

# A Framework for Ontology Based Queries in a Semistructured Database for World Heritage

Chris Poulsen  
Martin R. N. Christensen

Kgs. Lyngby 2003  
IMM-THESIS-2003-53



# A Framework for Ontology Based Queries in a Semistructured Database for World Heritage

Chris Poulsen  
Martin R. N. Christensen

Kgs. Lyngby 2003

Technical University of Denmark  
Informatics and Mathematical Modelling  
Building 321, DK-2800 Lyngby, Denmark  
Phone +45 45253351, Fax +45 45882673  
[reception@imm.dtu.dk](mailto:reception@imm.dtu.dk)  
[www.imm.dtu.dk](http://www.imm.dtu.dk)

IMM-THESIS: ISSN 1601-233X

## **Abstract**

World Heritage (WH) is an organization which aims at preserving particularly interesting areas, monuments etc. Each of these “sites” are described on a website.

In order to help users navigate the existing World Heritage website, some categorizations have been created. For instance it is possible to browse categories based on location or site type.

It is difficult to make good categorizations and take advantage of the possibilities that they offer. But good categorizations expresses a lot of information about the sites that they cover. Categorizations can be used to make some complex queries. For example it is possible to suggest sites that are related to each other based on some category property.

The goal of this project is to explore the possibilities that emerging XML technologies offer, and based on the technologies suggest a way of making categorizations of semistructured data. Furthermore we explore the possibilities that categorizations of semistructured data offer, and create a framework that supports easy generation of categorizations. We explore how queries can take advantage of categorizations and how query results can be presented to the users on the WH website in a usable manner.

The WH site list contains many different sites, and many of them do not have much in common. This makes it hard to describe all the sites using the same schema. To avoid this problem we use a semistructured data model, and implement a software system that illustrates some of the different principles that applies to semistructured data. The implementation is based on Open Source Software and XML specifications from the World Wide Web Consortium such as XQuery and XPath.

Keywords: ontology, classification, XQuery, XML, World Heritage, semistructured data



## Resumé

World Heritage (WH) er en organisation, hvis mål er at bevare specielt interessante områder, monumenter mm. Hver af disse "lokaliteter" er beskrevet på en webside.

For at hjælpe brugere med at finde rundt i den eksisterende World Heritage webside, er der lavet nogle kategoriseringer. For eksempel er det muligt at "browse" kategorier baseret på beliggenhed eller type.

Det kan være svært at lave gode kategoriseringer og udnytte de muligheder, som de giver. Men gode kategoriseringer udtrykker en masse information, omkring de lokaliteter de kategoriserer. Kategoriseringer kan anvendes til at konstruere komplekse forespørgsler. For eksempel er det muligt at lave forespørgsler, som foreslår andre lokaliteter, der er relaterede til en valgt lokalitet. Relationen mellem den valgte lokalitet og de relaterede lokaliteter, er indeholdt i kategoriseringerne.

Formålet med dette projekt er at udforske de muligheder, som nye XML teknologier tilbyder, samt foreslå, hvordan disse kan benyttes til kategorisering af semistruktureret data. Desuden undersøger vi de anvendelsesmuligheder som kategoriseringer tilbyder, og laver et "framework", der kan benyttes til at lave kategoriseringer. Vi undersøger, hvordan forespørgsler kan udnytte kategoriseringer til at lave gode søgefaciliteter, samt hvordan søgeresultater kan præsenteres for besøgende på World Heritage websiden.

Listen af lokaliteter under World Heritage indeholder mange forskelligartede lokaliteter, og mange af dem har kun få ting til fælles. Dette gør det problematisk at beskrive alle lokaliteter vha. et fælles skema. For at undgå disse problemer benytter vi en semistruktureret datamodel, og implementerer et softwaresystem, som viser de forskellige principper omkring anvendelse af semistruktureret data.

Implementationen er baseret på open source software og XML specifikationer fra *World Wide Web Consortium* eksempelvis XPath og XQuery.

Nøgleord: ontologi, klassifikation, XQuery, XML, World Heritage, semistruktureret data





# Preface

Everybody was talking about XML a few years ago, switching to the simple textual data format was a giant leap forward for integration of different systems. Instead of using proprietary data formats XML made interfacing between all sorts of systems much easier. The XML format has been widely adopted and a lot of exciting technologies have started to spawn around it. One XML query language - XPath is reasonably mature now and another more complex query language XQuery is in the works.

Since XML has proven itself as being a good choice for certain applications, a demand for database systems that can handle and query XML is rising. Some database management system vendors have already made more or less complete solutions for storing and working with XML.

One of the two goals of this project is to take some of the new XML technologies for a test drive, in order to see how usable and mature some of the open source implementations are. One of the greatest strengths of XML is its flexible nature, it is a very good tool for representing semistructured data and data with hierarchical structure.

Having attended a course in knowledge based systems, where some of the problems with representing complex data in a way that computers can handle, was being reviewed using a UNESCO project called World Heritage as case study. We thought that it would be exciting to see if XML, and the new XML technologies, could be used to solve some of the problems that exist in that domain.

The other goal of this project is to explore the possibilities of querying semistructured data represented in XML. Querying semistructured data raises a few interesting issues, and we would like to see if it is possible to enable ordinary web users to query the data without being exposed to the complexity.

The readers of this report should be familiar with Java, Java2 Enterprise Edition (J2EE) , UML and XML as the applications developed in this project relies on these technologies / notations.

The best way to read this report, is to read the introduction in chapter 1 in order to get some basic understanding of the domain and the problems in it. Then go to the website, developed as a part of this project, and try out a couple of queries,

in order to get a little “feel” for the system. The “Advanced Search” page is the place to visit, as it contains the interesting functionality. The website is located at: <http://csdbs.it.dtu.dk/whapp> - a server that Hans Bruun, IMM has been kind enough to put to our disposal for the demonstration of the software developed in this project. The webpage contains links to everything created during this project; source code, documentation and applications.

