data is exceeded or the data needs to be interconnected and easily shared between the collaborators, an appropriate storage system, i.e. a database, is needed.

The research data, which needs to be managed by the database system, varies strongly in its structure, from flat relational tables to deeply structured recursive trees. Since XML is best suitable to handle a wide spectrum of document formats, the decision was made to store future data as XML documents. In order to share these documents, a central XML Database should then serve as a common storage system.

Essentially, the task is now, to select an appropriate database product, to install the server application on a computer of the Institute, as well as to enable the connection of *Studierstube* applications to the server.

3.2 XML Database server

Since the implementation of a fully featured database server is out of the scope of this work, we need to employ one of the products, which are available on the market. Currently, the number of alternatives is still a bit limited. Thus, the choice of candidates can be narrowed down to the following three databases:

- *Tamino* [33], a commercial product of Software AG
- *Xindice* [34], an Open Source project of the Apache group, and
- *eXist* [35], another Open Source database, founded by Wolfgang Meier

At the time of writing, XML Databases are a relatively new technology. They cannot be expected to be fully matured yet. Since that, we need more information about them to facilitate our decision for a particular candidate.

Thus, the first step of this work will be to install these databases on a computer in order to gain first experiences and to check out the features they support. Thereby the focus of the investigation should be directed to programming interfaces and to administration applications, which are available for each candidate. Especially, a tool is needed to maintain the data, preferably one providing a graphical user interface.

Furthermore, the databases should support an XML query language like XPath or XQuery, and an update language like XUpdate. Advanced database features like multi user access, transactions and backup strategies are not that important, but cannot be completely neglected, since this work is part of a long running project. In the case of a successful outcome, it will be the base for several other works.

Results of this work step:

- Test installation of the database candidates on a test server
- A report, which is describing and comparing their features, with focus on programming interfaces

3.3 Client interface

In order to connect *Studierstube* applications to the database server, a C++ library should be developed on top of the database API, mainly for reading access. (In the following text this library is also referred as *Studierstube XML Database API.*) This library should be designed in a way, which optimizes the usability for *Studierstube* applications. A database query should then support the following features:

- Specification of the source database:
  - Network address of the server and database
  - Specification of the wanted data by the query language XPath
- Query results in the following forms:
  - XML document as text in the computer memory
  - XML-DOM as SoXML model (see below)
- Specification of an XSLT Transformation, which is applied to the query result on the server side
- Specification of an XSLT Transformation, which is applied to the query result on the client side

For the processing of XSLT a library, like Xalan, should be selected and linked to the library. Furthermore, query parameters, like XSLT style sheets and XPath statements, should be easily reusable within an application. A suitable object oriented container class has to be designed to achieve this. The client API should also be designed in a modular manner, that is possible to utilize only parts of the API. In particular, the transformation features should be usable also by Non-*Studierstube* applications.

Results of this work step:

- A specification of the Client API

3.4 Data representation in form of SoXML

Studierstube is built on top of *Coin2* (formerly *Open Inventor*), a toolkit for the interaction and rendering of 3D graphics. *Coin* uses a scene graph model, where graphical objects are represented as nodes of a tree. In order to integrate XML data into this model, a connection between XML and *Coin* is needed. The hierarchical tree structure of the scene graph can be used as a base for an XML DOM model. This way, *Studierstube* applications will be able to access XML data in the same way as graphical objects.

In order to implement this feature, a new class, which is derived from the *Coin* base class *SoGroup*, can be defined as the following structure:

```
SoXML {
    SoSFString name
    SoMFString attributes
    SoMFString values
    SoSFString content
}
```

SoSFString and *SoMFString* are *Coin* classes for the representation of single strings (S) and string arrays (M).

The tag name of an XML element is hold by ”name”, its text content is stored in the member variable ”content”. Attribute names are stored in ”attributes”, attribute values in ”values”. Nested elements are put into the child list of *SoGroup*.

Example. Assume the following XML fragment:

```
<el1 attr1="A">
    <el2 attr2="1.1" el3="B">
        <el3 attr4="true" />
        <el4>Hello</el4>
    </el2>
</el1>
```

This fragment can be converted by an XSLT stylesheet into following *SoXML* representation:

```
XMLElement {

    name "el1"
    attributes ["attr1"]
    values ["A"]

    XMLElement {
        name "el2"
        attributes ["attr2","attr3"]
        values ["1.1","B"]

            XMLElement {
                name "el3"
                attributes ["attr4"]
                values ["true"]
            }

            XMLElement {
                name "el4"
                content "hello"
            }
    }
}
```

In case the *SoXML* model is represented as text in main memory, it can be easily converted to a corresponding scene graph.

Results of this work step:

- Specification of *SoXML*

Eventually, namespaces should be also regarded in the design. Therefore, the definition of *XMLElement* must be extended as in the following example:

```
SoXML {
    SoSFString name
    SoSFString namespace
    SoMFString attributes
    SoMFString values
    SoMFString attributeNS
    SoSFString content
}
```

Alternatively, the namespace could be stored as a prefix in the string variables "name" and "attribute", like "ns:name". But probably, this might not comfortable enough for the users of the library, because in this way the string needs to be parsed first.

3.5 Specification of a test application

An application should be designed to test the implemented components with real and large data. This application has to use *SoXML* for data representation in order to demonstrate the connection to *Coin*.

Results of this work step:

- Specification of a test application

3.6 Implementation

The last step of this work is to implement and test all of previously described features.

Results of this work step:

- Implementation of the *Studierstube* XML Database API
- Implementation of the *SoXML* component
- Implementation of the test/sample application

- Documentation of the components and applications in form of manuals and web pages

Chapter 4

Choosing a database product

4.1 Introduction

The first step of this work was to evaluate and select an appropriate XML Database for this project. Since the data to be processed concerns differently structured XML documents, the choice was quickly narrowed down to one of the following three Native XML Databases: An older version of *Tamino* [33], a commercial product developed by Software AG. *Xindice* [34], an Open Source XML database by the Apache Group and *eXist* [35], an Open Source XML database founded by Wolfgang Meier at the Technical University Darmstadt, Germany. There are a plenty of other commercial alternatives, which did not come into consideration, just because they were not available to us and there are further Open Source solutions, which simply did not fit our requirements. We were looking for a Native XML Database system, supporting XPath [9] and XUpdate, which comes with a set of usable and working administration applications, preferably with support for sever side XSLT processing and a C++ API for easy integration into our system.

This chapter tries to evaluate and compare aforementioned systems, showing the strengths and weaknesses of each in order to have a basis to select one for our task. Additionally, appendix A of this work contains several charts, which list and compare the features we were interested in.

4.2 Tamino

Tamino [33] is a commercial Native Database system, developed by Software AG. First of all it has to be mentioned that the version used for this evaluation (v3.1.2) is not the latest version available, which is version v4.1.4 at the time of writing. Unfortunately we did not have a possibility to upgrade to the latest version and so we had to stick to the older one. Therefore everything said here does not apply to the latest version, which has undergone major improvements. (Most notably the added support for XQuery [12] and the XML:DB API [31]).

4.2.1 Installation

Installing *Tamino* was not as easy as one would expect. The first attempt to run the installation program failed because of *Tamino's* licensing system: The version we have does not install on Windows XP, because it is restricted to be set up on Windows 9x or Windows 2000. Using Windows 2000 instead of XP helped.

At first the installation wizard asks to setup an HTTP-server, preferably *Apache*, which is included on the CD. This server is needed, because some of the supplied applications, in particular the main administration application *Tamino Manager*, are implemented as web applications. Done this and starting the installation program again, the next thing we had to do was to update Internet Explorer and the Microsoft Java VM. Using Internet Explorer as client-side browser application, one needs to have a more recent version of Java VM than the one that comes with Windows 2000.

Already a bit exasperated by the length of the installation procedure, we had to discover that Microsoft is not supporting the Java VM anymore. After an odyssey to find a way to get this update, we succeeded by running the Windows update function and installed Java VM 3810 and in the end *Tamino*.

4.2.2 Applications

The *Tamino* database system comes with a number of applications. The following sections will try to give a short description of each program.

Tamino Manager, implemented as web-application, lets tune and configure the database server from a remote location, manage databases and run the backup/restore function.

Tamino Schema Editor, a Java application, is needed to edit *Tamino's* own XML Schema, an adopted version of the W3C schema. On the one hand it enhances this schema by adding logical and physical properties, which describe storage and indexing information. On the other hand it lacks of support for the more advanced features of XML Schema, like derivation, recursion and complex basic datatypes. These limitations seriously restrict the area of application of this language, especially when one needs to work with recursive data structures.

Tamino Explorer, another Java application, is used to browse the database through an explorer-like user interface. Database contents can be viewed and edited quite comfortably. The Query windows allows to commit XPath queries to the database server and to view and inspect the result set.

The last application to mention is *Tamino Interactive Interface*, a simple Web-form for querying and updating the database using a Web browser.

All in all, the impression of the delivered applications is quite good, with pleasing user interfaces and useful features. None of them ever crashed or had troubles connecting to the database during the test.

4.2.3 Database features

Since this is an older version of *Tamino* it lacks of support for a query language capable of doing updates. As a consequence, partial updates of documents in the database are not available. It is only possible to update documents as a whole. More mature are the features for querying the database. W3C's XPath query language has been implemented in an extended form and - in order to cause confusion - is called X-Query. (Note the dash between the X and the Q.)

On the one hand X-Query enhances XPath with additional functions and operators, and offers the use of variables as well, but on the other hand there are some W3C functions left out, which means that it is not fully W3C compliant. *Tamino* also allows to define a schema for data structures. Using W3C's schema definition language as basis, *Tamino's* schema language covers - in addition to the standardized features - the definition of database indexes and the use of data types for elements and attributes. The feature to define data types is unique among the tested candidates and is especially useful when working with a lot of data-centric information.

4.2.4 API's

As usual within the XML domain, most of the programming language API's are written in Java. Although there is a JAXP and JDOM implementation, this version of *Tamino* does not implement the de facto standard API for Java, the XML:DB API. Support for C++ is given in form of an Active-X control which communicates to the server over HTTP. Server side XSLT processing can be implemented using *XTension*, the interface for enhancing server functionality.

4.2.5 Documentation

If there is one thing outstanding in *Tamino*, then it is the documentation and help system. There are about 3000 pages describing *Tamino's* features and applications, including pictures, graphics and many practical examples and about 1500 additional pages documenting SQL and ODBC features. Although the HTML search function is not as comfortable as the one within the native Windows help system, it is still useful and helps to cut through this vast amount of information.

4.3 Xindice

Xindice [34] formerly known as *dbXML*, is part of the *Apache* XML project. It is an Open Source XML database developed by volunteers under the *Apache* license. In spite of the fact, that the first official version (version 1.0) of *Xindice* was released more than a year ago, it is currently undergoing major changes. As a result of this modifications there are actual two different versions of *Xindice* now, each of it providing different features. Although the newer version (v1.1) is still in beta state, it is the one discussed here, because it represents the future of *Xindice*. Differences to the older version are mentioned when it's relevant.

4.3.1 Installation

Due to the very short installation documentation that comes with *Xindice*, it took us several hours to find out how to get a working system with a running server, get the command line tools working and the *XMLdbGUI* graphical client application connecting properly. In order to share my experience, I'll try to give a short installation guide including some tips on installing *Xindice*.

On the download page [34] you can see that *Xindice* 1.1 is delivered in form of three zipped files. The two files named "xml-xindice-xxx-war.zip" and "xml-xindice-xxx-jar.zip" contain the binary distribution and "xml-xindice-xxx-src.zip" contains the source distribution. The source distribution is needed when you want to contribute to the project or when you want to use the included *Jetty* servlet engine. We use the binary distribution instead.

Because support for the standalone server version has been dropped in the current release, the database server of *Xindice* is delivered as a servlet and hence needs a servlet engine like *Apache Tomcat* to be installed first. Installation of *Tomcat* is straightforward, just download the setup program and follow the instructions. Next copy the *.war-file, which is included in "xml-xindice-xxx-war.zip", into the "webapps" folder of *Tomcat* and restart the *Tomcat* application. It will then unpack the *.war-file and install the servlet. The *.war-file can be safely deleted then. Note: When using Tomcat 4.1 as servlet engine you have to rename the *.war-file to "xindice.war" before installation, otherwise you will not be able to connect to the database server!

Furthermore, it is recommended to make more memory available to *Xindice* than

64MB, which is the default configuration. Add the parameter "-Xmx999m" to the startup line of the *Tomcat* server, where 999 stands for the maximum size of memory (in megabytes) used for *Tomcat*. You should then check the installation by browsing to "http://localhost:8080/xindice". If the installation was successful, a web page for browsing the database, called "The ugly debug tool", will show up.

Next, we install the command-line client tools, which are included in "xml-xindice-xxx-jar.zip". Unpack the contents of the zip-file to "C:\Xindice". If you use this directory you will not have to edit the batch file "xindice.bat", which starts the command line programs. *Hint*: If the batch file is not working properly, this is most likely because of missing quotation marks around environment variables. Attention! This does not apply to the environment variable "%CL%". Check the installation of the command line tools by starting the following line from a command prompt:

```
Xindice lc -c xmldb:xindice://localhost:8080/db.
```

This should list the current database content, which of course only consists of the system collections at the very beginning.

Next, we install *XMLdbGUI*, a Java application with a graphical user interface used to browse and edit database content. Since the full version of *XMLdbGUI* currently only supports *Xindice* 1.0 and the included Java libraries are already out of date, it is recommended to download the light version and copy the required files as described in the documentation. Furthermore you will need to edit the configuration file "conf.xml", which is also explained in the documentation that comes with *XMLdbGUI*. *Hint*: Use the URI "xmldb:xindice://localhost:8080", if you do not get a connection to the database.

4.3.2 Applications

On the Internet one finds a number of client applications written for *Xindice*, although most of them did not leave beta states until today.

Most outstanding are *Xindice Webadmin*, a simple web interface for creating collections, querying and uploading data and *XMLdbGUI*, a Java application with similar features.

Unfortunately, due to the changes made to the new database version, applications like *Xindice Browser*, which access the database directly, do not work with *Xindice* anymore.

All in all, the user interface provided by the applications does not cover all features of *Xindice*. In order to be able to adjust and fine tune database specific settings, one still has to edit configuration files by hand. The availability of *XMLdbGUI* as a comfortable user interface seems to be the crucial factor for the usability of *Xindice*.

4.3.3 Database features

Xindice is currently supporting XPath, XUpdate and additionally SiXDML, a simplified non-standard query language. The XPath implementation is taken from the *Apache Xalan* project, which is said to be fully W3C compliant. But this W3C specification does not cover XPath queries across collections and recursive queries of sub collections, which are currently not supported by *Xindice*. Database indexes are important to speed up queries. Therefore *Xindice* offers manual indexes, which means, that for each element one wants to be included in the index, one has to add an entry in the *Xindice* configuration file.

The note in the *Xindice* FAQ [34], that the database does not process big sized documents made us to do a simple test. We took the sample phonebook files that come with *Tamino* (they are stepwise sized from 1000 entries up to 8000 entries, which matches 470KB to 3,7MB), tried to insert them into the database and then run some queries.

Using *Xindice* out of the box, with the default maximum main memory setting of 64MB, resulted in memory exceptions and wrong or empty query results. This malfunction could be eliminated by increasing this memory limit to 256MB. Though we then got correct query results, we noticed that inserting more big sized documents into the database dramatically decreased query speed.

The problem with big sized documents originates from the storage and indexing technique used by *Xindice* [36]. Documents are stored as a whole into the database and each time a part of the document needs to be read or written it must be completely loaded into the memory. Furthermore indexes do not store positional information of tags in documents, but are solely used to find candidate documents for queries and updates.

4.3.4 API's

Xindice supports the XML:DB API for Java and a language independent XML-RPC API, which takes the place of the *CORBA* API of version 1.0. Server side XSLT processing is currently not available.

4.3.5 Documentation

There is a documentation, which is dedicated to the latest version of *Xindice* and reflects the changes made to the database. While it can be said to be accurate, currently it lacks of a comprehensive installation guide and a description of the XML-RPC API, which is needed when working with programming languages other than Java.

4.4 eXist

eXist [35] [38] is another Open Source project founded by Wolfgang Meier. Like *Xindice*, *eXist* is completely implemented in Java.

4.4.1 Installation

Installing *eXist* is simply accomplished by copying and unpacking the supplied *.zip or *.war file to a proper location of a hard disk and by editing an environment variable.

The *eXist* database server supports three modes of operation: Standalone, servlet and embedded mode, each of which has its own advantages and disadvantages. In order to run *eXist* in servlet mode the distribution already contains the servlet engine *Jetty*, so there is no need to install one. For starting and stopping the server and client applications, there are some batch files supplied. We had to edit these batch files slightly to get them to work.

4.4.2 Applications

eXist comes with an integrated, combined graphical and command-line Java application called *Client Shell*. It offers functions for managing collections, querying and uploading data and for creating database backups. This application is similar to the previously mentioned *XMLdbGUI*, which can be used with *eXist*, too.

Furthermore *eXist* comes with a local copy of its web site, which includes several simple forms for upload, query and a web administration interface.

All in all, to adjust database specific features, one still has to edit the configuration file of *eXist* with the help of a text editor, but the daily used functions are covered sufficiently by the *Client Shell* and by *XMLdbGUI*.

4.4.3 Database features

eXist currently supports XPath and XUpdate and support for XQuery is on its way. Though, on the one hand XPath is implemented with extensions, offering additional functions to speed up query times, its implementation is not fully W3C compliant, because it has not been finished yet.

To improve query speed *eXist* makes use of an automated full text index, which can also be turned off if necessary. Furthermore the user is able to adjust the nesting level of the index, which means that solely element tags down to a distinct hierarchy level are covered by the index. This indexing strategy seems to be a good tradeoff between user comfort and index size. Hereby, the database user does not have to think too much about indexes, but query speed and storage costs can be kept within an acceptable range.

In contrast to *Xindice*, *eXist* uses a B-Tree data model to organize XML documents. Additionally, document fragments are stored as persistent DOM [37]. Due to this, *eXist* can be expected to handle big sized documents better than *Xindice*.

4.4.4 API's

eXist supports a number of different API's, depending on the server mode used. First of all is the XML:DB API, the de facto standard Java API for XML Databases. The feature to query the database from a web browser is provided by the HTTP-API. This API also supports server side XSLT processing. Furthermore, there are two language independent APIs, namely XML-RPC and SOAP. The second one is implemented as a Java servlet and therefore only available when running *eXist* within *Tomcat* or any other servlet container.

4.4.5 Documentation

Though *eXist* has the smallest documentation set out of the three databases, it is still useable and informative.

4.5 Conclusion

Taking into account that the *latest version of Tamino* has undergone major improvements and hopefully also got a less complicated installation procedure, it will be the first choice, if one does not mind its costs. It convinces by providing a well done set of client applications and, last but not least, a really extensive help system.

If *Tamino* is too expensive and if one can temporarily live with an incomplete implementation of XPath, *eXist* is the way to go. Though it is still in beta state it seems to be stable enough to be employed for a project. It provides sufficient and accurate documentation in order to implement own client applications.

Next choice would be the *elder version of Tamino*. But then one will have to live without the feature to update documents with XUpdate, what can become cumbersome, very soon.

Unfortunately, the current version of *Xindice* can not be recommended by us to be utilized for this project. The missing XML-RPC documentation, the problems with big sized documents, the lack of support for server side XSLT processing and last but not least the recurring rumor that it should be retired, give it the lowest rank out of the three candidates. But future will show what happens to it.

Chapter 5

Design

"Everything should be made as simple as possible,
but not simpler."

Albert Einstein

5.1 Overview

This chapter discusses the design of the *Studierstube XML Database* system. Having analyzed and compared the usability of the three candidate databases (see previous chapter), we decided to employ *Tamino*, a commercial database system of Software AG. Now, we will elaborate a system concept, that meets our needs and integrates the features offered by *Tamino*.

Starting from the specification we first analyze the future operating environment of our project. Then we work out a number of common design issues like portability and reusability, in order to ensure and improve the quality of our work. The topics outlined here serve as a basis for the next chapters, where we select an appropriate programming interface for the client and develop a model of the planned system architecture. Then we split up the model into two parts, namely server and client. Each of these parts is then discussed and refined separately. Finally, we evaluate the end result and try to point out the pros and cons of our work.

5.2 Environment analysis

A system can hardly ever be built completely on its own. Most of the time you will have to consider the environment, in which the system has to operate. Just like in our case, where the operating environment is given by the local area network of the Institute. Hence, we need to have a look at this network in order to properly

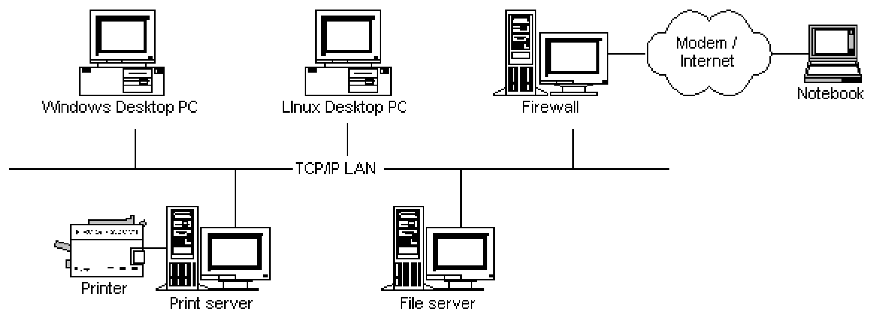


Figure 5.1: Local area network of the Institute

integrate the database system. Figure 5.1 on page 48 illustrates the basic structure of the network.

Staff members and students work on Desktop PCs running different operating systems like Microsoft Windows or Linux. The TCP/IP network, which they are connected to, enables them to share common resources, like printers and file servers. Additional to the local workstations, the resources can be accessed from outside of the LAN through a firewall. These are important details, because we want our database to be accessible in the same ways.

Recent projects of the Institute deal with navigation systems and will use our database as primary storage medium. In the context of their work, the collaborators of these projects often have to leave the building and thus have to access the network from outside. We have to consider that, when we choose one of the database interfaces provided by *Tamino*.

The technology used in this network to access resources is the widely adopted Client/Server architecture. The *Tamino* database system is designed following the paradigms of this technology. It provides a database server and several interfaces, which can be used on the client side to access the database. Thus, it perfectly fits into the network of the Institute and we can benefit from utilizing the Client/Server model and get enhanced flexibility, centralized management of data, lower total cost and so forth.

5.3 Design issues

Generally, developers tend to concentrate on the functional features of their architectures, and seldom address the ways in which their architectures support quality concerns such as portability, modularity and usability [1]. We will try to overcome this deficiency by defining a set of design criteria according to the requirements of our project.

The functional aspects of the system have already been outlined in the specification. In this chapter, we will work out the characteristics of a solution to our problem. Neglecting the constraining facts of the real working environment at first and thinking about an ideal system, helps to get the most out of our work and allows to obtain a broader view of the problem domain. Moreover, by clarifying things that remain informal otherwise, we get some kind of checklist that serves as a basis for design decisions. The considerations made here will influence the overall system design and have a major impact on which components we chose.

In the following sections we investigate software characteristics like portability, modularity and reusability and evaluate whether they are important to us or not. For those, considered to be important, we will outline what has to be done to achieve them. This also includes characteristics, which are regarded as negligible at the moment, but could possibly turn out to be essential for future versions of the project.

5.3.1 Portability (P)

As today's PC systems become more and more heterogeneous, designing portable software becomes an important issue. *Studierstube* is available for several operating systems (Linux, IRIX, Windows). Most important of all are Windows and Linux, which run on the desktop computers of Institute's staff members.

Hence, it is required that we consider this when designing our software. At least the client side of our system should be portable between these operating systems. Generally speaking, it will not be necessary to demand such a flexibility from the server side. Since *Tamino* is also available for above operating systems, except for IRIX, we should try to accomplish this portability for the server, too. Thus, we demand:

(P1) The entire system, including the server, should be portable between different operating systems.

We can realize this by selecting appropriate components and programming languages, which are also available for mentioned operating systems.

5.3.2 Modularity and Reusability (M)

Though the importance of modularity seems to be widely appreciated, the enormous increases in the quantity of software production in the past decades have not yet led to a similar increase in the quality of software. Modularity is the key to improve software quality and to build robust applications [2]. Sticking to this paradigm, we can acquire reliable and reusable components and reduce the effort of testing the implementation. Furthermore, a modular design enables us to easily exchange parts of the system at a later point of time, if necessary. Since these are worthwhile features in general, we demand that:

(M1) The system should be built of components, which can be exchanged and reused later.

In order to reach this goal, we apply the techniques of object oriented design, information hiding and modularization of both, data and processing. We will split up the system into small, reasonable units, with tight interfaces between them.

5.3.3 Usability and Acceptability (U)

Usability is a function of the ease of use and the acceptability of a product; where ease of use affects the user performance and satisfaction, and acceptability affects whether or not the product is used [3]. Usability related questions usually deal with the design of graphical user interfaces in order to improve the productivity and acceptance of an application. Likewise, the same characteristics can be applied to programming interfaces, which are the user interfaces for developers. According to [4], the usability of the languages and libraries that developers use, even have a significant impact on their ability to successfully complete a set of development tasks.

Serious usability work includes tasks like investigating the target user group, creating usage scenarios, running usability studies and so forth. Since this is out of the scope of our work, we will take an easier approach and try to improve the usability of the client by keeping it as simple as possible, while still providing all of the required features. We should aim to get:

(U1) A simple, easy to learn and easy to handle API

(U2) An easy to use setup procedure for the client

5.3.4 Performance (S)

The tests of the *Xindice* server revealed (see previous chapter), that we can not simply act on the assumption that the database system delivers an acceptable performance. Depending on the structure and size of the retrieved data, the response time of a query can vary heavily. Additional to the time needed by the database server to execute a query and deliver it to the client, some time will be required to accomplish several post processing steps on the client. An overall response time for a typical query beyond a few seconds will frustrate users and could even demolish our work completely. Therefore we demand that:

(S1) The response time of a typical query, including post processing, should be kept within the range of a few seconds (at maximum 5 seconds).

Since we can hardly impact the performance of the server, apart from adjusting its configuration, performance tuning will mainly be an issue of optimizing the source code of the client.

5.3.5 Extensibility (E)

This section discusses topics regarding possible future extensions to the database system. This mainly covers typical database related features, which are standard in most database products, but not explicitly required by our specification. Nevertheless, we want to consider these features in our design, that an easy integration is possible at a later point of time:

(E1) The client should be completely independent of the particular database used, in order that the database server can be transparently replaced.

(E2) A subsequent integration of transactions should be considered.

(E3) User authentication should be considered.

(E4) Support for binary data and Unicode should be considered.

5.3.6 Scalability, Availability and Security

Scalability, Availability and Security are important aspects of system design in many projects, especially when they deal with databases. However, we do not consider these characteristics to be relevant to our project. Anyhow the former two are mainly a matter of the database server and the latter is partially accomplished by our demand for authentication (E3).

5.3.7 Cost

Since this is a non-commercial project, we do not have a big budget at our disposal that can be spent on resources. Therefore, we will have to rely on royalty-free software, preferably Open Source Software, when we are in need of additional third party products.

5.4 Selecting an interface

Preceding to the system design we have to select the interface we want to use at the client side. Each interface needs additional components in order to be able to work properly. Therefore, the decision for a particular interface has a great impact on the structure of our system, especially to that of the server.

Tamino provides several ways to access the database, among them are interfaces for Java, Java Script and Microsoft.NET / C#. These three options are out of question to us, because we have to use the implementation language of *Studierstube*, namely C++.

When reducing the available possibilities on the basis of programming languages, the following interfaces are left:

- *Tamino API for C*
- *HTTP Client API for ActiveX*
- Native HTTP Client API

The following sections investigate these interfaces in order to have a basis for our decision. If appropriate, we refer to the previously outlined design characteristics by their code.

5.4.1 Tamino API for C

The *Tamino API for C* allows client applications to access a *Tamino* XML Server in a direct way, without going through a web server, like most other *Tamino* APIs. Since this API is entirely written in the programming language C, it is very well suited for client applications written in C or C++. The *Tamino* API for C is available for Windows as well as for UNIX platforms. It provides roughly the same functionality as the *X-Machine* programming interface. (The *X-Machine* interface is a low level interface to *Tamino*, which offers a set of commands for storing, retrieving and deleting database objects, creating or erasing collections or schemas, performing transaction processing and diagnostic testing.)

In order to run an existing application that uses the *Tamino API for C* one needs the library for webserverless access and a local installation of the Software AG product *eXtended Transport Services*, which is part of the *Tamino* setup.

The Pros and cons of this interface are:

+ Portable between platforms (P1)

+ Good performance due to direct access to the *X-Machine* (S1)

+ Supports transactions and user authentication (E2,E3)

+ Supports binary data and Unicode (E4)

– Does not support server side XSLT processing

– Needs a local installation of *eXtended Transport Services* (U2)

5.4.2 HTTP Client API for ActiveX

The *HTTP Client API for ActiveX* consists of two controls, which communicate with the *Tamino* XML Server at HTTP protocol level. One is for accessing and manipulating XML documents and the other for non-XML documents, including binary data. It is a more convenient way of communicating with the server than using the native HTTP protocol. Since it uses ActiveX technology, it is only available for the Microsoft Windows operating systems. The API offers a number of properties and methods to manipulate data in *Tamino* via document names, perform queries, local and distributed transactions and for lock management. The API's methods return a DOM object as a result or require a DOM object as input. The DOM model supported by the ActiveX API is the Microsoft DOM.

In order to use the API from within an application, you need the Microsoft Foundation Classes library and Microsoft Internet Explorer version 5.x or 6.x to be installed on the system.

The Pros and cons of this interface are:

+ Supports transactions and user authentication (E2,E3)

+ Supports binary data and Unicode (E4)

– Does not support server side XSLT processing

– No portability between platforms due to the ActiveX technology

– Reduced performance because of the use of DOM (S1) Services (U2)

5.4.3 Native HTTP Client API

Last but not least, the database can simply be accessed by utilizing the HTTP protocol. This requires a prior installation of an HTTP server on the server computer and an additional module, which connects the server to the database. As with the

Tamino API for C, the native interface communicates directly with the *X-Machine*, and thus, shares the same functionality.

Furthermore, the HTTP API can be used in conjunction with the *Tamino Passthru Servlet*, a Java component, which runs on the server and supports the execution of XSLT stylesheets. The *Passthru Servlet* in turn can only be executed in a proper system environment, namely a java servlet engine.

The Pros and cons of this interface are:

+ Portable between platforms (P1)

+ Good performance due to direct access to the *X-Machine* (S1)

+ Supports server side XSLT processing

+ Does not require additional software on the client (U2)

+ Supports transactions and user authentication (E2,E3)

+ Supports binary data and Unicode (E4) Services (U2)

5.4.4 Conclusion

As one can easily see, the best choice out of these three is the last one, the Native HTTP interface. Apart from the mentioned properties, there are further arguments to choose this interface:

First, the HTTP protocol is designed for transporting text data. Since XML documents are completely made of plain text lines, this is quite suitable for that.

Second, most other vendors also provide an HTTP interface for their XML Database. Thus, migrating to a different database, if needed, is much easier than it would be, when we would use the proprietary API for C.

Third, server side XSLT processing is only supported by the *Tamino Passthru Servlet*, which implies the use of the native interface. This way, we can save work and must not develop a server process, which implements the XSLT processing.

Fourth, the usage of the HTTP protocol facilitates the routing of database queries over firewalls and proxies.

5.5 System architecture

This section discusses the architecture of the system and the components it is composed of. Before proceeding to the description of the system model, a few words have to be said about the standard XML Database API.

5.5.1 XML:DB API

Since the XML:DB Initiative already defined an open standard for accessing XML Databases, namely the XML:DB API, one could argue that this is the right way to implement the client interface. Although we considered to follow this standard at first, we decided to use our own approach for the following reasons:

- Due to its architecture, the XML:DB API is quite flexible to use, but relatively complicated to handle. This is in contrast to our demand for usability (U1).
- In order to be as flexible as possible the class framework of the XML:DB API covers a lot of features like a driver concept, a service mechanism, collection management and several more. While these are worthwhile features in general, they are not needed for our solution. Thus, we would have to invest a lot of work in order to implement the basic framework, just to comply with the standard.
- The XML:DB API neither supports the query language XQuery nor the execution of server side stylesheets. Thus, we would have to implement a new service extension, for instance, like *XQueryServiceWithStylesheets*, which does not comply with the standard. Which in turn would foil the utilization of the standard.
- Though the XML:DB API is defined using IDL, an implementation in C++ can be quite quirky. See the *XinCJ* implementation to get an impression of what that means.

Thus it is clear, why a simple, but efficient implementation of the client interface looks much more reasonable.

5.5.2 System model

After completing all preparatory work for the design, we are now ready to create a model of our system. It is meant to describe the logical structure of the system and should outline all of the major components the system is composed of. Furthermore, each component should get a well defined task and a well defined interface in order to be easily replaceable (M1).

By employing the Client/Server architecture for our project, we are able to split up the system into two major blocks, namely client and server, which are interconnected by an interface based on the HTTP protocol. Hence, we can treat them as separate units and design them (almost) independently.

Each of these blocks can then be further divided into several components. We identify these components by analyzing the requirements of the specification and by applying the paradigms of object oriented design, most notably encapsulation and

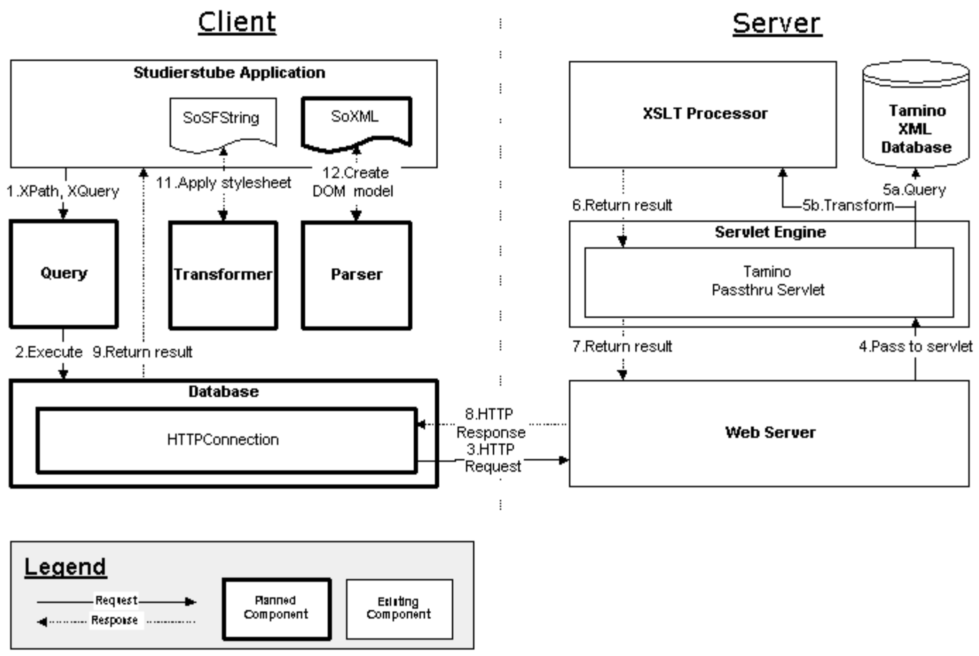


Figure 5.2: Dataflow diagram of the system architecture

inheritance. Using object oriented terminology, we can also speak of objects instead of components.

Figure 5.2 on page 56 illustrates the system architecture by means of a dataflow diagram. It shows a typical scenario of the system, where a *Studierstube* application queries the *Tamino* database utilizing the proposed client interface. Rectangles framed by a bold line denote planned components, which have to be implemented by us, whereas existing applications and components are marked by a thin frame. Arrows between these components indicate the direction of the data flow that is caused by the query. Solid arrows show the data flow of the client request and dotted arrows the response of the database server. The dashed line in the middle of the diagram divides the system into client and server, which are discussed separately in the following sections.

5.5.3 Client

The left side of figure 5.2 depicts the client, which is designed according to our own approach instead of implementing the XML:DB API. All topics discussed in this section refer to this part of the diagram. The components with a bold frame together form the *Studierstube XML Database API*, which can be utilized by *Studierstube* applications to query the database and to process XML data. Each of these components can be thought of as an instance of a C++ class offering various methods.

The most important part of this diagram is the *Query/Database* pair, which is used to execute database queries. This concept is adopted from the Borland VCL class library, because of its simplicity and effectiveness. The basic idea is to have a dedicated, passive container class, which subsumes all parameters related to a database query in a single component. This set of parameters, which is represented by the class *Query*, includes the XPath or XQuery script and the path to the server side stylesheet. Due to this design the same query can be consecutively executed on different databases.