Readers who wants to try the “ClassificationDesigner” can also connect to [csdbs.it.dtu.dk](http://csdbs.it.dtu.dk) at port 1099. A userguide for the ClassificationDesigner is found in appendix I on page 151. Username for the server is: `whuser` and the password is: `fraggel`.

After having tried the dataguide-based search, the rest of the chapters should be read subsequently. Chapter two covers the theory that constitutes the base for the project. Chapter three contains the modeling and design of the software systems. Chapter four describes the implementation of the software systems. Chapter five is a summary of the report, it discusses the solution created, possible extensions and the technologies used.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	What is World Heritage . . . . .	1
1.2	Detailed Problem Description . . . . .	2
<b>2</b>	<b>Theory</b>	<b>7</b>
2.1	Semistructured Data Models . . . . .	7
2.1.1	The eXtensible Markup Language – XML . . . . .	12
2.1.2	The XML Format . . . . .	13
2.1.3	A Semistructured WH Site . . . . .	16
2.1.4	XML Data Models . . . . .	18
2.2	Schemas for Semistructured Data . . . . .	20
2.2.1	Schema formalisms . . . . .	21
2.2.2	Obtaining a schema . . . . .	25
2.3	Querying SSD . . . . .	26
2.3.1	Path expressions . . . . .	27
2.3.2	The Generic Query Language . . . . .	28
2.3.3	XPath . . . . .	29
2.3.4	XQuery . . . . .	30
2.4	World Heritage Classifications . . . . .	32
2.4.1	Partial Orders and Lattices . . . . .	33
2.4.2	Classifications as Lattices . . . . .	34
2.4.3	Representing Ontologies . . . . .	36
2.4.4	Taking Advantage of Ontologies . . . . .	37

---

2.5	Querying the Classifications . . . . .	39
2.5.1	Searching Marked Categories . . . . .	39
2.5.2	Finding Related Sites . . . . .	40
2.5.3	Finding the Best Match . . . . .	42
2.5.4	Presenting the Query Results . . . . .	42
2.6	Summary . . . . .	43
<b>3</b>	<b>Application Modeling and Design</b>	<b>45</b>
3.1	System Description . . . . .	45
3.2	Specification of the XML Documents . . . . .	47
3.2.1	Modeling Classifications . . . . .	47
3.2.2	Modeling the Site Document . . . . .	49
3.2.3	Generating XML Data from Existing Data . . . . .	53
3.2.4	The Connection from Classification to Data Document . . . . .	54
3.3	WH System Model . . . . .	55
3.3.1	Actors . . . . .	55
3.3.2	Use Cases . . . . .	55
3.4	WH System Design . . . . .	58
3.4.1	The Model View Controller design pattern . . . . .	59
3.4.2	Design of the WH System . . . . .	60
3.4.3	Introducing XQueries in the Application . . . . .	63
3.4.4	Database Design . . . . .	63
3.5	WH System Class Specification . . . . .	64
3.5.1	JavaBeans and JSP Pages . . . . .	64
3.5.2	Front Controller Servlet . . . . .	65
3.5.3	Request Handlers . . . . .	65
3.5.4	Enterprise JavaBeans . . . . .	66
3.6	ClassificationDesigner Model . . . . .	67
3.6.1	Actors . . . . .	68
3.6.2	Use Cases . . . . .	68
3.7	ClassificationDesigner Design . . . . .	70
3.8	Summary . . . . .	72

---

<b>4</b>	<b>Implementation and Testing</b>	<b>75</b>
4.1	Choice of Software . . . . .	75
4.1.1	Choice of Software for the WH System . . . . .	76
4.1.2	Choice of Software for the ClassificationDesigner . . . . .	78
4.2	Implementation of the WH System . . . . .	78
4.2.1	Overview of Components in the WH System . . . . .	79
4.2.2	Implementing the Web Archive . . . . .	79
4.2.3	Implementing the EJB Component . . . . .	82
4.2.4	Performance Considerations . . . . .	84
4.2.5	The XQueries . . . . .	86
4.2.6	General Notes About Query Results . . . . .	88
4.3	Implementation of the ClassificationDesigner . . . . .	89
4.3.1	Important Notes about the ClassificationDesigner . . . . .	89
4.3.2	Class Description for the ClassificationDesigner . . . . .	91
4.4	Tests . . . . .	102
4.4.1	Tests of the WH System . . . . .	102
4.4.2	Test of ClassificationDesigner . . . . .	102
<b>5</b>	<b>Discussion</b>	<b>103</b>
5.1	Improvements to the WH System . . . . .	103
5.2	Reuse of WH System Components . . . . .	104
5.3	Improvements to the ClassificationDesigner . . . . .	104
5.3.1	Relational Back End . . . . .	104
5.3.2	Editing of Data Documents . . . . .	104
5.3.3	Better List Handling . . . . .	105
<b>6</b>	<b>Conclusion</b>	<b>107</b>
<b>A</b>	<b>XML Document – CD Catalog</b>	<b>111</b>
<b>B</b>	<b>Schema for the CD Catalog</b>	<b>119</b>

---

<b>C</b>	<b>Deployment</b>	<b>121</b>
C.1	Deployment of the WH System . . . . .	121
C.2	Deployment of the ClassificationDesigner . . . . .	122
<b>D</b>	<b>Screen Shots of the WH Web Application</b>	<b>123</b>
D.1	Welcome Page . . . . .	123
D.2	About Page . . . . .	124
D.3	Search for Categories . . . . .	125
D.4	Result of Search for Categories . . . . .	126
D.5	Advanced Search Form . . . . .	127
D.6	Advanced Search Result as Dataguide . . . . .	128
D.7	Advanced Search Result as List . . . . .	129
D.8	Site Information about Kronborg Castle . . . . .	130
D.9	Site Information about Roskilde Cathedral . . . . .	131
D.10	“Similar Sites” to Roskilde Cathedral . . . . .	132
D.11	Simple Search Form, Showing Help Info . . . . .	133
D.12	Result of a Simple Search . . . . .	134
D.13	Error Page . . . . .	135
<b>E</b>	<b>EJB Classes from the WH System</b>	<b>137</b>
<b>F</b>	<b>XPath Performance Tests</b>	<b>139</b>
F.1	Test Programs for eXist . . . . .	139
F.2	Test Programs for Saxon . . . . .	140
<b>G</b>	<b>Classification Schema</b>	<b>143</b>
<b>H</b>	<b>Database Schema for the Current WH Relational Database</b>	<b>145</b>
<b>I</b>	<b>User Manual for ClassificationDesigner</b>	<b>151</b>
I.1	Installation . . . . .	151
I.2	Making a Sample Classification . . . . .	151
<b>J</b>	<b>XML XQuery Document</b>	<b>163</b>
	<b>Glossary</b>	<b>174</b>