The class *Database* is an abstraction of a database instance available on the server. (Remember: *Tamino* is capable of serving multiple databases at the same time.) The job of this class is to store network related parameters, manage an HTTP connection to the server and to translate query parameters to HTTP requests. This way we are also getting a layer, making it transparent, which interface is used to access the database.

According to our demand for reusability (M1), we provide the HTTP connection in form of a dedicated class, encapsulating the parameters and features, which are associated with this protocol.

The remaining components implement the post processing features of the API, which can be optionally applied to a query response. The class *Transformer* offers methods to execute a locally hosted stylesheet. Since there are freely available XML processors, which can be downloaded from the Internet, we use one of these packages to implement this feature. The same applies to the class Parser, which uses an XML Parser to create a *SoXML* DOM model from an XML document and vice versa.

For the lack of space, the *SoXML* DOM model is depicted only by a single symbol in figure 5.2. Usually, this model consists of a tree of nodes, which represent the elements and tags of the corresponding XML document.

5.5.4 Server

The right side of figure 5.2 shows the server, which is completely composed of existing components. This means that we do not have to develop any new components for this part of the system. All we need to do is install, configure and optimize the components of the server in order to make them work properly together.

Since we want to perform server side stylesheet processing, we need an XSLT processor on the server, which can be used by the *Tamino Passthru Servlet* to transform the query results. As with the client, this feature will be implemented by integrating one of the freely available software packages.

The *Tamino Passthru Servlet* in turn needs a proper operating environment in order to be executed. For this purpose, we have to install a Java servlet engine and embed the *Passthru Servlet* into it. Normally, an additional web server would not be required to run a servlet, but in this implementation, the *Passthru Servlet* utilizes the web server to access the database by forwarding the query request to it. For the ease of reading we have omitted this step in the diagram of figure 5.2. Furthermore, the web server is needed to run *Tamino Manager*, the administration application of the database system.

5.5.5 Workflow of a query

Having introduced all of the components of the client and the server, this section explains how they work together in order to query the database. Therefore we discuss the scenario depicted in figure 5.2. The numbers on the arrows in this diagram indicate the order of the performed steps. In the following, we will refer to these numbers.

(1.) If we want to query the database, we have to use an instance of the class *Query* and pass the query script and the path of the server side stylesheet to it.

(2.) Then we execute the query on a *Database* object, which creates a corresponding HTTP request.

(3.) The *Database* object sends the request over the network to the web server by utilizing the *HTTPConnection* component.

(4.) The server forwards it to the servlet engine and thus to the *Tamino Passthru Servlet*.

(5.-8.) Then the servlet analyses the HTTP request and extracts the part, which contains the path of the stylesheet. The remaining HTTP request is then sent back to the web server (not depicted here), which executes the query on the database (5a.). When the database server returns, the servlet applies the stylesheet to the query result (5b.) and delivers the processed data back to the *Database* object (6.-8.).

(9.-12.) The *Database* object passes the query result to the application (9.), once the HTTP response is validated, and thus completes the database query. The query result can now be post processed using the classes *Transformer* (11.) and *Parser* (12.).

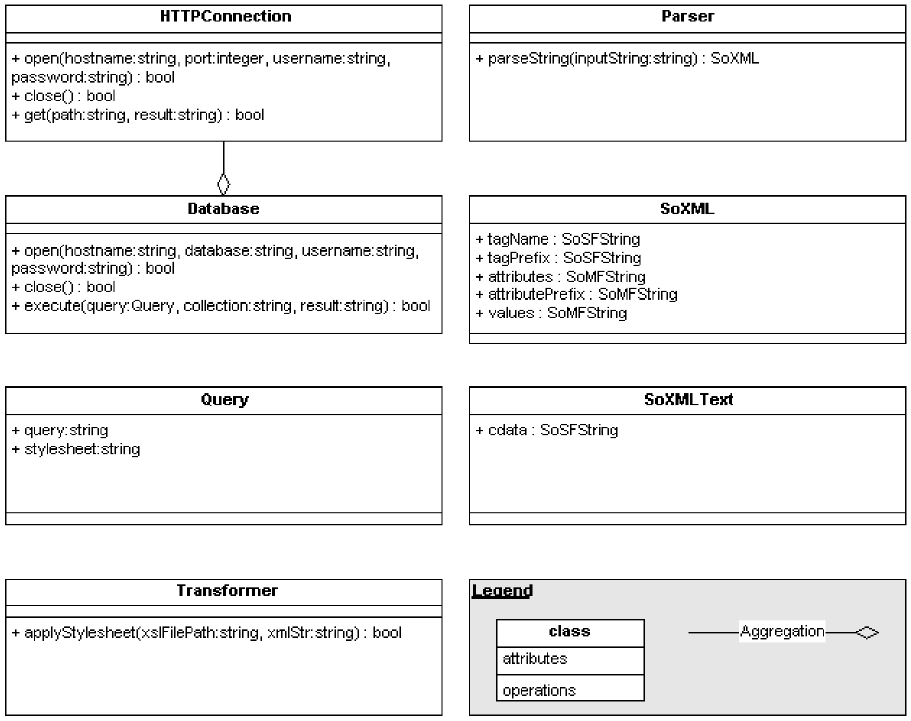

HTTPConnection
+ open(hostname:string, port:integer, username:string, password:string) : bool
+ close() : bool
+ get(path:string, result:string) : bool
Parser
+ parseString(inputString:string) : SoXML
Database
+ open(hostname:string, database:string, username:string, password:string) : bool
+ close() : bool
+ execute(query:Query, collection:string, result:string) : bool
SoXML
+ tagName : SoSFString
+ tagPrefix : SoSFString
+ attributes : SoMFString
+ attributePrefix : SoMFString
+ values : SoMFString
Query
+ query:string
+ stylesheet:string
SoXMLText
+ cdata : SoSFString
Transformer
+ applyStylesheet(xslFilePath:string, xmlStr:string) : bool
Legend
class
attributes
operations
Aggregation

Figure 5.3: UML class diagram of the Studierstube XML Database API

5.6 API Classes

This section presents the design of the *Studierstube XML Database API* classes in form of a UML class diagram (see figure 5.3 on page 59). For the ease of reading we included only the most important attributes and operations. Most classes are self-explanatory, because we have already discussed their features in the previous chapter. Therefore, we will merely discuss topics, which are in need of further explanation.

5.6.1 HTTPConnection

Normally, HTTP is a stateless protocol and thus would not require a dedicated start- and endpoint as provided by the "open()"/"close()" methods of the *HTTPConnection* class. But the Keep-Alive extension to HTTP, as defined by the HTTP/1.1 draft, allows persistent connections. These long-lived HTTP sessions allow multiple requests to be sent over the same TCP connection, and, as a consequence, will speed up database queries. Moreover, adding the "open()"/"close()" pair to these classes facilitates to replace the HTTP protocol by any connection-oriented protocol at a later point of time, if necessary (M1).

In order to support authentication (E3) we also added the parameters ”username” and ”password” to the ”open()” methods of the classes *HTTPConnection* and *Database*.

5.6.2 String as query result type

Examining the ”execute()” method of the class *Database* reveals that the data type of a query result is a string. This is in contrast to our demand for the support of binary data and Unicode (E4). Here we preferred simplicity (U1) over extensibility (E4). Our first draft of the API provided a resource class to store arbitrary query results, but this turned out to be impractical. Anyway, binary data can be encoded in strings, for example in form of Uuencoded text, if really needed.

5.7 Conclusion

This section resumes the system design and tries to point out the pros and cons of our approach with the help of the following list:

Pros

+ Depending on the components we choose, we can achieve a completely portable system. (P1)

+ The client system can even be divided into three independent parts: *Query&Database*, *Transformer*, *Parser&SoXML* (M1)

+ We have developed a fairly simple solution for the problem (U1), which can be installed simply by adding the modules to the *Studierstube* setup (U2).

+ Due to direct access to the *X-Machine*, we can expect a good performance (S1).

+ The usage of HTTP facilitates the exchange of the underlying database (E1).

+ Transactions can be supported by adding appropriate methods to the *Database* class (E2).

+ User Authentication has been considered (E3).

Cons

– Support for binary data and Unicode was dropped for simplicity (E4)

Chapter 6

Implementation

In this chapter we present the implementation of our project. We give an in-depth explanation of the system components and show how they work together. In this regard, we investigate the classes and applications involved in the implementation and discuss topics concerning the performance of the system. Furthermore, at this point, we replace all remaining abstract components of the design with appropriate, real software packages. As in the previous chapter, we discuss client and server separately.

6.1 Client

The client side of the system consists of a C++ class library and several third party software packages. The classes of this library can be divided into three logical groups: The database access classes, the transformation classes and the classes for generating the *SoXML* DOM Model. The standard C++ class string serves as an interface between these groups. Thus, each of them can be compiled and used independently. In the following we discuss each of these groups separately.

6.1.1 Database access classes

The database access classes provide the database query features of the client API. As outlined in the design chapter, we use the HTTP protocol to access the *Tamino* database. Thus, we need an appropriate, portable software package, which enables us to communicate through a TCP/IP network. Since the *Studierstube* system uses the ACE package to implement networking functions and ACE is a well designed, highly portable and approved C++ network library, we chose to use ACE as well. However, the drawback of utilizing ACE is - although it supplies the basic networking components for building client applications - it does not provide direct support for the HTTP protocol. Therefore, we had to implement the protocol by using the ACE

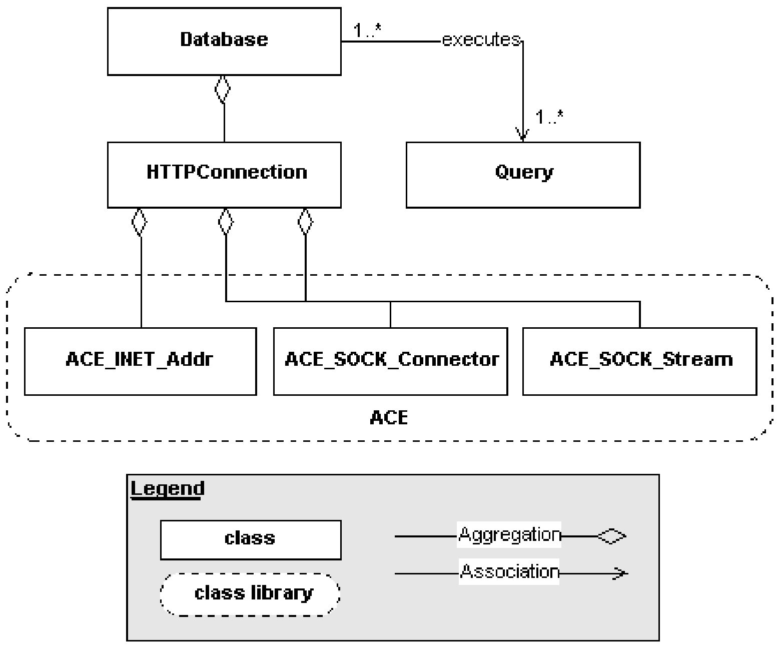


Figure 6.1: UML class diagram of the Database access classes

socket classes as depicted in figure 6.1 on page 62.

The *HTTPConnection* class encapsulates the implementation of the HTTP protocol in a way that it is easily reusable by other applications and classes. Hence, the class *Database* can utilize an instance of *HTTPConnection* to transmit the query parameters as corresponding HTTP requests. The next section shows how this translation is accomplished.

Accessing Tamino using HTTP

One of the central parts of the *Tamino* XML Database is the *XML-engine*, also referred to as *X-Machine*. This engine provides an HTTP based interface, which offers various operations to access the XML data store. The *X-Machine* interface allows to send special commands to the engine either encoded in URLs, when using the HTTP GET method, or as multipart form data, when using the HTTP POST method. If the GET method is used, the commands are provided in the URL as keyword/value pairs in the search part of the URL (The part that is separated by a question mark.) This way the *X-Machine* is accessible through a wide range of operations. Those utilized by the *Database* class are described in the following table.

Command	Meaning
_xql	Retrieves one or more objects using the XPath query language
_xquery	Specifies a query using the XQuery query language
_xslsrc	Executes a server-side stylesheet; additionally supported by the *Tamino Passthru Servlet*

Assume that we want to get a list of all surnames of our staff members and their personal data is contained in the database "addresses" and the collection "staff". Using the query language XPath, the corresponding HTTP request would look like this:

```
http://hostname/tamino/addresses/staff?\_xql=person/surname
```

Where "hostname" stands for the IP address of the database server and "person" for the XML root tag of the data. Multiple keyword/value pairs can be connected by using an ampersand (&).

The objects returned by this query are enclosed in the HTTP response body inside an XML document. By default, the document consists of elements and attributes defined in the *Tamino* namespace "http://namespaces.softwareag.com/tamino/response2" using the prefix "ino". See the section "Database access module" of the developer manual in order to get an explanation of how this document is mapped to our own namespace.

Limitations of the Apache web server

Some web servers restrict the length of the URLs that they can process. In the case of the *Apache* web server this limit is around 8KB for the total sum of all query parameters. Especially when using the insert and update features of XQuery, where whole XML objects have to be passed to the *X-Machine*, this limit can be exceeded with ease. Since we plan to process big sized objects, which are magnitudes bigger than the limit, we had to overcome this problem by utilizing the HTTP POST method instead of the GET method. This way we were able to shift the query scripts, which are stored in the parameter "_xquery", to the body of the HTTP request and so reduce the size of the URL. The remaining parameters are still transmitted in the search part of the URL.

The servlet path

The *Database* class provides a static member variable called "servletpath". This string variable holds the path, which points to the location of the *Passthru servlet.* By default, it is set to the value "ino/servlet/transform/tamino" and only depends on the configuration of the database server. It has to be updated, when the servlet is stored in a different location.

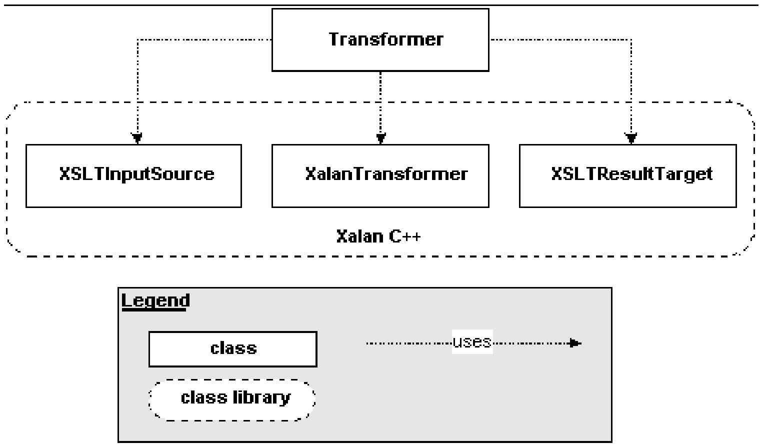


Figure 6.2: UML class diagram of the Transformation classes

6.1.2 Transformation classes

The second group of the client interface is represented by the transformation classes. It supports the execution of XSLT stylesheets in order to transform XML documents. Thus, we needed to find an appropriate XSLT processor, which can be used to implement this feature.

The current *Studierstube* setup already includes a recent version of the XML parser *Xerces* C++, which is developed by the *Apache* group. Since the members of this group are usually known for delivering well done products and they also feature a portable XSLT processor, namely *Xalan C++*, it was the candidate of first choice for our project. The *Transformer* class of the client API is utilizing this XSLT processor library (see figure 6.2 on page 64). In order to facilitate the handling of this library, *Transformer* builds a simple wrapper around the *Xalan* library and offers an interface, which is much easier to learn and understand.

6.1.3 SoXML DOM Model classes

The *SoXML* DOM Model classes cover the features to transform an XML document to a simple DOM model and vice versa. Since the *Xerces* C++ parser is already part of the *Studierstube* project, we employed it to implement the translation. In order to feed the parser with an XML document, which is contained in a Standard C++ stream, we had to implement an appropriate class providing streams as input source.

We accomplished this by deriving the classes *StreamInputStream* and *StreamInputSource* from the *Xerces* base classes *BinInputStream* and *InputSource* as depicted in figure 6.3 on page 65.

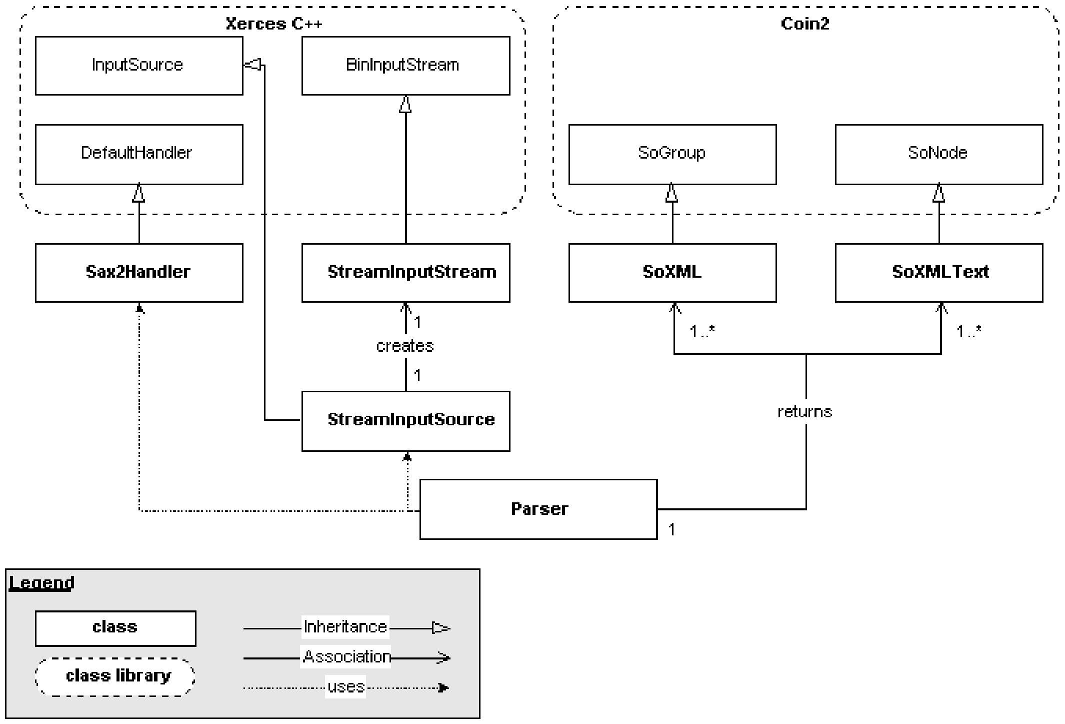


Figure 6.3: UML class diagram of the SoXML DOM Model classes

In order to be able to use the SAX interface of *Xerces*, we had to derive a new class from *Defaulthandler*, namely *Sax2Handler*. The *Sax2Handler* class implements the actual translation process by overwriting the virtual methods of it's base class.

The output of this translation process is a tree, which is composed of instances of the classes *SoXML* and *SoXMLText*. In order to access this tree with the scene graph operations of *Coin2*, we derived these classes from the *Coin* base class *SoNode* and *SoGroup*. Read the section "SoXML DOM Model module" of the developer manual in order to get additional details about these classes.

6.1.4 Optimizing the API

While developing the demonstration application *BAUMLBrowser* we found that the performance of the client API was somewhat poor. It took about 20 seconds to query and process the mid-sized BAUML object "Karlskirche_bd", what we found way too long. The basic steps involved in this operation are to query the database, apply a stylesheet in order to extract the result set ("getResult.xslt") and apply a second stylesheet ("bauml2iv.xslt") to transform it into a *Coin* script. After we profiled the concerned *BAUMLBrowser* method "getRepresentation()" we got the following surprising results:

Processing times before optimization

Operation	Time (seconds)	Time (percentage)
getRepresentation()	19.3 sec	100%
querying the database	1.9 sec	10%
applying the stylesheet ”getResult.xslt”	11.7 sec	60%
applying the stylesheet ”bauml2iv.xslt”	5.9 sec	30%

From the table above one can see that applying the two stylesheets took about 90% of the entire operation, while the query itself only took 10%!

Further investigation and profiling revealed the reason for this performance problem. The *stringstream* template classes *<sstream>* of the Microsoft C++ runtime library are implemented in a way, that slows down string stream processing. Fortunately, this library contains another string stream implementation, namely *<strstream>*, which performs much better. Simply by replacing the stream library, we were able to significantly speed up the processing, which is shown in the following table.

Processing times after optimization

Operation	Time (seconds)	Time (percentage)
getRepresentation()	5.9 sec	100%
querying the database	1.9 sec	32%
applying the stylesheet ”getResult.xslt”	2.3 sec	38%
applying the stylesheet ”bauml2iv.xslt”	1.7 sec	28%

The processing time of each operation is now about one third of the entire task and the overall performance gain is about 300%.

6.2 Server

This section discusses the implementation of the database server. Apart from modifying the *Passthru Servlet*, the work that had to be done was mainly a matter of installation and configuration.

The following section describes which components were chosen in order to implement the server and how they are configured to work together. Then we discuss why the *Passthru Serlvet* had to be adopted and the modifications which were made to it. The last section deals with some unexpected and strange performance problems which came up when the server was tested with big sized XML documents.

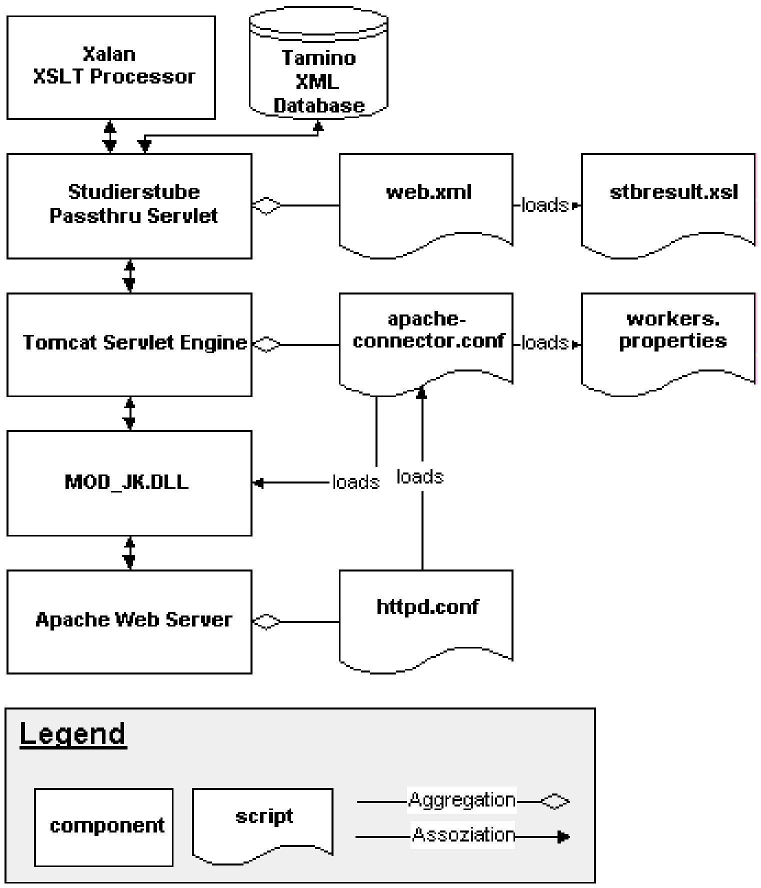


Figure 6.4: Diagram of server components and scripts

6.2.1 Server components

As outlined in the design chapter, an essential portion of the database server is made of third party software. Since we decided to employ the *Tamino Passthru Servlet* in our project, we also needed to choose a servlet engine and a web server in order to properly run the servlet. Figure 6.4 on page 67 depicts the logical structure of the server including the most important configuration scripts. In the following sections we only discuss the function of these components, (see the appendix of this work for a detailed description of how to install them).

Apache Web Server

Apart from the fact that several other components of the *Tamino* XML server require the installation of a web server, the *Tamino Passthru Servlet* needs one in order to query database. Basically, this task can be carried out by any web server, which is capable of forwarding requests to particular web locations. Since the *Tamino* setup procedure directly supports the use of the Microsoft IIS and that of the *Apache* web server, it is easiest to employ one of these two. As *Apache* is, in contrast to the Microsoft IIS, free of any charges and we only need it to implement the database access, we decided to employ *Apache*. Furthermore, *Apache* is available for several operating systems, including Windows and Linux (design request P1), and, at the time of writing, the most popular web server on the Internet and thus a well tested piece of software.

Tomcat servlet engine

Every Java servlet needs a proper operating environment in order to be executed. In principle the *Tamino Passthru Servlet* can be used with any Java servlet container that supports the Java servlet specification version 1.2. Since the main environment used by the Software AG developer team was *Tomcat 4* and the *Tamino* manual gives detailed instructions on installation and configuration, we decided to employ this version of *Tomcat*. Furthermore, it is freely available from the *Apache* website, completely written in Java (design request P1) and used in the official reference implementation for the Java Servlet technologies.

Note: There is also a later version of *Tomcat* available (version 5.x), but this one is not compatible with the *Tamino Passthru Servlet*, because it does not support the JSP 1.2 specifications anymore.

MOD_JK module

The MOD_JK module is a plug-in to *Apache* that handles the communication between the *Apache* web server and *Tomcat*. It is, of course, also available for UNIX platforms.

Studierstube Passthru Servlet

The *Studierstube Passthru Servlet* is an enhanced version of the *Tamino Passthru Servlet*. It supports the execution of an additional, fixed, server-side stylesheet. This is discussed in detail later in this chapter.

Xalan XSLT processor

Since we want to execute stylesheets on the server, we need an XSLT processor to accomplish this. The *Tamino Passthru Servlet* will work with any XSLT processor that implements the Java API defined in the JAXP 1.1 specification. There are several, freely available processors we can choose from, for example *Saxon*, *XSLTC*, *jd.xslt* and *Xalan-J*.

In order to get transformation results, which are as close as possible to that of the client (when using the same stylesheet), we decided to employ the Java version of *Xalan*, *Xalan-J*. This facilitates to shift the execution of a stylesheet from the client to the server and vice versa, without the need to modify it.

Configuration scripts

The configuration scripts depicted in figure 6.4 are an important part of the database server, because they define how the components are interconnected. In the following section we give a short description for each of the scripts.

httpd.conf is the main configuration file of the *Apache* web server. The *Tamino* setup modifies this file in order to enable HTTP access to the database and for the execution of its web-based services. Furthermore it includes a link to the configuration script of *Tomcat*.

apache-connector.conf is responsible for loading the MOD_JK module and defines the mount points of the servlets. Furthermore it includes a link to the workers.properties configuration script.

workers.properties gives *Tomcat* information about where to listen for requests from the *Apache* web server.

web.xml is the main configuration script of a *Tomcat* servlet. It defines the mapping of URLs to servlet code and provides a set of user-defined parameters, which can be used to configure the servlet. In the case of the *Studierstube Servlet*, one of these parameters points to the fixed server-side stylesheet ”stbresult.xsl”.

6.2.2 Studierstube Passthru Servlet

The *Studierstube Passthru Servlet*, also referred to as *Studierstube Servlet*, is a modification of the *Tamino Passthru Servlet*. It enhances the original *Tamino* servlet by adding support for the execution of an additional, fixed stylesheet. This stylesheet is executed prior to the user defined stylesheet, which can be specified by client applications using the ”stylesheet” member variable of the *Query* class.

Since there is no common standard for result sets of XML Databases so far, each database manufacturer is defining its own format. The purpose of the enhancement to the *Tamino Passthru Servlet* is to transform the query results of the *Tamino* database server into a form, which is specified by us. Without this transformation the code of the client applications would highly depend on *Tamino* and thus, make it very difficult to replace the database server at a later point of time. As mentioned in the previous section, the stylesheet defining the fixed transformation can be configured in the "web.xml" script of the servlet. Read the developer manual of the Client API for more details about the usage of this servlet.

Modifying the servlet

Due to a lack of source code documentation, it was quite difficult to modify the servlet and to integrate the extension. In order to facilitate further enhancements of the servlet, this section gives an overview of the modifications made.

The complete Java source code of the servlet can be found in the Java directory "com.softwareag.tamino.passthru". Included in this directory is a class called *TaminoFilter*, which is derived from the class *HttpServlet*. This is the only class which needed to be modified. In order to preserve the original source code, the first step was to copy it to a new class called *StudierstubeFilter*. Appropriate comments are given for all modifications being made.

At startup time of the servlet the method "init()" reads several parameters from the configuration file "web.xml". We added an additional parameter called "fixedStylesheet" to it, which contains the URL of the fixed stylesheet. The processing of the stylesheets is implemented in the "doGet()" method, which is called upon each HTTP GET/POST request. We had to restructure this method in order to add the extension and moved a portion of its code to a new method "cacheFile()", which caches the compiled version of stylesheets. Prior to the user defined stylesheet, which is stored in the HTTP parameter "_xslsrc", we apply the fixed stylesheet and write the transformation result back to the variable "xmlSource", which is then used by the remaining part as input source.

The following section lists the most important variables used and intends to facilitate the understanding of the "doGet()" method.

fixedStylesheet: A member variable storing the path to the fixed stylesheet; it is initialized by the method "init()"

xslstr: A local string variable storing the path of the stylesheet contained in the "_xslsrc" HTTP parameter.

xmlSource: A string variable storing the intermediate results of the stylesheet processing.

xmlResult: A string variable storing the final result of the stylesheet processing.

Compiling the servlet

The command line script "build.cmd" in the "\bin" directory of the *Passthru Servlet* compiles the source code and creates the Java library "passthru.jar", which can then be copied to *Tomcat's* servlet folder. For a successful compilation we needed the following prerequisites.

Sun Java SDK 1.3.x The *Passthru Servlet* was developed by using version 1.3.x of the Java SDK. Though it is not a problem to execute it under a later version, it did fail to successfully compile using Java SDK 1.4.1.

JSDK 2.0 The Java Servlet development kit.

JAXP 1.1 The Java API library for XML processing.

6.2.3 Performance problems

During the system test we stumbled over a strange problem regarding the performance of the *Tamino* database server. While the query response time was usually within the range of a few seconds when using a certain test data set, it abruptly became worse, after new XML objects were added to this set. Depending on the size of these objects the response time was even extended up to two minutes.

Luckily, with the help of a service member of Software AG, we were able to solve this problem. The internal cache size of the XQuery processor is set to a very low value by default and thus creates a bottleneck in the query engine. We were advised to insert the string parameter "XQuery_document_cache_size" (type REG_SZ) into the Windows registry to the key
"HKEY_LOCAL_MACHINE/SOFTWARE/SoftwareAG/
Tamino/servers/<database name>" and set its value to 100 (Megabytes).

After that, the original performance of the server was restored. Even if we doubled the size of the test data then it did not have a noteworthy impact on the query speed.

Chapter 7

Sample application

7.1 Introduction

Having completed the implementation of the client API and the modifications on the server, we wanted to test the software by means of a small sample application. In order to obtain something more useful than a simple demonstration application, we searched for a proper task in the scope of the *Studierstube* project. We wanted to run the software in a real world environment and, at the same time, achieve an application, which can be utilized later, too. At last the *SignPost* subproject offered what we were looking for.

SignPost is a *Studierstube* application that is able to guide a person through an unfamiliar building. It is an Augmented Reality navigation system, where the person wears a mobile equipment consisting of a head mounted display, a camera and a tracking system. *SignPost* relies on an XML-based data structure, called **B**uilding **AU**gmentation **M**arkup **L**anguage (BAUML), that holds geometric information of rooms and buildings, as well as the placement of markers, which allow to determine the current position of the user.

Since one of the future application fields of our XML Database is to provide BAUML data, we decided to write a small application, which is capable of browsing and manipulating BAUML documents stored in a database. The following sections present this application, called *BAUMLBrowser*, after giving a basic introduction to the BAUML language.

7.2 BAUML Language

BAUML is an XML language for the representation of geometric information. It allows to describe any object or set of objects by specifying the position of its vertices and surfaces in a three dimensional coordinate system. Some BAUML objects can be further refined and described in more detail by adding a set of child objects. The resulting data structure is a tree of BAUML objects, where each additional tree level refines the associated objects above. Due to its recursive definition, BAUML allows an infinite nesting of objects and thus enables to specify an arbitrary level of detail.

There are several types of BAUML objects, embedded in a type hierarchy, and each type defines a different aspect of the BAUML language. The most basic types are listed in the following text.

ObjectType is the basic object type. It defines an "annotation" tag and an "id" attribute. All other object types are directly or indirectly derived from it.

SpatialObjectType is derived from *ObjectType*. It is the basic type for all spatial objects and defines a "pose" tag and a "representation" tag.

SpatialContainerType is derived from *SpatialObjectType*. It is the basic type for all spatial objects containing child objects. It defines a "children" tag.

The geometric information mentioned above (vertices and surfaces) is stored in the "representation" tag of the *SpatialObjectType*. All spatial objects must be derived from this type and they must utilize this tag in order to specify their representation. The same counts for the "children" tag of the *SpatialContainerType*, which holds a list of child objects and thus describes the parent object in more detail.

These are all abstract types forming the base of the BAUML language. Abstract means that there must not be instances of these types, but instead they are used to derive further types from them. In addition to these abstract types, BAUML also defines concrete elements like *SpatialObject*, *SpatialContainer*, *Room*, *Building*, *ARToolkitMarker* and *Waypoint*, which are all derived from the above types. Each of these elements extends its basic type with additional tags and attributes specific to its needs.

In the following example, we use the BAUML language to define a model of a simple rectangular room (width=5, depth=10, height=3) consisting of a floor and four walls.

```
<SpatialObject id="Simple room" baseType="SpatialObjectType">
  <annotation>
    Simple room model for demonstration purposes
  </annotation>
  <representation>
    <Vertex position="0 0 0"/>
    <Vertex position="5 0 0"/>
    <Vertex position="5 10 0"/>
    <Vertex position="0 10 0"/>
    <Vertex position="0 0 3"/>
```

```
        <Vertex position="5 0 3"/>
        <Vertex position="5 10 3"/>
        <Vertex position="0 10 3"/>
        <Polygon type="floor" vertices="0 1 2 3"/>
        <Polygon type="wall" vertices="0 1 5 4"/>
        <Polygon type="wall" vertices="1 2 6 5"/>
        <Polygon type="wall" vertices="2 3 7 6"/>
        <Polygon type="wall" vertices="0 3 7 4"/>
      </representation>
    </SpatialObject>
```

Since this room has no doors or windows, we use the generic *SpatialObject* type for that. We define the model by specifying the eight vertices of the cuboid and five polygons (a floor and four walls). For this aim, the "vertices" attribute of a polygon tag contains a list of numbers, indexes of the cuboid vertices. Furthermore, we have annotated the object in order to document its use. If we use the *BAUMLBrowser* application this model can be depicted as in figure 7.1 on page 77.
This example only demonstrates the most basic features of BAUML language. Read chapter 5 of [5] in order to get a comprehensive description of the language concepts and more advanced features.

7.3 BAUMLBrowser Application

BAUMLBrowser is our test application for the *Studierstube XML Database API*. It was written to test the functionality and usability of the API and was meant to gain first experiences utilizing an XML Database in a real world application. Therefore, it covers all of the features offered by the API and thus can be seen as a reference for implementing *Studierstube* XML Database clients. Apart from that, it is a handy tool when dealing with BAUML documents, which are stored in an XML Database.

7.3.1 Features

BAUMLBrowser is a Microsoft Windows application, which is capable of browsing and editing BAUML documents stored in an XML Database. It covers features for viewing the structure of a BAUML document, displaying a graphical representation of BAUML objects, converting BAUML data from and to *Coin2* scripts and editing the nodes of a BAUML document. The editing capabilities include functions for inserting, updating and deleting BAUML objects. In addition to that, the application offers two special functions: A function for testing object intersection and one for testing point inclusion.

7.3.2 Core component

In order to be able to reuse the features of the browser in other applications, we have separated the parts, which are independent of the operating system. All functions, which are not related to the Microsoft Windows system, have been encapsulated in a class called *BAUMLBrowser*. Mostly, this concerns functions, which are not dealing with the user interface of the application. The *BAUMLBrowser* class builds a layer on top of the database API and offers functions to access and process BAUML documents. This way, the browser can be easily ported to other operating systems like Linux.

The appendix of this work contains a developer manual, which gives a comprehensive description of this class in order to implement a database browser.