# Chapter 1

## Introduction

*The first section in this chapter describes what “World Heritage” is all about and who started the World Heritage project.*

*The second section describes the problems with the current World Heritage website, and explains how the website could be improved. This section should be read really carefully, because some important terms are introduced.*

### 1.1 What is World Heritage

*World Heritage* is a convention started by the organization UNESCO. According to the official UNESCO website their objective is the following:

The main objective of UNESCO is to contribute to peace and security in the world by promoting collaboration among nations through education, science, culture and communication in order to further universal respect for justice, for the rule of law and for the human rights and fundamental freedoms which are affirmed for the people of the world, without distinction of race, sex, language or religion, by the Charter of the United Nations.

So one might say that the purpose of UNESCO is “to make the world a better place for all of us”. The World Heritage convention was started by UNESCO in order to ensure protection of the world’s natural and cultural heritage. When a country signs the convention, the government of that country agrees, that it will try to preserve the natural and cultural heritage in that country. The World Heritage list is a list of sites which have such special properties, that they should be preserved for the future generations.

Currently World Heritage has a website containing all sorts of information about the convention and the approximately 730 different sites inscribed in the WH list.

The purpose of the website must be to inform as many people as possible about the WH convention. Unfortunately the website is not very user friendly, hence people visiting their website probably often leaves the website after a quick visit, without actually digging deeper into World Heritage.

Notice that the people responsible for the WH website has realized the problem, and has started working on a prototype website with improved useability.

## 1.2 Detailed Problem Description

This project is using the prototype of the World Heritage website as a case study [[vr-heritage prototype](#)], whenever the “WH site” is mentioned, this is the website that is being referred to. The WH site offers visitors two different ways of locating interesting sites:

1. Navigation to interesting sites based on lists.
2. Search by keywords.

The figure on the right is a manipulated screen shot from the WH site. Some of the items in “Search by Theme”-list have been removed and the “Search by Keyword” construct has been moved to the bottom.

As the screen shot shows, the “Search” page allows the user to select a list of sites based on theme, region or country. If the user clicks on a theme like “Fossil Sites” the system will generate a list of relevant sites and the user can click the links in the list to navigate to the interesting sites.

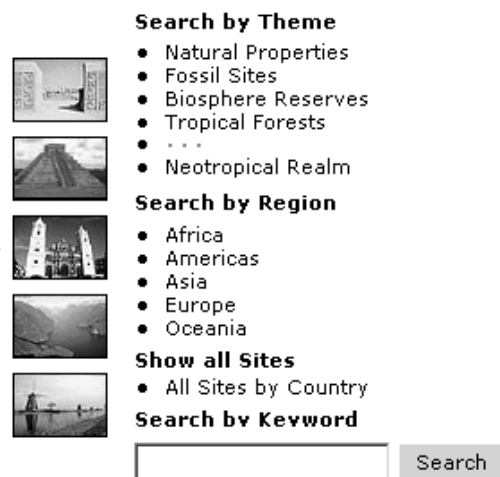


Figure 1.1: Screen shot from the WH site

If the user wishes to search by keyword instead of navigating the lists, the “Search by Keyword” box should of course be used.

The “Search by Region” result list is a bit interesting, the result lists are actually specializations of the “All Sites by Country”-list, for instance if the region “Europe” is chosen, the result is a list with all the sites in Europe sorted by country. This indicates that “Region” is related to “Country” in some way. This relation is of course trivial, but it is important to be aware that it exists.

If the WH Group decided to refine the region search even further, they could split each region into more precise sub-regions. For instance could “Europe” contain



four sub-regions: “Eastern Europe”, “Northern Europe”, “Southern Europe” and “Western Europe”.

The search could be made in the same way as the “Search by Region” search, filling in another category called “Search by Sub-region” containing the four entries mentioned above, as well as the entries that sub-regions will introduce for the other regions. These “extra” lists could be helpful for a visitor looking for sites in e.g. Eastern Europe.

However this is not an optimal path to follow. The “Search” page would grow a lot and become even less intuitive to navigate for the mundane user.

Since the “refinements” actually are specializations of the “All Sites by Country”, it is possible to build a hierarchy based on the specializations of “All Sites by Country”. Hierarchical structures are intuitively to navigate for users and can be presented in the so called *dataguides*.

A dataguide based on the location of the sites could look like the one in figure 1.2 on the next page.

Please note that the user only has the choice between searching for sites using keywords or trying to navigate to the site using the lists in the existing system.

The attentive reader will notice that the two “trees” in the figure - also denoted dataguides<sup>1</sup>, have the word *classification* in their names. This is because they actually classify the contained sites by some property. The expanded dataguide is based on a classification of sites by location.

*The term “Classification” is used to describe the structure behind a “Dataguide”. A classification is usually an XML file having a structure like the one described in section 3.2.1 on page 47.*

Dataguides allow the user to navigate to interesting entries, just like lists do, but in a dataguide the relation between a child and parent “category” is much more obvious.

*Please note that further uses of “dataguide” refer to a object like the one shown in figure 1.2 on the next page*

Each category in the dataguides have a little check box next to its name (Denmark is selected in the example). This allows the user to improve his site queries. By selecting one or more categories that may contain interesting sites and entering keywords, the user is able to make much more complex queries compared to the simple queries in the existing system. It is also possible to combine selected categories in more than one classification, in order to put together even more sophisticated queries.