7.3.3 User interface

Since BAUML is a data format, which is based on a tree structure, we decided to give the application an Explorer-like user interface. *BAUMLBrowser* is a so-called dialog based application, which means that its user interface basically consists of a single window containing all the widgets needed for operation. At the left side of the dialog (see figure 7.2 on page 77) one finds a tree view displaying the structure and the nodes of a BAUML document. It can be used to browse through a document and to select certain BAUML objects for manipulation. Right to the tree view is a *Coin* viewer window, which shows a graphical representation of the currently selected BAUML object. Furthermore, the dialog contains various widgets, which are used to connect to a database and to edit the contents of a BAUML document. The appendix of this work contains a user interface guide, which explains the meaning of the various widgets and describes how one can use the application to edit a BAUML document. Read it to see some screenshots of the application and to get further information about its user interface.

7.3.4 Implementation

During the implementation of the browser, we stumbled over a strange performance problem. This topic has already been treated in the implementation chapter and thus will not be discussed here any further.

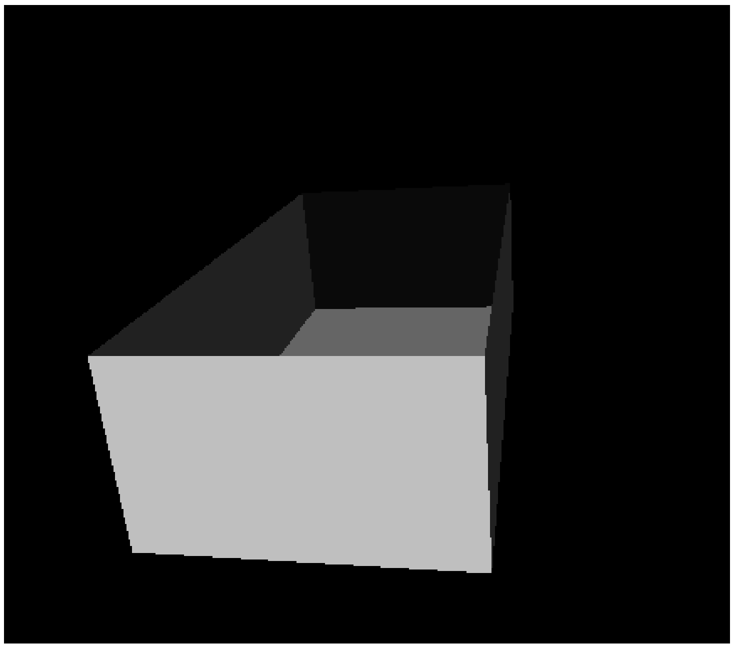
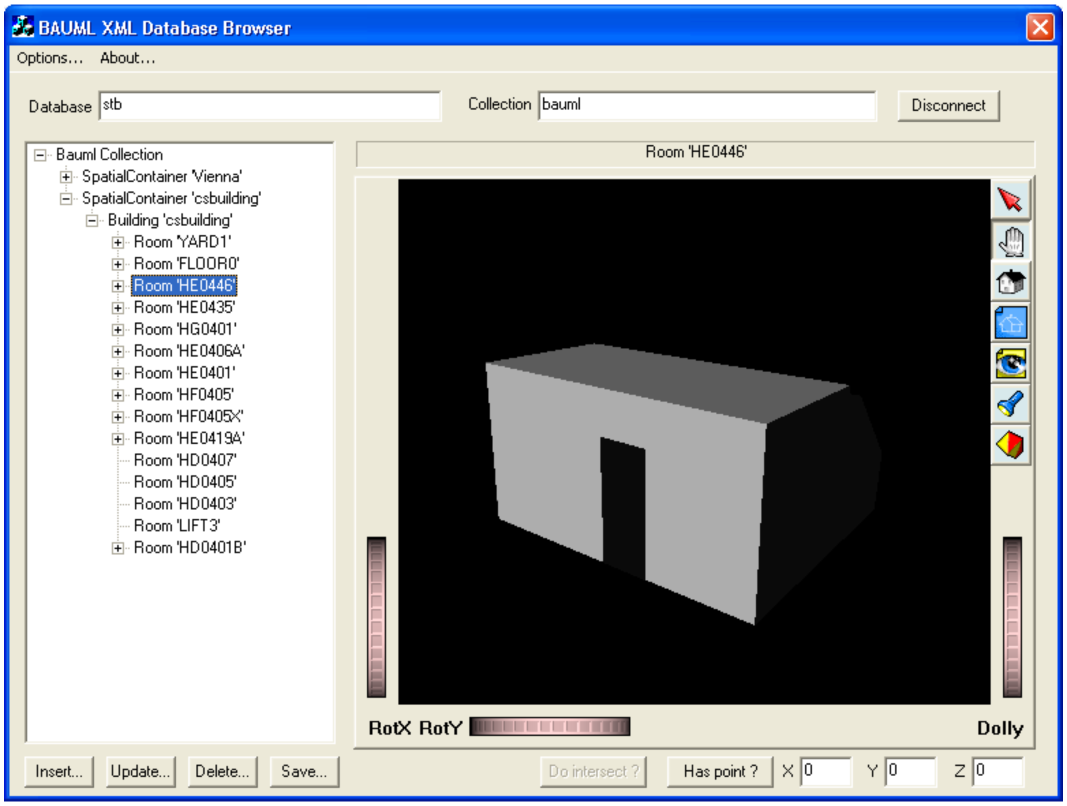


Figure 7.1: Simple room model



Database stb
Collection bauml

- Bauml Collection
  - SpatialContainer 'Vienna'
  - SpatialContainer 'csbuilding'
    - Building 'csbuilding'
      - Room 'YARD1'
      - Room 'FLOOR0'
      - Room 'HE0446'
      - Room 'HE0435'
      - Room 'HG0401'
      - Room 'HE0406A'
      - Room 'HE0401'
      - Room 'HF0405'
      - Room 'HF0405X'
      - Room 'HE0419A'
      - Room 'HD0407'
      - Room 'HD0405'
      - Room 'HD0403'
      - Room 'LIFT3'
      - Room 'HD0401B'

Room 'HE0446'

X 0 Y 0 Z 0

Figure 7.2: Screenshot of the BAUMLBrowser application

Chapter 8

Summary

This work is the base for employing an XML database system for *Studierstube* applications. It shows that it is possible to utilize this new technology in the context of Augmented Reality. Though there are still features missing, like full XML Schema support and insert, delete and update of root level documents, the installed database product proved to be a useful storage medium for a certain spectrum of XML documents. Furthermore, we are expecting, that one of the next releases of *Tamino* will also remove those last deficiencies.

On top of the database, we have implemented an API for *Studierstube* applications, the *Studierstube XML Database API*, which enables to seamlessly integrate XML Database functionality.

Thus, Augmented Reality applications can be realized, which work with big data sets, much bigger than it is possible with single XML documents on a file basis. So, for instance, it is possible to store and retrieve detailed, comprehensive graphical information, which originates from urban geographical information systems. Another conceivable application is to employ the database for storing the state of distributed *Studierstube* applications and utilize it to exchange information between them. The potentials of applications are virtually unlimited.

The API can be easily extended to support advanced database features like transactions and sub collections. In order to fully benefit from employing an XML Database system, we recommend to implement these features in a future project.

Appendix A

Database Comparison Charts

General			
Product name	**Tamino**	**Xindice**	**eXist**
Home URL	http://www.tamino.com/	http://xml.apache.org/xindice	http://exist.sourceforge.net/
Version	**v3.1.2.1** (Sept 2002)	**v1.1b3-dev** (Dec 2003)	**v0.9.2** (Aug 2003)
License	**Commercial**	**Open Source**	**Open Source**
Supported Platforms	• Windows NT • Windows 2000 • Sun Solaris 7 and 8 • IBM AIX 4.3.3 • HP-UX 11.0 • SuSE Linux for Intel and IBM S/390 • OS/390 mainframe	Implemented in Java, platform independent. Tested on • MS Windows • Unix/Linux	Implemented in Java, platform independent. Tested on • Windows 2000 • Windows XP • Linux (SuSE 7.1, Mandrake 9.1) • Solaris 8
Required Software before installation	• Apache Web Server >= 1.3.24 • Microsoft Java Virtual Machine >= 3805 (hard to get)	• Sun Java SDK Version 1.3 or higher • Tomcat Servlet Container Version 4.1.12 or higher •	• Sun Java SDK Version 1.4
Test system	• Intel Pentium III-800Mhz with 256 MB RAM • MS Windows 2000 Professional (German) with Service Pack 4 • Internet Explorer 6, Service Pack 4 • Windows Update (Security update & Java VM 3810) • MS Office 97 - Word, Excel, Power Point, Access (German), Frontpage 2000 (German) • MS Visual C++ 6 (English) • Sun Java 2 SDK v1.4.2_02 • Apache v1.3.24 Web server • Tamino XML Server 3.1.2 Patch Level 1 • Tomcat Servlet Container v4.1.12 • Xindice v1.0 and v1.1b3-dev • eXist v0.92		

<u>XML standards</u>

	Tamino	XIndice	eXist
XPath Path-like data query	Yes, in an adopted form called <u>X-Query</u> or <u>XQL</u>	Yes, implemented by <u>Xalan</u>	Yes
XQuery SQL-like data query	No	No	Yes, but not complete yet
XSD XML Schema Definition	Yes, in a form called <u>Tamino schema</u>, not fully compliant with the standard	No	No
XSL/XSLT Data Transformation (Server side)	Yes, by using X-Tension and a third-party XSLT processor	No	Yes, when using HTTP
XUpdate Data update	No	Yes, using the <u>Lexus</u> implementation	Yes, but not complete yet
SiXDML Simple XML Data Manipulation Language	No	Yes, by <u>sixdml</u> at sourceforge	No
DTD Document Type Definition	Yes, by importing into Tamino Schema Editor	No	Yes
XPointer Data Selection	No	No	Yes, partially
XLink Links across document boundaries	Yes	No	No
XInclude Including document fragments	No	Yes, partially	Yes

Database features

	Tamino	XIndice	eXist
Schema support	Yes, but not mandatory	No	No
Collection support	Yes, but no support for nesting	Yes, but no support for XPath queries across collections	Yes
Indexes	Yes, based on schemas	Yes, user indexes only	Yes, automatic full text index
Server extensions	X-Tension service (Java, C, C++)	Yes, by programming extensions for the servlet engine	Yes, by programming extensions for the servlet engine
Transaction support	Yes	No	No
Authorization mechanism	Yes, at different object levels	No	Yes, Unix-like access permissions for users/groups at collection- and document level
Multi user access, concurrency	Yes, designed for high concurrence.	Unclear	Yes
Update granularity	"The smallest unit of XML that Tamino can process is a document.You can use NodeLevelUpdate if you use Microsoft Internet Information Server (IIS) as your web gateway to Tamino."	Logically at node level and physically at document level	At node level
Backup/Restore strategy	Yes. Done manually in Tamino Manager.	No (Shutdown, Copy&Delete complete database)	Yes. Done via Command-Line or Gui Client.

Programmatic APIs			
	Tamino	**XIndice**	**eXist**
Language independent API's			
SAX Simple API for XML	Yes, as part of Tamino API for Java	Yes, implemented in Java	Yes, implemented in Java as part of the XML:DB API
DOM Document Object Model	● HTTP Client API for ActiveX (C++, Visual Basic), Java, JScript ● Tamino API for Java (DOM2)	Yes, implemented in Java	Yes, implemented in Java as part of the XML:DB API
Java only API's			
XML:DB API Data update (JDBC for XML)	No	Yes, Core Level 1 implementation + XUpdateQueryService + CollectionManagementServ ice	Yes, Core Level 1 implementation + UserManagementService + DatabaseInstanceManager + IndexQueryService
JAXP Java API for XML Processing	Yes	No	No
JDOM Java Document Object Model	Yes	No	No
Other	API for the X-Machine programming language	XinCJ a subset of the XML:DB interface for C++	

Documentation

	Tamino	XIndice	eXist
Available formats	● HTML ● PDF	● HTML ● PDF	● HTML
Size	~ 3000 printable pages	~ 80 printable pages	~ 50 printable pages
Quality	Very good, with many examples, although sometimes a bit digressing	Good, but missing a comprehensive installation guide and XML-RPC documentation	Small, but accurate
Search functions	Full-text HTML search engine	None	None

Supported network protocols

	Tamino	XIndice	eXist
HTTP Query the database with a web browser	Yes	No, just document retrieval by Xindice HTTP	Yes (in standalone mode)
XML-RPC Remote procedure calls using XML over HTTP	No	Yes, by Xindice XML-RPC module written in Java	Yes, by Apache XML-RPC
WebDAV Web-based Distributed Authoring and Versioning	Yes	Yes, by XinCon and Xindice Webadmin (in servlet mode)	Yes, by XinCon (in servlet mode)
SOAP Simple Object Access Protocol	No	No	Yes (in servlet mode)

Applications

	Tamino	XIndice	eXist
System configuration & administration	● Tamino Manager a graphical web application	● Xindice Webadmin a graphical web application (in servlet mode only)	● eXist Client Shell an integrated and combined Command-Line and Gui Java Client based on XML::DB API
DTD & Schema editors	● Tamino Schema Editor, a graphical Java application, editing a schema as a tree	-	-
Data editors	● Tamino X-Plorer a graphical Java application ● Tamino Interactive Interface, a simple form-based web-interface	● Xindice Webadmin a graphical web application (in servlet mode only) ● YAB is an Explorer-like browser for Xindice (Java) ● XDataFinder v0.6 is an application for browsing and querying XML files and native XML databases (Java) ● XinCon a web & webdav administration interface ● XIndice Browser v0.85 a tool to browse a local Xindice 1.0 database (Java)	● eXist Client Shell - an integrated and combined graphical and command-line Java application ● eXist x-admin interface - a simple administrative web-interface
		XMLdbGUI v1.3.1 is an application that allows the user to browse and modify databases conforming to the XML:DB API specification (Java)	
Data import/export	● Tamino Data Loader (Command line) ● Java Loader (Command line)	● Xindice command line client Export/import collections to/from files in a directory hierarchy	● eXist Client Shell - using the upload and backup feature

Pros & Cons

Tamino	XIndice	eXist
+ Good and extensive documentation + Good administration applications + Support for schema's and data types + Sophisticated database engine	+ Supports standard API's + Additional embedded mode	+ Supports standard API's + Good administration applications + Automated index generation + Additional embedded mode
- No support for XUpdate, it's not possible to update documents partially - Less support for standard API's - Needs MS Java VM >= 3805, which is hard to get - Expensive	- Does not support querying across collections and subcollections - Inappropriate storage technology - Only manual indexes - Is said to be retired	- No manual indexes - XPath and XUpdate implementation not complete yet - Still in beta state

Other free available XML Databases

Product	**URL**	**Comment**
dbXML	http://www.dbxml.com/	Seems that the Open Source Version is a performance reduced lite version of the commercial product.
Ashpool	http://ashpool.sourceforge.net/	Uses standard SQL92 syntax to query, add, update, and delete XML documents
4Suite, 4Suite Server	http://4suite.org/	4Suite is a collection of Python tools for XML processing and object database management.
Berkeley DB XML	http://www.sleepycat.com/products/xml.shtml	Berkeley DB XML is supplied as a library that links directly into the application's address space. No support for XUpdate.
DBDOM	http://dbdom.sourceforge.net/	Is an implementation of the DOM over a relational database.
ozone	http://ozone-db.org/	A Java based, object-oriented database management system. No support for XPath or XUpdate.
XDBM	http://sourceforge.net/projects/xdbm	Is "based upon the DOM standard".
XDB	http://zvon.org/index.php?nav_id=61	Built on a relational database.

Appendix B

Database API manual

The *Studierstube XML Database API* is a programming interface, which provides access to a *Tamino* XML Database. Using the API, you can query and update XML documents in a database, transform query results using XSLT and build a DOM Model for accessing the data. The API is logically divided into two modules, which can be used independently of each other:

- *Database access module* - Provides classes for database access and data transformation
- *SoXML DOM Model module* - Provides classes to build a DOM model based on Open inventor nodes

B.1 Database access module

Tamino XML Server is a data management system for storing XML documents. It helps with finding and managing XML data and to search effectively the information. A single *Tamino* server is able to run several databases, which can be accessed by name, and is able to supply many clients. A *Tamino* database is organized in form of collections. A collection is the largest unit of information within a database and can contain multiple XML documents. You can think of it as a folder in a file system, with the difference that it stores XML documents instead of files.

In order to search for information within a collection, *Tamino* supports two query languages, XQuery and XPath. Since XQuery is the standard query language in *Tamino* and a superset of XPath, the following sections will focus on it. Beside the ability to retrieve objects or parts of an object, *Tamino XQuery* is capable of inserting, updating and deleting objects and also to compose the query result using constructors. It, more or less, combines the features of XPath and XSLT in one comprehensive query language. Although we recommend to use XQuery, because this will be the future standard for XML Databases, it is up to you to choose the

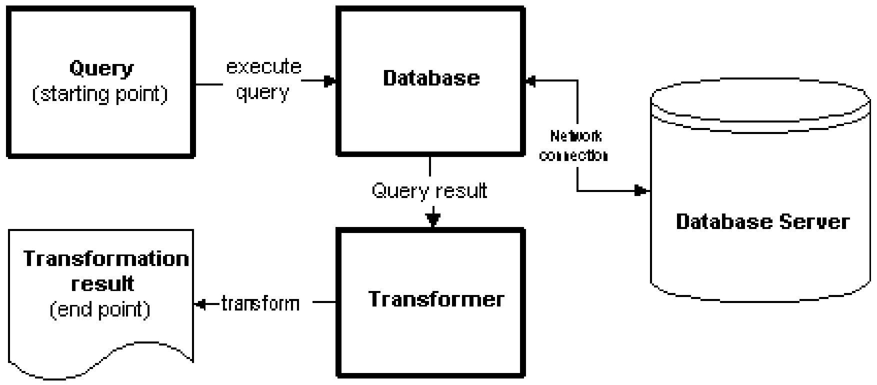


Figure B.1: Collaboration diagram of database access classes

appropriate language for your database application. Both alternatives have their own advantages and disadvantages. If you, for example, already own a set of XSLT stylesheets matching your needs you will prefer to use the combination of XPath and XSLT. Whereas it will be a good idea to choose XQuery, if you start a new project from scratch.

The following sections assume that you are familiar with the basics of *Tamino XQuery*. The documentation of *Tamino* [24] includes a comprehensive guide about it. Read it first, it will help you to understand this manual.