Imagine a user interested in “Castles”, “Historic Sites and Towns” and “Sites in Denmark” . The user could select the two categories and enter the keyword “Castle”, this would yield around 12 hits in our WH System, while the user would have

<sup>1</sup>The first one is fully collapsed, while the second is somewhat expanded.

Use the form below, if you want to make an advanced query in the WH site list



- [-] **Classification by geographical location**

This dataguide categorizes the WH sites by their geographical location around the world

- [-] **All sites in geographical location classification**

- [+] **Oceania**

- [+] **Asia**

- [+] **Americas**

- [-] **Europe**

- [+] **Western Europe**

- [+] **Eastern Europe**

- [+] **Southern Europe**

- [-] **Northern Europe**

- [+] **Sweden**

- [-] **Isle of Man**

- [-] **Iceland**

- [-] **Denmark**

- Jelling Mounds, Runic Stones and Church

- Roskilde Cathedral

- Kronborg Castle

- [-] **Svalbard and Jan Mayen Islands**

- [+] **Latvia**

- [+] **United Kingdom**

- [+] **Estonia**

- [+] **Lithuania**

- [+] **Finland**

- [+] **Norway**

- [-] **Faeroe islands**

- [-] **Channel Islands**

- [+] **Africa**

- [+] **Classification by miscellaneous categories**

Figure 1.2: A Dataguide generated by the tools developed for this report.

to click through each of the two lists in the search page of the existing system or simply enter “Castle” in the keyword search. The first is very tedious and the latter is very inaccurate, as it yields around 40 hits. Compare this to the same query using dataguides and the existing system returns 28 matches that lie outside the scope that the user wanted.

Another interesting problem in the WH domain, is the problem of making a good model for the site data. All the sites have some mutual properties like name, description and inscription criteria, but because of the radically different type of sites in the system, there are also a lot of properties that only are relevant for a subset of the sites. For instance a cathedral site would probably have some architectural style, a year it was constructed and so on, while these properties make no sense to put into a site describing a coral reef or a rain forest.

Over the past years the XML technology has evolved a lot and recently, complex query languages and databases for XML have started emerging. The fact that the XML technologies seem to have become somewhat usable, and that the different sites in the WH domain have so many different properties, makes it interesting to see if it is possible to take advantage of the semistructured nature of XML, when the WH sites need to be put into some format that computers can handle.

XML provides a more natural approach to holding semistructured data, while a traditional relational database requires tables, fields and types to be declared before each piece of data can be stored. Whereas an XML database just need to know that it is containing valid XML. This means that if a cathedral site needs to have a property describing its architectural style, that property would just be put right into the XML data describing the site. This makes storing the different sites very easy, but of course some problems arise when the data need to be extracted from the XML database, and used later on.

XML also have natural support for tree-like structures, thus making it a good choice for storing hierarchical data such as classifications.

The goal of this project is to explore the possibilities that the emerging XML technologies, such as XQuery and XML Databases, provide. We would like to explore the possibilities that querying of classifications introduce. Thereby making a more user friendly system, where the users have a much wider variety of query possibilities compared the the current system. One of the problems that arise is: How to create a simple usable interface, while still maintaining the complex query options beneath the surface.

The outcome should be a demo system that can take advantage of the data already typed into the existing WH system. A tool for creating classifications for the system should also be created. The demo system should be implemented using free software (preferably open source) and open standards, where applicable.



## Chapter 2

# Theory

*This section introduces the concept of semistructured data (SSD) and explains some of the advantages semistructured data models has over more traditional data models. XML is introduced as an example of practical use of SSD. A theoretical “generic query language” is introduced to give the reader an idea of how semistructured data generally is queried, and the XML query language “XQuery” is explained with a few examples.*

*The most important parts of this chapter explains how SSD can be used to represent classifications of WH sites, and how the classifications can be used for making some search facilities.*

### 2.1 Semistructured Data Models

Traditionally, data for computer programs is stored in a very structured manner, meaning that the data models used for the programs use some sort of schema, that describes the exact structure of the data. These data models are not very flexible, when it comes to making changes in the structure of the data. In order to change the structure, one often needs to redefine the schema. Traditional relational databases and object oriented databases are examples of programs, which uses these somewhat rigid data models.

Sometimes the exact structure of the data is not known, or the structure is subject to frequent changes. The data is then called *Semistructured Data* (SSD) and it can be an advantage to make use of semistructured data models instead of the “traditional” data models in those cases.

Because SSD has no schema associated, it is necessary for the SSD to be self describing. Attribute names must give a hint of what kind of data the attribute holds.

The World Heritage sites are not subject to frequent changes, but the heterogeneous structure of the WH sites, indicate that it could be an advantage to store WH site information in a semistructured format anyway.

In order to explain the differences between the different data models, we introduce a simple example just to illustrate the basic principles, the relational schema for the WH database is quite large, so the SSD modeling of that schema is postponed until the basics are covered. The example in figure 2.1 shows how a collection of music CD's could be represented in a RDBMS. The notation is more compact than

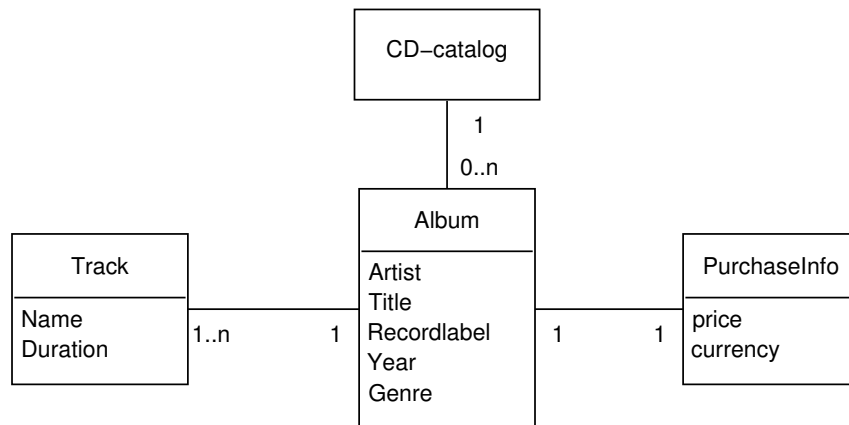


Figure 2.1: CD-catalog schema

that of the traditional ER diagram:

- Boxes represents entities.
- When a box is split by a horizontal line, the upper part contains the entity name and the lower part contains a list of attributes. Otherwise the box just contains an entity name.
- The lines between the boxes represents binary associations between entities. The numbers at the lines are the cardinalities – e.g. a CD-catalog may contain any number of albums and an album must have at least one track.