B.1.1 Getting started

This section describes the *Database access module*, which is part of the *Studierstube XML Database API*. The *Database access module* is a small set of C++ classes, providing access to a *Tamino* XML Database and for executing XSLT transformations. It consists of the three classes *Database*, *Query* and *Transformer*. The class *Database* manages a connection to an XML Database server, handling the transport of database queries and query results over a TCP/IP network. It also provides a method to execute queries in form of instances of the class *Query*, which holds a query script and other query related parameters. Query results can then be further processed with XSLT stylesheets using the *Transformer* class. See figure B.1 on page 90 for an illustration of how these classes collaborate. The discussed classes are marked by a bold frame.

The following sections will take you step by step through some typical basic operations you will probably want to do. They will teach you how to use the classes to connect to a database server, query and update XML documents, how to interpret database responses and how to process query results with XSLT stylesheets. Furthermore, a section that discusses advanced database topics is included. It is recommended to read this guide from the beginning to the end, so that you will have a complete overview after you have finished reading.

B.1.2 Sample database

The following sections rely on code examples, which work on a database containing information about staff meetings. Assume that we have created an XML Database "stuff" and a collection "meetings". An XML document of this collection stores information about our staff members' meetings. Further assume, that the database already contains the following document.

```
<meetings>
    <meeting number="1">
        <title>XML for Augmented Reality</title>
        <date>1/3/2004</date>
        <people>
            <person>
                <firstname>Edward</firstname>
                <lastname>Samson</lastname>
            </person>
            <person>
                <firstname>Ernestine</firstname>
                <lastname>Johnson</lastname>
            </person>
            <person>
                <firstname>Betty</firstname>
                <lastname>Richardson</lastname>
            </person>
        </people>
    </meeting>
    <meeting number="2">
        <title>XML databases</title>
            <date>31/5/2004</date>
            <people>
                <person>
                    <firstname>Ernestine</firstname>
                    <lastname>Johnson</lastname>
                </person>
                <person>
                    <firstname>Betty</firstname>
                    <lastname>Richardson</lastname>
                </person>
            </people>
        </meeting>
</meetings>
```

The root tag <meetings> of this sample document contains a list of the staff meetings, which have been held. Currently there are two meetings in the list. A meeting, enclosed by the tag <meeting>, is identified by a unique "number" attribute, stores a "title", the "date" when the meeting was and a list of "people" participating. For each "person" of the "people" list there are two child tags containing the persons first name and last name. This will be the basic data for the source code examples of the next sections.

B.1.3 Retrieving objects from the database

Using the database, which was defined in the previous section, this section shows how to open the database and how to run a simple query on it. For the ease of reading assume that all operations of the following code example succeed, so that we do not have to add any error handling code to it. Anyway, there is a special section about error handling later in this manual.

```
//1. include header files
#include <stbxml/Database.h>
#include <stbxml/Query.h>
#include <string>

//2. use namespaces
using namespace std;
using namespace stbxml;

//3. Define and open the database
Database database;
database.open("192.168.0.1", "stuff");

//4. Setup a Query object
Query query;
query.language = Query::XQUERY;
query.query = "input()/meetings/meeting/title";

//5. Run the query on the database
string response;
database.execute(query, "meetings", response)

//6. Print the result
printf("%s", response.c_str());

//7. Close the database
database.close();
```

(1.) First of all you need to include the correct header files. The ones related to the database API reside in the folder "stbxml", which should be in the search path of your compiler. This example uses the classes *Database*, *Query* and the standard C++ class *string* to store the response of the database server.

(2.) All classes of the database API are defined in the namespace "stbxml" in order to avoid name conflicts. It is a good idea to specify the namespace, in order to work with simpler class names.

(3.) These lines declare a Database object and open a connection to the database "stuff" on an XML server with the following IP address "192.168.0.1". By the way, the open method also supports the usage of a host name instead of an IP address. Keep in mind, that the current implementation of the open method does not check for the existence of the database or collection. Thus, take care of supplying correct names to it.

(4.) Since the *Query* class supports two query languages, namely XQuery and XPath, you have to specify which language should be used when defining the query. Here we use the language XQuery to specify a query command, which retrieves a list of all meeting titles contained in the database.

(5.-6.) Then we run the query on the database collection "meetings" by calling the method "execute()" of the database object. The response of the database server is stored in the string variable "response" and printed (6.) to standard output.

(7.) The final step closes the connection to the database.

Next, we will analyze the answer of the database server, which should contain the result set of the query. As you probably expect, the database answers in form of an XML document.

When all operations have been successfully executed, the output of the sample program should look like the following text.

```
<?xml version="1.0" encoding="UTF-8"?>

<stbxml:response
xmlns:stbxml="http://www.studierstube.org/xml/xmldb/response" ... >
  <stbxml:message>
     <ino:message ino:returnvalue="0">
        <ino:messageline>XQuery Request processing</ino:messageline>
     </ino:message>
     <ino:message ino:returnvalue="0">
        <ino:messageline>XQuery Request processed</ino:messageline>
     </ino:message>
    </stbxml:message>
    <stbxml:result>
      <title>XML for Augmented Reality</title>
```

```
        <title>XML databases</title>
    </stbxml:result>
   </stbxml:response>
```

Since the schema of an XML Database result has not been standardized until now and each database server uses a different format, we decided to define our own format in order to be independent of the manufacturer.

The root element ”stbxml:response” of the result document declares a namespace ”stbxml”, which is used to identify responses of the *Studierstube* database server. It is composed of the sub elements ”message” and ”result”, which are also declared in this namespace. These sub elements are mandatory and thus always present in a response document.

The ”message” part contains manufacturer specific information about the query, most notably error messages. You will find this information useful, when debugging your application.

Whereas the ”result” part holds the actual result set of the query, the data you are interested in. When using XQuery, the schema of this part will be the one which you defined in your query and is more or less manufacturer independent. You will need to extract the result set from the database response in order to be able to further process the information. The easiest way to achieve this is by using the class *Transformer*, which has not been discussed so far. It is shown in form of an example in the next section.

B.1.4 Transforming query results

In addition to the query operations, which were introduced in the previous section, the *Database access module* also offers ways to process information using XSLT stylesheets. There are two possibilities, which we will discuss separately: You can choose between client-side stylesheets and server-side stylesheets or even use both alternatives when needed. While the server uses the Java version of the *Xalan* transformer, the client transformation is implemented using the C++ version. Due to the fact that these are independent projects, the output of both alternatives sometimes shows minor differences when running the same stylesheet. Keep that in mind when you decide to change the side of transformation at a later point. You will have to adjust your stylesheet accordingly. But don't worry, this can be done with little effort.

Client side transformation

Client side transformation is supported by the class *Transformer*, which is part of the *Database access module*. *Transformer* is a wrapper class around the *Xalan* C++ library and tries to hide the quite flexible, but sometimes really complex, interface of *Xalan*. The following example uses this class to extract the result set from the database response of the previous section. For this purpose we need an appropriate stylesheet, which implements the extraction. The following stylesheet is copied from the *BAUMLBrowser* demo application and does exactly what we want.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:stbxml="http://www.studierstube.org/xml/xmldb/response">

<xsl:output method="xml" indent="yes" omit-xml-declaration="yes"/>

<xsl:template match="/">
    <xsl:copy-of select = "/stbxml:response/stbxml:result/*"/>
</xsl:template>

</xsl:stylesheet>
```

The stylesheet copies the content of the path "/stbxml:response/stbxml:result" to the output, when the root "/" of the input document is reached. Note that we had to declare the *Studierstube* namespace in line two to be able to access the right tags. Assuming that this stylesheet is stored in the file "getresult.xslt", the following code example shows how to apply the transformation to the server response of the previous section. It extends the previous source code sample by adding a transformation before the output is printed to the screen.

```
...
///added 1: we additionally need the Transformer header file
#include <transformer.h>

...

//5. Run the query on the database
string response;
database.execute(query, "meetings", response)

///added 2: Extract the query result
Transformer transformer;
transformer.applyStylesheet("getresult.xslt", response);

//6. Print the result
printf("%s", response.c_str());
...
```

After including the header file (added 1.), we add an instance of *Transformer* and call the method "applyStylesheet()" (added 2.) to transform the server response by applying the stylesheet file "getresult.xslt". The processed result set and thus the program output, will then look like the following text.

```
<title>XML for Augmented Reality</title>

<title>XML databases</title>
```

Server side transformation

Beside the possibility to run client side transformations, the database access module also supports the execution of stylesheets on the server. You will prefer this method, if you want to keep network bandwidth low or use some basic stylesheets in many different applications, for example. The downside of this method is, that the server load increases with the execution of each stylesheet and that the server's error messages are discarded before the result reaches the client. This might be an annoying fact, when it comes to debug your application. You will have to temporarily disable the execution of the stylesheet in your source code in order to find out what is wrong.

The transformation is specified by setting the parameter "stylesheet" of the class *Query*, which is used in the example of the previous section. This parameter is a URL identifying the location of the stylesheet. The stylesheet might be retrieved from file store, or directly from the XML server. If the value of the parameter is an absolute URL, the stylesheet is retrieved using that URL. If the value of the parameter is a relative URL, the stylesheet is retrieved from the server, as a named document relative to the collection of the current query. See the following examples:

Assume that the queries are run on the collection "meetings" in the database "staff". When the query of the first example is executed, it processes the result of this query using the stylesheet "getresult.xslt" found in collection "stylesheets" of the database "staff".

```
query.stylesheet="../stylesheets/getresult.xslt"
```

The second example obtains the stylesheet from the file store of the web server, not from the XML server. Here, the stylesheet "getresult.xslt" is located in the directory "stylesheets" of the local web server.

```
query.stylesheet="http://localhost/stylesheets/getresult.xslt"
```

Updating documents

The W3C consortium has already passed a standard for the XML query language XQuery. But the current version, 1.0, does not comprise any update operations. The *Tamino XQuery* language enhances this standard by supporting additional language constructs for inserting, updating, deleting and renaming parts of XML documents. All operations work on the node level, which means you are able to update all parts of a document, which can be addressed by an XPath expression, even at attribute level.

All update operations begin with the keyword update. The following example utilizes the update functions of XQuery to insert an additional person to the second meeting record of our sample collection "meetings". For this reason we substitute the query definition (4.) of our source example with a query that performs the update.

```
//4. Insert a new person
Query query;

query.language = Query::XQUERY;

query.query =
"update for $a in input()/meetings/meeting
 where $a/@number="2"
 do insert
 <person>
   <firstname>David</firstname>
   <lastname>Johnson</lastname>
   </person>
into $a/people";
```

As a result, the document will contain the additional element and the database server will respond with a document, which indicates whether the operation was successful or not. Apart from "into" there are two other keywords that you can use when inserting element nodes. Using "preceding" the element nodes will be inserted as preceding siblings to the update node. Using "following" the element nodes will be inserted as siblings following the update node.

The "delete", "update" and "rename" operations are specified using a similar language construct. The "For-Let-Where"-clause selects the nodes, which should be updated, and allows to define variables and the "do"-clause specifies how the data should be changed. Read the *Tamino* documentation [24] about XQuery for further details.

B.1.5 Advanced topics

This section deals with topics, which did not fit into one of the previous sections, while still being important enough to be mentioned when working with *Tamino* in the real world.

Limitations of Tamino XQuery

Though *Tamino XQuery* enhances the W3C standard with update operations, it lacks some of the features mentioned. The following list mentions the most important drawbacks and limitations when working with *Tamino*:

- XQuery does not support the unabbreviated syntax of location paths using named axes. For example, it is not possible to use the ancestor axis in XQuery. This should be changed in the next release of *Tamino*.
- Not all of the functions specified by the W3C consortium have been implemented yet. As a workaround they can be implemented using *Tamino* server extensions.
- It is not possible to insert or delete a document of a collection using XQuery. Which means that you have to use an external tool like *Tamino XPlorer* to insert and delete top level nodes.

The document identifier ino:id

Normally, a query is working on all documents contained in the current collection. Though sometimes it is necessary to restrict the scope of a query to one specific document. Since there is no standard way to achieve this, each manufacturer has implemented it's own solution. In *Tamino* each document gets a unique identifier, namely "ino:id", which can be used to refer to this document in a query. The following XQuery example shows how to retrieve the document with the ino:id "100". Note, that you have to specify the namespace for *Tamino* functions, because the function "getInoId()", which delivers the document ID of any node, is specific to *Tamino*.

```
declare namespace
tf="http://namespaces.softwareag.com/tamino/TaminoFunction";
for $a
in input()/*
where tf:getInoId($a)="100"
return $a
```

If the the "where"-clause was omitted in this query, it would change the result set to all documents of the collection instead of the document with ino:id "100".

Nested collections

Although this is a de facto standard of XML Databases, *Tamino* does not yet provide any support for nested collections. It means, that you are not able to divide a collection into further sub collections. If you are in need of utilizing sub collections for your project, you will have to map the tree structure of your design to the flat collection structure of *Tamino* by using an appropriate naming scheme.

Error handling

When working with the *Database access module*, you are utilizing a number of different components, each of which can fail to operate successfully. The error messages produced by these components, which contain details about the error reason, are delivered in quite different formats. Some components use plain text, others HTML and the third one XML in order to help the user solve the problem. To be able to subsume these different formats into one, the *Database access module* uses the most common of these, the text format. Most of the module's methods return a Boolean value indicating whether the operation was successful or not. In the case of a failure you can call the "getLastError()" method of the object to find out what was going wrong. The format of this message depends on the component, which caused the error.

As mentioned before, the result of a query operation can also contain an error message, even when the "Database::execute()" method returns true. The *Database* class does not analyze the server response, because it cannot distinguish between an error message and a valid database response. When using server side stylesheets one can produce any imaginable output, which also includes error messages. Thus, it is left to the user to decide, whether a server response is a valid query result or not.

B.2 SoXML DOM Model module

This section describes the *SoXML* DOM Model module, which is part of the *Studierstube Database API*. The *SoXML* DOM Model provides a simple document object model to represent and process XML data. When using this model you are able to access and update the content and structure of XML documents. It offers operations to convert the text representation of XML documents to a simple tree based object model and vice versa. The nodes of this tree model are made of objects, which are derived from *Coin* base classes, like *SoNode* and *SoGroup*. The parent-child relationship of the tree nodes is directly mapped to the model of *Coin* and implemented by utilizing the data structures of *SoGroup*. That way, XML data can be seamlessly integrated into a *Coin* scene graph and being processed by using standard *Coin* methods, like tree parsing and callback actions.

The advantage of deploying the *SoXML* DOM Model in your application is that it is easy to learn and use and its direct integration into *Coin*, while the drawbacks are that it does not support all kinds of XML tags, only provides simple methods to deal with namespaces and does not feature any data types. Probably the most useful application of the *SoXML* DOM Model is for reading and writing simply structured configuration data, that does not require sophisticated XML processing. The following sections show how you can utilize the classes of this module to read and write XML based configuration data and how to access and modify its content.

B.2.1 Getting started

The *SoXML* DOM Model module consists of three classes, namely the classes *Parser*, *SoXML* and *SoXMLText*. The *Parser* class can be seen as the active part of the module. It provides methods to parse and serialize XML data, where "parse" means to convert XML text to a *SoXML* DOM Model and "serialize" stands for generating text from a DOM tree. Parser uses the SAX interface of the C++ *Xerces* parser in order to convert an XML document to a DOM model, whereas the opposite way is accomplished by utilizing the tree traversing mechanisms of *Coin*.

When parsing an XML document, the parser class creates a tree, which is composed of instances of the classes *SoXML* and *SoXMLText*. As mentioned before, the model supports only a limited set of XML elements, or to be more precise, it supports XML tags, attributes and text data. Any other elements, like XML comments and processing instructions, though they are allowed to be present, are discarded during the parsing process.

An XML tag is represented by the class *SoXML*, which also stores the attributes of the tag and a list of its child elements. This child list can contain an arbitrary number of *SoXML* and *SoXMLText* objects in any order. Due to the fact that an XML document owns at least one tag, which is called root or document node, and that the *SoXML* DOM model does not provide any document class, the root node of a *SoXML* tree is always an instance of the class *SoXML*. Whereas the third class of this module, *SoXMLText*, which holds the text fragments of an XML document, is always a leaf node. Figure B.2 on page 101 illustrates the collaboration of aforementioned classes.

Following the solid arrows, the XML document on the left side of figure B.2 is fed to the Parser by calling its method "parse()", which creates the *SoXML* tree on the right side. The root object "root" of this tree contains two child objects. One *SoXMLText* object storing the text "Hello world !" and one *SoXML* object holding the content of the tag "element". The dotted line shows the opposite way, where the

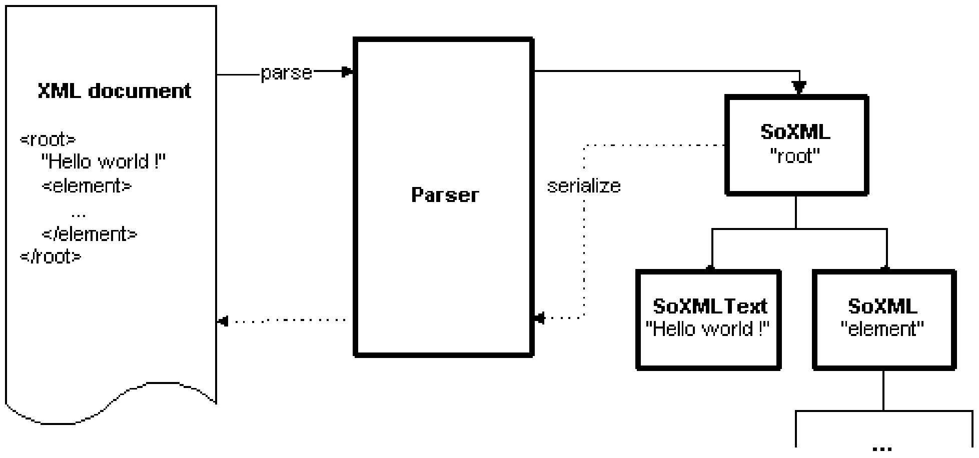


Figure B.2: Collaboration diagram of SoXML classes

SoXML tree is serialized by the parser and generates an XML document.

B.2.2 Parsing an XML document

This section shows how to parse an XML file and how to access the tags and attributes of the resulting *SoXML* tree. Imagine that we are writing an application, which uses a configuration file based on XML. These application settings should cover the language of the application and the background and foreground color of the main window. For this purpose we create an XML file with the following content.

```
<appSettings>
    <language>eng</language>
    <color bg="white" fg="black"/>
</appSettings>
```

The root element "appSettings" of this document embeds two child tags, which define the configuration data for our application. The tag "language" encloses a token, setting the application language to English and the tag "color" contains the attributes "bg" and "fg", which sets the background color to white and the foreground color to black. Assume, that this script is stored in a file named "configuration.xml". The following code example parses this file and prints the contained settings to the screen.

```
//1. Include header files
#include <stbxml/Parser.h>
#include<stbxml/SoXML.h>
#include <stbxml/SoXMLText.h>
```

```
//2. Use the DB API namespace
using namespace stbxml;

//3. Parse the configuration file
Parser parser;
SoXML* appSettings = parser.parseFile("configuration.xml");

//4. Get child tags
SoXML* language = appSettings->getChildByName("language");
SoXML* color = appSettings->getChildByName("color");

//5. Print the content of the tags
printf("language=%s\n", language->getText().getString());
printf("background=%s\n", color->getAttributeByName("bg").getString());
printf("foreground=%s\n", color->getAttributeByName("fg").getString());
```

(1.) The database API header files are located in the folder "stbxml", which should be included in the search path of your compiler. First of all we need to include the header files of the *SoXML* module.

(2.) In order to be able to use the *SoXML* classes without specifying any namespace prefix, we declare the namespace "stbxml".

(3.) Then we use the class *Parser* to parse the configuration file and create a *SoXML* DOM model by calling the "parseFile()" method of the class, which takes a file path as input parameter. Other input sources, like streams and URLs are also supported by this class. After successfully parsing, the tree's root node gets stored in the variable "appSettings". In the case of an error the parse method would return NULL to indicate that something was going wrong. You can then call the "getLastError()" method of the class to obtain further details about the error. Let us assume that the operation succeeds, though.

(4.) Starting from the root node, we access the child tags of the document by calling the method "getChildByName()", which is provided by the class *SoXML*. This is much more convenient than accessing the tags by using the member functions of the base class *SoGroup*. Knowing that Whitespace characters, like carriage return and space, also generate *SoXMLText* objects, which are stored in the child list, we cannot access the child tags simply by specifying a constant index. This means that the first object in the child list of the "appSettings" element in the example above will be an *SoXMLText* object containing "\n " and not the child tag "language", as one would probably expect. Thus, utilizing the "getChildByName()" method really makes sense.

(5.) Finally, the last part of the example prints the content of the tags to the screen. The language token is extracted by using the method ”getText()”, which concatenates all text objects contained in the tag ”language” and the background and foreground attributes of the ”color” tag are accessed with the help of the ”getAttributeByName()” method, which delivers the value of an attribute by supplying its name.

The output of the sample program should then look like this:

```
language=eng
background=white
foreground=black
```

B.2.3 Constructing an XML document

In contrast to the previous section, this section shows how to construct and serialize an XML document from scratch by utilizing the *SoXML* classes. First we will build a *SoXML* DOM model in memory, which represents the XML document, then we serialize the document using the class *Parser* and finally we write the result of this process to disk. This way, for instance, you are able to create a default configuration file solely from within your application. The following example constructs the XML document of the previous section and stores it to the file ”configuration.xml”.

```
//Include header files
#include <string>
#include <fstream>

...

//1. Create the root tag
SoXML* appSettings = new SoXML("appSettings");

//2. ref the root object
appSettings->ref();

//3. Add the language tag
SoXML* language = new SoXML("language");
language->addChild(new SoXMLText("eng"));
appSettings->addChild(language);

//4. Add the color tag
SoXML* color = new SoXML("color");
color->addAttribute("bg", "white");
color->addAttribute("fg", "black");
appSettings->addChild(color);

//5. Serialize the DOM tree using a parser object
```

```
    Parser parser;
    string xmlText = parser.serialize(appSettings);

    //6. Write the XML document to a file
    ofstream os("configuration.xml");
    os << xmlText;

    //7. unref the root object
    appSettings->unref();
```

(1.) After including all necessary header files and defining the namespaces to use, we create the root tag of the document. This is accomplished by using a special constructor of the *SoXML* class, which takes the name of the tag as parameter.

(2.) Since the *SoXML* class is derived from *SoGroup*, we also need to increment the reference counter in order to be able to access the object.

(3.) Then we create the child tag "language", add a text object to its child list holding the value of the tag and add it to the child list of the "appSettings" tag.

(4.) Next, the "color" tag is created and the attributes storing the background and foreground color are added to it by calling the method "addAttribute()", which takes the name and the value of the new attribute. Having done this, we have finished constructing the DOM model and are ready to convert it to text format.

(5.-6.) We instantiate a *Parser* object, call the "serialize()" method and pass the root object of the DOM model to it. The result of the conversion is buffered in the string variable "xmlText" and then written to the XML file "configuration.xml" (6.)

(7.) Finally, the reference counter of the root object needs to be decreased in order to delete it.

Appendix C

BAUMLBrowser manual

This manual describes the *BAUMLBrowser* application, a database browsing tool, which is used to browse and manipulate geometric information stored in an XML Database. BAUML, the "**B**uilding **AU**gmentation **M**arkup **L**anguage", is an XML language to store geometric information of buildings and parts of those buildings like walls, floors and corridors. The recursive definition of the language allows to create a tree structure of spatial objects, where objects are composed of a number of smaller objects.

The *BAUMLBrowser* application is capable of viewing and browsing such a BAUML tree and offers functions to insert, update and delete tree nodes. On the one hand it is an example application that demonstrates the usage of the *Studierstube XML Database API* and on the other hand it provides a reusable layer on top of this API to process BAUML documents. The following sections describe the user interface of the application and the reusable core component.

C.1 User Interface Guide

BAUMLBrowser is a dialog based application written with Microsoft Visual C utilizing the Microsoft Foundation Classes (MFC). The following section describes the meaning of the various control elements of this dialog and how you can use it to edit a BAUML document.

C.1.1 Tree control

Figure C.1 on page 106 shows a screenshot of the *BAUMLBrowser* application dialog. On the left side of the dialog you see a tree control displaying the structure of the BAUML objects which are stored in the database collection. Each node in the tree control matches a BAUML object in the database and each child node of a tree node corresponds to a child object of a BAUML object. Other element tags, like

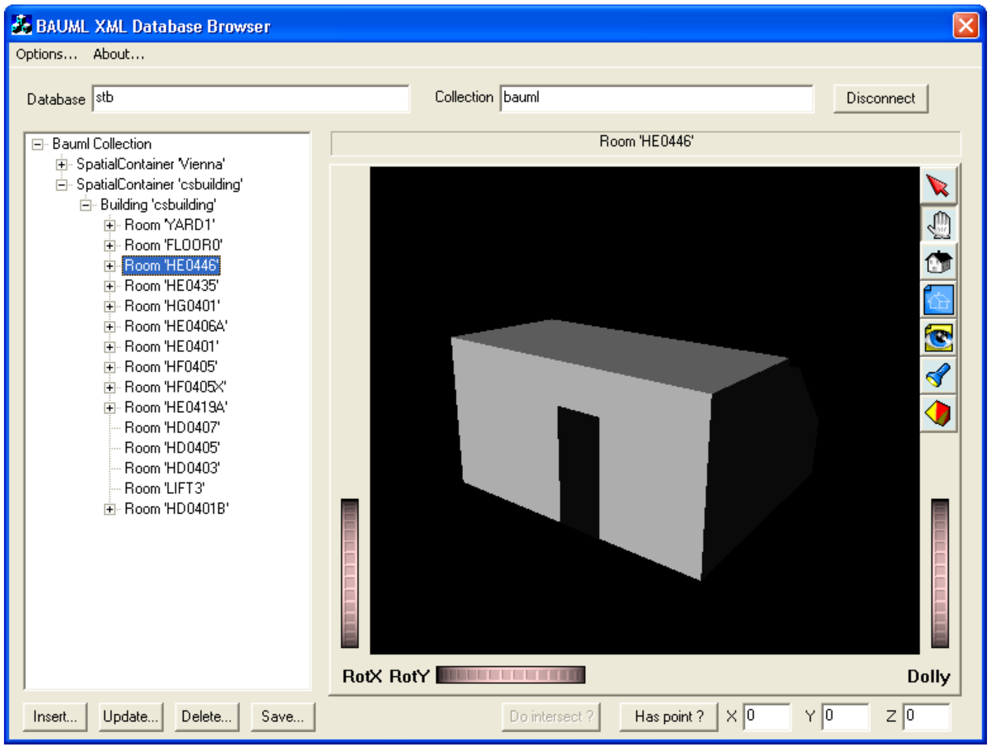
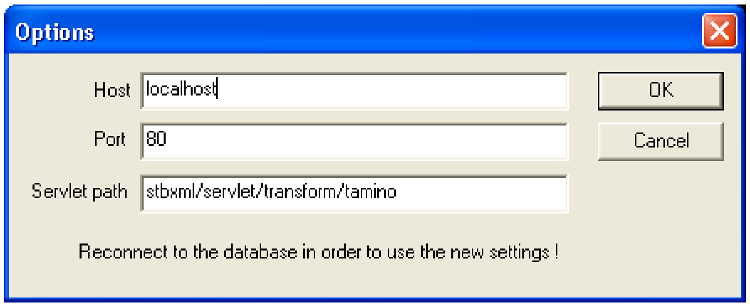


Figure C.1: BAUMLBrowser user interface

”representation” and ”pose” tags, which can also be part of a BAUML object are not shown by the browser. The name of such a tree node is composed of the object type name (i.e. the name of the XML element that represents the object) and the value of its ’id’ attribute, if available. The plus sign you can see on the left side of some of the tree nodes indicates that the corresponding BAUML object contains some child objects. Click on it with the left mouse button to open the list of child nodes.

C.1.2 Graphics window

Right to the tree control of figure C.1 you can see a window showing the graphical representation of a room object. An embedded *Coin* viewer is used to display the representation part of the BAUML objects. You can view an object simply by selecting it with the mouse in the tree control. The title bar on top of the viewer window then shows the name of the object viewed. If an object does not have a representation part the text ”no representation” is displayed instead. Consider that the representation of an object does not include the representation parts of its child objects.


Options

Host localhost

Port 80

Servlet path stbxml/servlet/transform/tamino

Reconnect to the database in order to use the new settings !

Figure C.2: BAUMLBrowser options dialog

C.1.3 Setting application options

The first thing you need to do, when working with the browser application is to set up the basic application options. Select the "Options" item from the program menu to open the dialog, which is depicted in figure C.2 on page 107.

Enter the correct host name or IP address of your XML Database server into the "Host" field. The "Port" and "Serlvet path" fields depend on the server installation and default to 80 for the port and "stbxml/servlet/transform/tamino" for the servlet path. Leave them unchanged if not told otherwise. Close the dialog by clicking on the "OK" Button. This saves the application options in the application configuration file. Consider that you need to close an open database connection and reconnect to apply the new settings.

C.1.4 Connecting to a database

The next step after configuration is to open a connection to an existing XML Database server. On top of the dialog in figure C.1 you see two edit fields named "Database" and "Collection". Enter the name of the database and collection you want to browse and press the button "Connect". After establishing a connection the application reads in the top level objects and the text of the "Connect" button changes to "Disconnect".

C.1.5 Inserting new objects

On the bottom left side of the application dialog in figure C.1 you find a number of buttons that offer operations to edit the current BAUML document. Located on the left is the button for inserting new objects. Press the button "Insert..." to open a file dialog which allows you to select a file to be inserted into the database. This operation supports two file types. You can either insert a BAUML object from an *.xml-file or from an *Coin* *.iv-script.

The first alternative creates a new BAUML object which is identical to the contents of the *.xml-file. Consider that this operation does not check the contents of the file. It is the responsibility of the user to ensure the validity of the inserted data.

The second alternative creates a new *SpatialContainer* object with a representation part that complies with the contents of the *.iv-file. Since the BAUML format represents all geometric data in form of polygons, Inventor shapes like cubes and spheres need to be converted to an appropriate form. With the exception of *IndexedFaceSet*, all shapes in the *.iv-file are transformed to a triangle representation using the triangle callback action of *Coin*. Whereas all polygons contained in an *IndexedFaceSet* shape are added to the representation as they are. During conversion any transformations defined in the *Coin* script are applied to the data. The newly created object is then inserted at the end of the child list of the currently selected object and the tree view is updated to reflect the changes.
Consider that is not possible to insert top level objects into the database with *BAUMLBrowser*. Use the *Tamino Xplorer* utility to accomplish that.

C.1.6 Updating objects

The update operation supports the same file types as the insert operation. Everything said about file validation and conversion applies to the update operation as well. Press the button "Update..." to open a file dialog and select an *.xml or *.iv file, which is used to update the currently selected BAUML object. Updating from an *.xml file replaces the selected object with the content of the *.xml file. Whereas updating from an *.iv file replaces only the representation part of the current BAUML object.

C.1.7 Deleting objects

The delete operation removes objects from the BAUML tree. Press the button "Delete..." to remove the currently selected object from the database. Consider that it is not possible to delete top level objects from the database with *BAUMLBrowser*. Use the *Tamino Xplorer* utility to do that.

C.1.8 Saving objects

The save operation stores a BAUML object or its representation to a file. Press the button "Save..." to open a file dialog and choose a directory and a filename. You can either save the currently selected object to an *.xml file in BAUML format or you can save only the representation part of the object as an Open Inventor scene graph to an *.iv file.

C.1.9 Object intersection test

The intersection operation tests whether the representation parts of two BAUML objects share a common region in space. In order to be able to use this function you need to select two objects from the tree view at first. Selecting two objects is done by clicking the first object with the left mouse button and clicking on a second object while holding the Ctrl-key. This enables the ”Do intersect?” button at the bottom of the graphics viewer in figure C.1. Press the button to start the test. The application then presents the result of the test in a message box showing either the string ”YES”, when an intersection was found, or ”NO”, otherwise. Keep in mind that this test is a bit limited due to the implementation of the *SoIntersectionDetectionAction* class of *Coin*. Which means that an intersection is only found, when the planes of the surface of two objects do intersect. So for example, if an object is completely contained in the other one, the intersection test will show a negative result.

C.1.10 ”Has point” operation

The ”Has Point?” operation tests whether a certain three-dimensional point is contained in the representation part of a BAUML object. On the bottom right of the application dialog you find three edit controls named ”X”, ”Y” and ”Z”. Here you enter the Cartesian world coordinates of your point to be tested. This function accepts floating point values if necessary. Start the test by pressing the ”Has point?” button on the left of the edit controls. The application then presents the test result in a message box showing either the string ”YES”, when the point is inside the object, or ”NO”, otherwise.

C.2 Core Component

This section describes the *BAUMLBrowser* C++ class, the core component of the *BAUMLBrowser* application. This class encapsulates the parts of the application which are independent of the Windows operating system and the Microsoft MFC class library. Thus it facilitates porting the Windows application to another operating system like Linux. The class basically defines a layer on top of the *Studierstube XML Database API*, which provides operations to deal with BAUML documents contained in an XML Database collection. This covers functions to insert, update and delete BAUML objects and their representation, as well as transformations of this representations to and from *Coin* scene graph scripts. In addition to that, two test functions have been implemented for checking object intersection and point position. The following sections show the basic ideas behind this component and illustrate how to utilize it to implement a database browser.

C.2.1 Basic concept

A BAUML document can be basically seen as a tree of BAUML objects. Each node of this tree is a target for manipulation through the user. Thus a mechanism is needed to uniquely identify each node. This is the job of the Node class. A Node object stores the location of a BAUML object and additional information like object type and name, which can be used to label nodes of a tree control. The Node class is a convenient way to work with BAUML objects, without having to deal with XML related implementation details. All database related methods make use of this class. When working with these methods you first get information about existing BAUML objects in form of Nodes and then use these Nodes to identify the BAUML object you want to update. Reading the entire content of a database at once would be quite time and memory consuming. Instead, only the information needed at the moment should be requested. As a tradeoff between reading all at once and getting the information node by node, *BAUMLBrowser* provides a method to get information about all child elements of a BAUML object. This way it is possible to traverse the BAUML tree beginning at the root node down to the leaf nodes, while keeping time and memory requirements within a reasonable limit. Moreover, this is exactly what you need to successively fill a tree control.

C.2.2 Class initialization

Before you are able to connect to a database, several initialization steps have to be performed. The following code example showcases the correct calling sequence of the methods used to setup a browser instance.

```
BAUMLBrowser browser;
...

//1. init the browser class
browser.init();

//2. set the Stylesheet directory to the current working directory
browser.setStylesheetDirectory(GetCurrentDirectory());

//3. verify that all stylesheet files exist
string msg;
if (!browser.verifyStylesheets(msg)) {
    printf(msg.c_str());
    exit(-1);
}

//4. load application options
BAUMLBrowser::Options options;
```

```
if (options.read())
  browser.setOptions(options);
```

(1.) First of all the *BAUMLBrowser* class needs to be initialized by calling the "init()" method. This initializes the third party libraries used by the browser like *Parser* and *Transformer*.

(2.) The next step is to set up the directory, where the stylesheet files reside. Normally, this is the directory path of the *BAUMLBrowser* executeable. Consider that you have to specify an absolute path here, so that the browser component is able to locate the stylesheets in the case the current working directory is changed during program execution.

(3.) Next, you should verify that all stylesheet files exist, which are needed by the browser. This is done by calling the method "verifyStylesheet()". In the case a stylesheet file is missing an error message is returned by the function, which can be reported to the user.

(4.) Finally, the fourth and last step loads application options from an XML configuration file. See the section about configuration for further details on application parameters.

C.2.3 Establishing a connection

After successful initialization, a connection to the database must be established, before you can start browsing it. At this point you need to know the name of the database and collection you want to edit. Use the "open()" method to accomplish this.

```
BAUMLBrowser browser;

...
browser.open("MyDatabase", "MyCollection");
...
```

Keep in mind that the current implementation of this function does not check for the existence of the database or collection. Thus, take care of supplying correct names to it.

C.2.4 Reading nodes from the database

This section shows how to gain information about BAUML objects contained in a database. As mentioned before, *BAUMLBrowser* uses the *Node* class to represent BAUML objects. A *Node* instance does not contain the XML object itself, but the information needed to build up a tree control illustrating the structure of the BAUML tree. Furthermore it holds location information, which is required to identify the associated BAUML object. The following code is a simplified version of the update function of the *BAUMLBrowser* application. It adds child items to an existing tree control item.