In a relational database the above schema is implemented with a table for each entity, and the data itself as records in these tables. The example below shows how this works. Some example data has been entered.

Album						
ID	CD-catalog	Artist	Title	RecordLabel	Year	Genre
1	1	Red Hot Chili Peppers	Blood Sugar Sex Magic	Warner Bros.	1991	Rock
2	1	Pearl Jam	Ten	Sony Music	1992	Rock
3	1	Red Hot Chili Peppers	By The Way	Warner Bros.	2002	Rock

Track		
Album	Name	Duration
1	The Power Of Equality	4.00
1	If You Have To Ask	4.11
2	Once	3.51
2	Even Flow	4.53
3	By The Way	3.37
3	Universally Speaking	4.19

PurchaseInfo		
Album	Price	Currency
1	NULL	DKK
2	NULL	DKK
3	8.99	GBP

CD-catalog
ID
1

There are a number of disadvantages to the data model used by relational databases:

- It is impossible to add new attributes to a table without changing the schema.
- Even if some attribute values are unknown, they must be present in the tables – e.g. the *price* attribute in the *PurchaseInfo* table has no value in two of the records, but a value must be present. Hence the *NULL* values.
- The table attributes are limited to simple types (e.g. character data or integers). It is impossible to nest a record inside another record. For instance, it is not possible to nest information about a track (from the *Track* table) inside a record in the *Album* table. Instead the *Track* table has an attribute (a *foreign key*), which points to the *ID* attribute of the *Album* table, describing a relation between the two.

Note that the “nesting feature” could be achieved with an object oriented database.

But the model also offers some noticeable advantages:

- Having a rigid schema with strict typing allows the RDBMS to perform quite a lot optimizations, resulting in good storage strategies and fast operations.
- RDBMS is a well-proven technology and is widely used.

The same CD-catalog can be described, without splitting it into several tables, using a semistructured data representation<sup>1</sup>:

```
{CD-catalog: {
  Album: {
    Artist: "Red Hot Chili Peppers",
    Title: "Blood Sugar Sex Magic",
    RecordLabel: "Warner Bros",
    Year: 1991,
    Genre: Rock,
    PurchaseInfo : {currency: "DKK"}
    Track: {Name: "The Power Of Equality", Duration: 4.00},
    Track: {Name: "If You Have To Ask", Duration: 4.11}
  }
}
```

<sup>1</sup>We adopt a notation used in [dotw] chapter for describing the data

```

Album: {
  Artist: "Perl Jam",
  Title: "Ten",
  RecordLabel: "Sony Music"
  Year: 1992,
  Genre: Rock,
  PurchaseInfo : {currency: "DKK"}
  Track: {Name: "Once", Duration: 3.51}
  Track: {Name: "Even Flow", Duration: 4.53}
}
Album: {
  Artist: "Red Hot Chili Peppers",
  Title: "By The Way",
  RecordLabel: "Warner Bros.",
  Year: 2002
  Genre: "Rock",
  PurchaseInfo : {price: 8.99, currency: "GBP"}
  Track: {Name: "By The Way", Duration: 3.37}
  Track: {Name: "Universally Speaking", Duration: 4.19}
}
}}

```

This representation of data is much more flexible than that of the relational database. In the following text the term *structure* refers to the “records” enclosed in curly braces, e.g.: {Attribute1:value1,Attribute2: value2}. Notice that it is possible to nest structures in arbitrary depth, hence the need for primary keys/foreign keys to define binary associations between structures is eliminated. Another advantage of this representation is, that undefined attributes can be completely omitted, since the data need not conform to some schema. The attribute *price* in the *PurchaseInfo* structure is now only present in the last album, where the price is known.

This data representation is actually a tree like structure. Figure 2.2 on the next page shows how the CD-catalog looks like<sup>2</sup>, when it is drawn as a tree. The simple attribute values become leaves and are illustrated with a bold font. The attribute names are shown as labels on the edges. The other nodes in the tree represents attributes.

Suppose a song appears on an album and also as part of a compilation of hit singles. In this case the track should appear under the album as well as under the compilation. This scenario is illustrated by the graph shown in figure 2.3 on the facing page. The node representing the track is a child of two other nodes: the album node and the compilation node. Obviously this is no longer a tree, and it cannot be represented directly with the textual notation used above. But it is necessary to specify how the “problem of multiple parents” should be handled. Two different approaches to solve this problem springs to mind:

1. A new attribute *ID* can be introduced in *Track* as a unique ID of that track. The track can then appear under the *Album* while the *Compilation* gets a new

<sup>2</sup>two of the albums from the CD-catalog has been omitted to make the figure fit the page



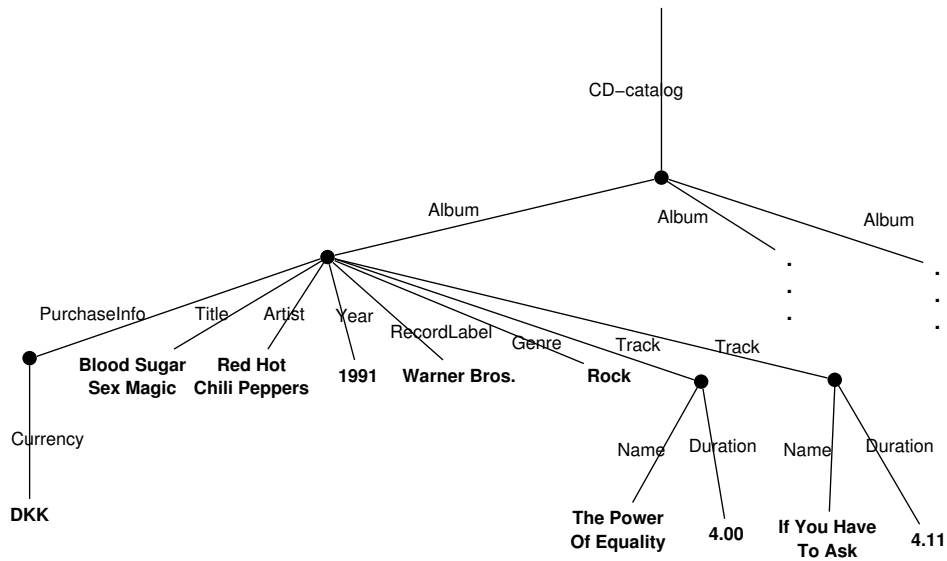


Figure 2.2: CD-catalog tree

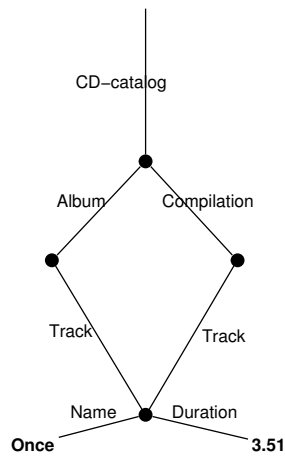


Figure 2.3: Track with two parents

attribute *TrackRef* which holds the value of the ID:

```
{CD-catalog: {
  Album: {
    Artist: "Pearl Jam",
    ...
    Track: {ID: "PJ1", Name: "Once", Duration: 3.51}
    ...
  }
  Compilation: {
    ...
  }
}
```

```

        TrackRef:  "PJ1"
        ...
    }
}}

```

This solution is very similar to the primary key/foreign key used in relational databases.

2. The notation could be extended with *pointers* or *references* like in object oriented data. [dotw] proposes the following notation for expressing references:

```

{CD-catalog: &o1{
  Album: &o2{
    Artist: "Pearl Jam",
    ...
    Track: &o3{Name: "Once", Duration: 3.51},
    ...
  }
  Compilation: &o4{
    ...
    Track:  &o3,
    ...
  }
}}

```

Each structure is assigned an object ID and other attributes can reference the structure by its object ID.

It is obvious, that semistructured data representations are more flexible than traditional “schema based” data representations. The biggest disadvantage of not using schemes is, that the attribute names must be chosen with care in order for the data to make sense. In most cases schema based data representations are also easier to use in applications, because the application developer knows all attribute names and types. Hence semistructured data representations should only be used, when it is impossible make a good structural representation of the data.

Having seen that SSD diverges from the common data structures and that SSD has its place in some applications, the question is how to actually represent SSD in a smart way, that is easy to handle using existing software. The answer is to use XML.

### 2.1.1 The eXtensible Markup Language – XML

The *eXtensible Markup Language* (XML) is a standard defined by the World Wide Web Consortium (W3C). The expression “XML” has been a buzzword for some years now, and everybody seems to agree, that the technology has proven its worth, but what is XML and how can it be used? In fact XML is merely a simple text format similar to HTML<sup>3</sup>, but as opposed to HTML, XML is not used for specifying

<sup>3</sup>this is no coincident since both formats are derived from the same markup language: *SGML*

how documents should be presented visually. XML alone is nothing but a text format suitable for storing textual data. Naturally there are a great deal of XML related technologies created for manipulating XML data, and these technologies makes XML a suitable format for many different types of applications.

This section and the next few sections will give an introduction to XML and the most important XML related technologies. The reader should gain a basic understanding of the different technologies, which is important since they are used in the implementation of the World Heritage application. Readers who are interested in a deeper understanding of these technologies should take a look at the W3C website [[w3c](#)].

### 2.1.2 The XML Format

Below is an example of an XML document describing the collection of CD's.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <CD-catalog>
3   <Album>
4     <PurchaseInfo currency="DKK"/>
5     <Artist>Red Hot Chili Peppers</Artist>
6     <Title>Blood Sugar Sex Magic</Title>
7     <RecordLabel>Warner Bros.</RecordLabel>
8     <Year>1991</Year>
9     <Genre>Rock</Genre>
10    <Track>
11      <Name>The Power Of Equality</Name>
12      <Duration>4.00</Duration>
13    </Track>
14    <Track>
15      <Name>If You Have To Ask</Name>
16      <Duration>4.11</Duration>
17    </Track>
18    .
19    .
20    .
21  </Album>
22  <Album>
23    <PurchaseInfo currency="DKK"/>
24    <Artist>Pearl Jam</Artist>
25    <Title>Ten</Title>
26    <RecordLabel>Sony Music</RecordLabel>
27    <Year>1992</Year>
28    <Genre>Rock</Genre>
29    <Track>
30      <Name>Once</Name>
31      <Duration>3.51</Duration>
32    </Track>
33    <Track>
34      <Name>Even Flow</Name>
35      <Duration>4.53</Duration>
```