```
void CBAUMLBrowserDlg::AddChildItems(HTREEITEM hItem) {
    //1. get the associated node of the tree view item
    BAUMLBrowser::Node *node =
      (BAUMLBrowser::Node*)m_TreeCtrl.GetItemData(hItem);

    //2. get all child items of the node from the database
    BAUMLBrowser::NodeVector childList;
    m_Browser.getNodeChildren(node, childList);

    //3. for each node found in the database
    //insert a new treeview item
    for (int i=0; i<childList.size(); i++)
    {
        TVINSERTSTRUCT tvInsert;
        BAUMLBrowser::Node *childNode = childList[i];

        //the new items are children of the current item
        tvInsert.hParent = hItem;

        //4. set the name of the new item
        tvInsert.item.pszText = _T(childNode->getName().c_str());

        //5. add a plus sign, if the Node has children
        tvInsert.item.cChildren = childNode->hasChildren() ? 1 : 0;

        //store the Node object in the treeview item
        tvInsert.item.lParam = LPARAM(childNode);

        //insert the item into the treeview
        m_TreeCtrl.InsertItem(&tvInsert);
    }
}
```

(1.) Each tree view item is associated with a *Node* object stored in the "ItemData" field. The first line of the code example above gets this object by calling the "GetItemData()" method of the tree control. This is Windows specific code, but other operating systems will have similar methods to store user data in a tree control.

(2.) The next step reads out all child items of this node from the database into a variable of type *NodeVector*, which is a container for dynamically created *Node* objects.

(3.-5.) The resulting list of child nodes is then processed and for each child node a new item is added to the tree view. Properties like the name of the node (4.) and its parenthood (5.) determine the appearance and the state of the new tree view item. Using this function a tree control can be filled step by step with node information, starting at the root node (by passing NULL as the first parameter to "getNodeChildren()") up to the leave nodes of a BAUML document. The structure of the tree view items then reflects the parent-child relationship of these BAUML objects.

C.2.5 Getting object data

While the previous section showed how to gain structural information from the database, this section will teach you how to get data about specific objects. The following code example is taken from the *BAUMLBrowser* application. It is part of a function, which displays the representation part of a BAUML object.

```
//1. various objects used in this example
  // create a Coin viewer object
BAUMLBrowser browser; SoWinExaminerViewer* viewer = ...
  // select an existing node to be displayed
BAUMLBrowser::Node *node = ...

//2. get the representation part of the current node
string ivData = browser.getRepresentation(*node);

//3. setup an input buffer for the SoDB::readAll() function
SoInput in;
in.setBuffer((void*)ivData.c_str(), ivData.length());

//4. make a scenegraph from the text representation
SoSeparator* root = NULL;
root = SoDB::readAll(&in);

//5. show the scene graph in the viewer
viewer->setSceneGraph(root);

viewer->show();
```

(1.) In order to keep this example short assume that a *Coin* viewer object has already been created and the *BAUMLBrowser* node to be viewed has been selected, too.

(2.) The *BAUMLBrowser* class provides various methods for getting object data. Probably the most interesting of these is the "getRepresentation()" function. It reads the representation part of a BAUML object from the database and converts it to a *Coin* script.

(3.-4.) Using an input buffer the *Coin* script is delivered to the "readAll()" function (4.), which creates a *Coin* scene graph.

(5.) This scene graph can then be displayed by the viewer object.

C.2.6 Updating objects

This section introduces functions for inserting, updating and deleting whole BAUML objects and functions to insert and update their representation part. Inserting an object to a node means to append it to the child list of the corresponding object in the database, while update and delete operations process the object itself.

Keep in mind that data manipulation functions like "insert()" and "delete()" change the relative position of all sibling objects following the processed object. Thus, the position information of these siblings, which is stored in Node objects, becomes invalid. *BAUMLBrowser* corrects this simply by rereading the entire child list of the parent object. This has proven to be a very straightforward and reliable way to overcome this effect.

Furthermore, a restriction you should know of is that you cannot insert and delete top level objects using the *BAUMLBrowser* class. This is caused by limitations of the *Tamino XQuery* language, which is not able to act at this level. Instead, use *Tamino Xplorer* as a workaround for this task. The following example reads a BAUML object from a file on disk and adds it to the child list of an object in the database.

```
//1. Browser and node object
BAUMLBrowser browser;
  // select an existing node
BAUMLBrowser::Node *node = ...

//2. Path of the file to be inserted
string filePath = "NewBAUMLObject.xml";

//3. read the file into a string
string s = readFile(filePath);

//4. append the object to the child list of <node>
browser.insertObject(*node, s);
```

(1.) Analogue to the previous example, assume that a *Node* object has been selected to which you want to add a new child object.

(2.)(3.) Using the function "readFile()" the child object is loaded from disk and stored in a string variable.

(4.) Finally, the content of the file is added to the child list of the BAUML object by calling "insertObject()".

C.2.7 Special functions

Beside the standard operations for querying and updating the database, the *BAUMLBrowser* class provides two test functions for object intersection and point inclusion. The "intersect()" method tests if the representation part of two BAUML objects share a common region in space. It uses the *SoIntersectionDetectionAction* class of *Coin* to implement this functionality. Due to this, the test is a bit limited, meaning that an intersection is found, only when the planes of the surface of two objects do intersect. So, for example, if an object is completely contained in the other one, the intersection test will fail, although it should deliver a positive result.

The second operation "containsPoint()" tests, whether a certain three-dimensional point is contained in the representation part of a BAUML object. It does this by intersecting the object with a ray, which starts at the test point, and counting the intersection points. In case of an odd quantity of intersection points, the test point lies inside the object, otherwise outside. The following example tests, whether the three dimensional point (x=1,y=2,z=3) is inside an object and writes the result of the test to standard out.

```
//1. Browser and node object
BAUMLBrowser browser;
  // select an existing node
BAUMLBrowser::Node *node = ...

//2. Test if the point (1,2,3) is inside the object
if (browser.containsPoint(*node, 1.0, 2.0, 3.0))
  printf("Yes! The point is inside.");
else
  printf("No, the object does not contain the point.");
```

(1.) The first lines of this example instantiate a *BAUMLBrowser* class and select a *Node* object, which is then used in the test.

(2.) Then the representation part of the BAUML object, which is represented by the variable node, is tested wether it includes the point (1,2,3) by calling the browsers "containsPoint()" method. Finally, depending on the test result, an appropriate message is printed.

Appendix D

Installation and Configuration guide

This appendix lists the software needed and describes the necessary steps to install and configure an XML server for *Studierstube*. The installation procedure also includes the server side part of the *Studierstube XML Database API*. It is strongly recommended to use the same versions of the software packages as specified here, since these are the versions the software was developed and tested for.

D.1 Installation

D.1.1 Hardware prerequisites

The required hardware components are mainly determined by prerequisites of the *Tamino* XML server, which currently are:

- Intel Pentium III, at minimum 450 MHz
- 256 MB RAM minimum
- approximately 600 MB free disk space

For an up-to-date list of prerequisites see the *Tamino XML Starterkit* web page (www.xmlstarterkit.com).

D.1.2 Software prerequisites

The following software packages and documents are needed for a complete installation of the Studierstube XML database:

- Windows 2000 Professional and Server or Windows XP Professional

- Java 2 SDK Version 1.4.1_05
- *Apache Web Server* 2.0.43 (included in the Tamino setup)
- *Apache Tomcat* 4.1.29
- mod_jk_1.2.5_2.0.47.dll
- *Tamino Passthru Servlet*
- *Studierstube Passthru Servlet*
- Server side stylesheet

D.1.3 Installation procedure

Install a compatible Operating System

The current version of *Tamino* XML server supports the following Windows versions:

- Windows 2000 Professional and Server
- Windows XP Professional

Hint: With Windows 2000 you will need to update Internet Explorer to a recent version (6.0)

Install Java

Get Java 2 SDK Version 1.4.1_05 by downloading it from Sun's Java developer web page (java.sun.com). Install it to the default installation directory C:\j2sdk1.4.1_05 using the default settings.

Hint: This SDK already includes the Java versions of Apache Xerces and Xalan, therefore these components need not to be installed separately.

Install Apache Web server

Install Apache Web Server 2.0.43. This web server version should be included with *Tamino*. In the case it is not included, you can download it from the *Apache Web Server* page (www.apache.org). Choose "Typical installation" from the menu and follow the installation instructions.

Install Tamino XML Server

Install *Tamino* XML Server 4.1.4. Choose "Complete" setup from the menu and use the default settings. You should have an installation CD for the *Tamino* software.

Otherwise you can get a 30-day trial version at the *XML Starterkit* page of Software AG (www.xmlstarterkit.com). You will also need to register to get a valid license file via e-mail.

Hint: You will need to reboot the system after installation.

Install Apache Tomcat

Install *Apache Tomcat* 4.1.29. This servlet container is needed to run the software, which executes the server side XSLT transformation. Download it from the *Apache Jakarta* Project page (jakarta.apache.org). During installation you will be asked to point to a directory containing a Java SDK, choose the one you have installed before and install *Tomcat* as "NT Service"

Install MOD_JK DLL

Install mod_jk_1.2.5_2.0.47.dll - This DLL is needed to connect *Tomcat* and the *Apache Web Server*. Get it from the *Apache* web page (www.apache.org) and copy it to C:\Program Files\Apache Group\Apache2\modules

Add the Passthru Servlets

Copy the contents of the *Tomcat* "webapps" directory from *Studierstube Subversion* server to "C:\Program Files\Apache Group\Tomcat 4.1\webapps". This directory contains the *Tamino Passthru Servlet* and the *Studierstube Passthru Servlet*.

Configure Tomcat

Copy the files from the *Tomcat* configuration (from *Studierstube Subversion* server) directory to "C:\Program Files\Apache Group\Tomcat 4.1\conf" and adjust the path settings as necessary.

Add the server side stylesheet

Copy the contents of the *Apache* "htdocs\stylesheets" (from *Studierstube Subversion* server) directory to "C:\Program Files\Apache Group\Apache2\htdocs\stylesheets".

Configure Apache Webserver

Add the line

```
Include "C:/Program Files/Apache Group/Tomcat4.1
/conf/apache-connector.conf"
```

to the Apache configuration file at
"C:\Program Files\Apache Group\Apache2\conf\httpd.conf"

D.2 Creating databases and collections

This section is a quick guide to database and collection management. It is meant solely as a starting point for working with *Tamino* databases. For a comprehensive description of this topic read the documentation that comes with *Tamino*.

D.2.1 Creating a database

Probably the first thing you want to do after installation is to set up a database. This is accomplished by using the web application *Tamino Manager*. After login by providing an administrator's user name and password you see a list of managed hosts at the top left of the Explorer window. Normally, this list will contain only one item, the name of your database server. Click on it and go to "Tamino/Databases". At the bottom left side of the Explorer window you will find a button labeled "Create database". Click on it and provide a name for your database. Push the button "Finish" to create the database.

In order to be able to access a database from a client application, the database has to be started first. When selecting your database from the database list, you will find a button labeled "Startup database". A click on it will start the server process, which manages your database.

Hint: You can also automate the startup process at server boot time by editing the server properties of your database and setting the parameter "autostart" to "yes".

D.2.2 Creating a collection

Since XML Databases are organized in form of collections, you will need to create a collection, too. Start the *Tamino X-plorer* application and login to your server by clicking on the server name with the right mouse button and select "connect" from the local menu. Provide an administrator's user name and password. (The first time logging in you will also need to provide the name of your database.) Right click at your database in the tree window and select ”New Collection”. Provide a name for your collection and push ”create”.

D.2.3 Providing a database scheme

In order to work with your database collection you will also need to define a schema of the XML documents it should contain. You do this by right clicking your database in *Tamino X-plorer* and selecting ”Define Schema”. As a starting point we have provided a simple schema here, which defines an element named ”MyElement”, which is able to hold arbitrary content. You will need to rename the string ”MyElement” according to the name of your XML root tag.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs =
"http://www.w3.org/2001/XMLSchema" xmlns:tsd =
"http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
  <xs:annotation>
     <xs:appinfo>
        <tsd:schemaInfo name = "MyElement">
           <tsd:collection name = "MyCollection"/>
              <tsd:doctype name = "MyElement">
                 <tsd:logical>
                    <tsd:content>closed</tsd:content>
                  </tsd:logical>
              </tsd:doctype>
           </tsd:collection>
        </tsd:schemaInfo>
      </xs:appinfo>
  </xs:annotation>
  <xs:element name = "MyElement"/>
</xs:schema>
```

D.2.4 Inserting documents

Since root level documents can not be inserted using the ”Studierstube XML Database API”, you need *Tamino X-plorer* to accomplish this. After login right click on your database in the tree window and select ”Insert instance” from the local menu. This will open up a dialog, which let’s you point to an XML document for insertion.

D.3 Configuring the fixed stylesheet

The modified version of the *Tamino Passthru servlet*, the *Studierstube Passthru Servlet*, offers a feature to apply a server-side stylesheet to each query result. It is meant to translate *Tamino* namespaces to namespaces, which are independent from the manufacturer. The name of the stylesheet used by the servlet can be configured by editing the "web.xml" configuration file, which is located in the "webapps/stbxml/WEB-INF" directory of *Tomcat*. The "servlet" tag of this XML file contains a list of initialization parameters, which are passed to the servlet at startup time. Assuming that the stylesheet is located in the web directory /stylesheets, the parameter, which points to the fixed stylesheet would look like the following example:

```
<init-param>
  <param-name>fixedStylesheet</param-name>
  <param-value>
    http://localhost/stylesheets/stbresult.xsl
  </param-value>
</init-param>
```

The tag "param-name" is the name of the parameter, which we have added, and must be set to "fixedStylesheet". The tag "param-value" contains an URL pointing to the stylesheet. In the case of an absolute URL, this URL is used to retrieve the stylesheet. In the case of a relative URL, for example "/stylehsheet/stbresult.xsl" the stylesheet is loaded directly from the *Tamino* database, where the first part of the path refers to the collection containing the stylesheet. In this example the name of the collection would be "stylesheets".

List of Figures

Bibliography

[1] Rick Kazman, Leonard J. Bass, Mike Webb, and Gregory D. Abowd. "SAAM: A method for analyzing the properties of software architectures". In International Conference on Software Engineering, pages 81–90. ICSE, 1994.

[2] Karl Crary, Robert Harper, Peter Lee, Frank Pfenning. "Modularity Matters Most". Carnegie Mellon University, Pittsburgh. October 31, 2001.

[3] Nigel Bevan, Jurek Kirakowski and Jonathan Maissel. "What is Usability?". In Proceedings of the 4th International Conference on HCI, Stuttgart, September 1991

[4] Steven Clarke. "Measuring API Usability". Dr. Dobb's Journal Special Windows/.NET Supplement, May 2004

[5] Gerhard Reitmayr. "On Software Design for Augmented Reality". Dissertation at Technical University of Vienna. 2004.

[6] Helmut Erlenkötter. "XML Extensible Markup Language von Anfang an". Rowohlt Taschenbuch Verlag. September 2003. ISBN 3 499 61209

[7] David Gulbransen, et al."Using XML". Second Edition, Que Publishing June 2002, ISBN 0-7897-2748-x

[8] W3C Consortium. "Extensible Markup Language (XML)". http://www.w3.org/XML, 1996-2003

[9] W3Schools. "XPath Tutorial". http://www.w3schools.com/xpath

[10] Miloslav Nic, Jiri Jirat. "XPath Tutorial". http://www.zvon.org/xxl/XPathTutorial/General/examples.html. 2000

[11] W3Schools. "XSLT Tutorial". http://www.w3schools.com/xsl

[12] W3Schools. "XQuery Tutorial" http://www.w3schools.com/xquery

[13] W3Consortium. "XQuery 1.0. An XML Query language".

http://www.w3.org/TR/xquery

[14] W3Consortium. "XML Syntax for XQuery 1.0. (XQueryX)". http://www.w3.org/TR/xqueryx

[15] W3Consortium. "XML Schema". http://www.w3c.org/XML/Schema

[16] W3Schools. "XML Schema Tutorial". http://www.w3schools.com/schema

[17] Roger L. Costello. "XML Schema Tutorial". http://www.xfront.com/xml-schema.html. 2001

[18] W3Consortium. "Document Object Model (DOM)". http://www.w3c.org/DOM

[19] Official website for SAX. http://www.saxproject.org/

[20] Sun Microsystems. "Java API for XML Processing". http://java.sun.com/xml/jaxp

[21] Open Source Project. "JDOM- Java Document Object Model". http://www.jdom.org/

[22] Open Source Project. "DOM4J - Document Object Model for Java". http://dom4j.org/

[23] Ronald Bourret. "XML Data Binding Resources". http://www.rpbourret.com/xml/XMLDataBinding.htm. 2001-2004

[24] Software AG. "Tamino XML Database Documentation". Tamino Version 4.1.4.1. 2004

[25] Ed Ort, Bhakti Mehta. "Java Architecture for XML Binding (JAXB)". http://java.sun.com/dev
March 2003

[26] INCITS H2 and ISO/IEC JTC1/SC32/WG3 standards groups. "SQLX - SQL & XML Working Together". http://www.sqlx.org/

[27] Ronald Bourret. "XML and Databases". http://www.rpbourret.com/xml/XMLAndDatabases.htm. July, 2003

[28] Microsoft Corp. MFC Reference Library. "ODBC classes". http://msdn.microsoft.com/library/en-us/vclib/html/_mfc_odbc_classes.asp

[29] Borland C++ Builder 6 User manual.

[30] XML:DB Initiative. "Application Programming Interface for XML Databases". http://xmldb-org.sourceforge.net/xapi/. 2000-2003.

[31] Kimbro Staken. "An Introduction to the XML:DB API".
http://www.xml.com/pub/a/2002/01/09/xmldb_api.html. January 09, 2002.

[32] Hauke von Bremen. "XinCJ - Xinidice C++ Database API".
http://www.codexperts.com/download.html

[33] Software AG. Websites for the XML Database Tamino.
http://www.softwareag.com/tamino
http://www.xmlstarterkit.com

[34] Apache group. Website for the XML Database Xindice.
http://xml.apache.org/xindice/

[35] Wolfgang Meier. Website for the XML Database eXist.
http://exist-db.org

[36] James Bates, Kevin O'Neill. "Xindice internals"
http://xml.apache.org/xindice/dev/guide-internals.html

[37] Hendrik Seffler. "XML Datenbank Exist". http://www.informatik.hu-berlin.de/ seffler/2-ausarbeitung.pdf. HU Berlin.

[38] Wolfgang Meier. "eXist: An Open Source Native XML Database". http://exist-db.org/webdb.pdf. Darmstadt University of Technology.