```
36     </Track>
37     .
38     .
39     .
40 </Album>
41 <Album>
42     <PurchaseInfo price="8.99" currency="GBP" />
43     <Artist>Red Hot Chili Peppers</Artist>
44     <Title>Be The Way</Title>
45     <RecordLabel>Warner Bros.</RecordLabel>
46     <Year>2002</Year>
47     <Genre>Rock</Genre>
48     <Track>
49         <Name>By The Way</Name>
50         <Duration>3.37</Duration>
51     </Track>
52     <Track>
53         <Name>Universally Speaking</Name>
54         <Duration>4.19</Duration>
55     </Track>
56     .
57     .
58     .
59 </Album>
60 <Album>
61     <PurchaseInfo price="115" currency="DKK" />
62     <Artist>D.A.D</Artist>
63     <Title>Riskin' It All</Title>
64     <RecordLabel>Medley Records</RecordLabel>
65     <Year>1991</Year>
66     <Genre>Rock</Genre>
67     <Track>
68         <Name>Bad Craziiness</Name>
69         <Duration>3.16</Duration>
70     </Track>
71     <Track>
72         <Name>D-Law</Name>
73         <Duration>3.48</Duration>
74     </Track>
75     .
76     .
77     .
78 </Album>
79 <Compilation>
80     <PurchaseInfo price="12.99" currency="GBP" />
81     <Title>The very best of MTV unplugged 2</Title>
82     <RecordLabel>Warner Music and Universal International Music</
      RecordLabel>
83     <Year>2003</Year>
84     <Genre>Pop/Rock</Genre>
85     <Track>
86         <Name>Every Breath You Take</Name>
87         <Artist>Sting</Artist>
88         <Duration>5.07</Duration>
```

```
89     </Track>
90     <Track>
91         <Name>Wicked Game</Name>
92         <Artist>Chris Isaak</Artist>
93         <Duration>4.54</Duration>
94     </Track>
95     .
96     .
97     .
98 </Compilation>
99 </CD-catalog>
```

The full document is in appendix A on page 111.

The first line of the document is not that interesting. It is an XML declaration, which says that we are using XML version 1.0 with the ISO8859-1 font encoding. The rest of the document is the XML data itself. Basically XML data consist of two types of entities: *elements* and *attributes*. An element always has a start tag, e.g. `<Album>` and a corresponding end tag: `</Album>`. An element is a complex type, and it can contain other elements and text. When an element does not contain any data at all e.g. `<Album></Album>` it is called an *empty element* and it can be typed with a shorter notation. Instead of using a start- and endtag, one can use an empty-tag: `<Album/>`.

An XML document must have a *root element*, that encloses all other elements. In the above example the `<CD-catalog>` element is the root element.

Attributes are simple types included in the start tag of an element. The CD catalog above contains info about the purchase of each CD:

```
<PurchaseInfo price="12.99" currency="GBP" />
```

`price` and `currency` are examples of attributes. In general attributes should be used for meta data<sup>4</sup> e.g. one would usually specify a unique ID as an attribute.

One should notice, that the order in which the XML-elements appear in a document is important, since the XML-recommendation specifies, that elements should be regarded as a sequence, whereas the order of XML-attributes has no meaning (they can be regarded as a *bag* of attributes). This is useful, since some XML technologies for querying XML data supports indexing of elements, allowing access to an element without traversing the entire sequence.

Also notice that XML elements must be *properly nested*, meaning that e.g. the expression

```
<Album>
  <Title>
</Album>
</Title>
```

---

<sup>4</sup>Information about data

is illegal because the `<Title>...</Title>` tags should be completely enclosed within the `<Album>...</Album>` tags.

An XML document is said to be *well formed*, when the structure conforms to the rules described above.

### 2.1.3 A Semistructured WH Site

Having illustrated how semistructured data can be expressed in XML, it is now time to try and express a WH site in XML. Luckily we have received a copy of the database schema for the existing WH database, so some of the data modeling has already been done by the original designers. The original schema gives a few hints about what properties, that we may want to capture in our representation of the WH site data.

As mentioned earlier, structure of the WH site data may vary quite a bit between sites, making it hard to create a good relational schema. The schema for the original WH site data can be found in appendix H on page 145.

The original schema has some odd structures that puzzled us a bit. For instance when looking at the first page of the schema, the “location” and the “year” table are connected. This seems to be correct because the “location”, which is linked to “country”, may depend on year. If borders change at some point, the site may be located in another location (since location is connected with country).

However the contents of the “year” table turns out to be information describing when the “site” was inscribed into the WH. The result is that location is not dependent on year information and that application designers would have to join “site”, “location” and “year” in order to find the inscription date, very confusing and not a good example of data modeling.

We stumbled upon a few other odd constructs while going over the schema, but as we are taking a semistructured approach, we just chose to handle them in another way. However, this illustrates that making a good relational schema for the WH sites is a complex task.

An example of how the site “Roskilde Cathedral” could look in semistructured data follows:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <Site id="site259">
3   <Name>Roskilde Cathedral</Name>
4   <Number>695</Number>
5   <BriefDescription>Built in the 12th and 13th centuries, this was
      Scandinavia's first Gothic cathedral to be built of brick
      and it encouraged the spread of this style throughout
      northern Europe. It has been the mausoleum of the Danish
      royal family since the 15th century. Porches and side
      chapels were added up to the end of the 19th century. Thus

```

```

        it provides a clear overview of the development of European
        religious architecture.
6    </BriefDescription>
7    <Resources>
8      <LinkResource>
9        <URL>http://www.natmus.dk/</URL>
10       <Name>National Museum of Denmark</Name>
11      </LinkResource>
12      <Report>
13        <Name>Report of the 19th Session of the Committee</Name>
14        <URL>http://whc.unesco.org/archive/repcom95.htm#695</URL>
15      </Report>
16    </Resources>
17    <Justification>
18      <Criterion>
19        <Number>ii</Number>
20        <CategoryType>C</CategoryType>
21        <Description>exhibit an important interchange of human
           values, over a span of time or within a cultural area of
           the world, on developments in architecture or
           technology, monumental arts, town-planning or landscape
           design.
22      </Description>
23    </Criterion>
24    <Criterion>
25      <Number>iv</Number>
26      <CategoryType>C</CategoryType>
27      <Description>be an outstanding example of a type of building
           or architectural or technological ensemble or landscape
           which illustrates (a) significant stage(s) in human
           history.
28    </Description>
29    </Criterion>
30  </Justification>
31  <Location>
32    <Name>Island of Sjaelland</Name>
33    <Country>Denmark</Country>
34    <Year start="12th century" />
35    <LocationPoint>
36      <Lat_degree>12</Lat_degree>
37      <Lat_minute>4</Lat_minute>
38      <Lat_second>47.2</Lat_second>
39      <Lat_hemi>E</Lat_hemi>
40      <Long_degree>55</Long_degree>
41      <Long_minute>38</Long_minute>
42      <Long_second>32.5</Long_second>
43      <Long_hemi>N</Long_hemi>
44    </LocationPoint>
45  </Location>
46  <Inscribed>1995</Inscribed>
47  <ArchitecturalStyle>Gothic</ArchitecturalStyle>
48 </Site>

```

As it can be seen, the <Location> construct have a <Year> element with a start

attribute, the element could also contain an `end` attribute indicating the “end” of a location. There are several possible ways to handle the “location - year” connection, but this one is very flexible, it actually allows a “site” to be moved to another physical location at some point, if that should be necessary.

The textual “type” of `<Year start="12th century"/>` prohibits us from performing calculations using the value, but it fits nicely into the layout created by the presentation layer.

The `<ArchitecturalStyle>` element has been added in order to illustrate how *extra* information about the site, compared to the data found on the original WH site, may be marked up.

Section 3.2.2 on page 49 gives a deeper description of the “site” data.

#### 2.1.4 XML Data Models

In order to make use of XML data in different programming languages it is necessary to define a common data model, that is used as a “standard” for processing XML data. The most well known data models for XML are the *Simple API for XML* (SAX) and the *Document Object Model* (DOM). The DOM is a “real” standard defined by the W3C, while SAX originally only was implemented in Java, but it is now a “de facto” standard with implementations in many different programming languages. SAX is a lightweight API and it is not nearly as powerful as the DOM, but therefore SAX is also much faster and this is probably the reason for its popularity.

This section describes the DOM and another data model very similar to the DOM, which is used by the XML related technologies *XPath* and *XQuery* (XML Query). SAX is not described, since it has no relevance for this project. Note that the data models are under continuous development, and there are different versions of the data models. The following applies to DOM level 1, XPath v2.0 and XQuery v1.0, but it is probably general enough to apply to all current versions of the data models.

DOM is used as a middle tier between XML and the internal representation of a classification in the application used to generate classifications for the WH system.

##### The DOM

DOM is an API which has some convenient classes and methods for managing XML documents. It is called a *Document Object Model* because it regards all XML data as documents and the different parts of the document are represented as “objects” or “classes”. The following text will avoid the term “object”, because people have different opinions about what an object is – instead the term “class” is used for an implemented API, and an “instance” is an instance of a class.



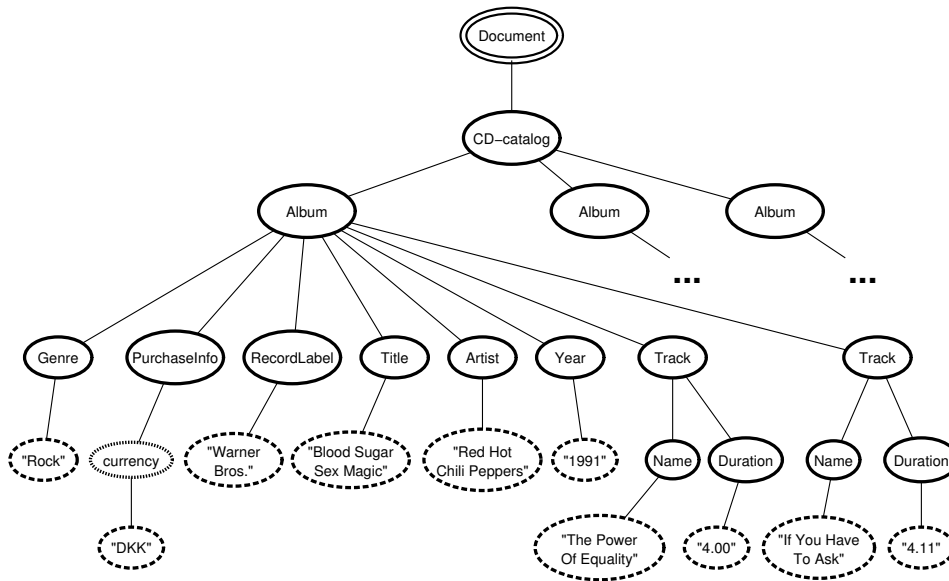


Figure 2.4: DOM representation of CD-catalog example

Figure 2.4 shows the DOM representation of the CD-catalog example. All the ellipses in the figure represents instances of classes. The most important thing to notice is, that all the instances are instances of the *Node* class. In other words: in the DOM *everything is a node*. This is convenient because there are several “universal” methods, which makes sense to use on a node, regardless of whether the node represents an attribute, a document, an element or other things.

Naturally there are operations you should be able to use on an element, that you cannot use on e.g. an attribute. For this reason there are also some more specialized classes that specifically represents elements, attributes, text, documents etc. In programming languages like Java and C++ the specialized classes extend the *Node* class and add additional methods.

The line styles in the figure indicates the types of specialized classes:

- Double bold lines indicate a *Document* class instance, which represents the entire document.
- Bold lines indicate *Element* instances.
- “Fuzzy” lines indicate *Attr* (attribute) instances. The figure only has one attribute called `currency`
- Bold punctuated lines indicate *Text* instances.

The figure does not show all the classes extending the *Node* class – there are a total of 12 subclasses to *Node*:

*Attr*, *CDATASection*, *Comment*, *Document*, *DocumentFragment*, *DocumentType*, *Element*, *Entity*, *EntityReference*, *Notation*, *ProcessingInstruction*, *Text*

In addition to the already mentioned classes, there are some other convenient classes for managing XML documents, e.g. a class for representing a collection of nodes. This section will not go into the boring details about the methods in the different classes, it is sufficient to know that there are some classes for removing nodes, fetching child nodes, adding new nodes etc.

### The XQuery and XPath Data Model

The data model used in XQuery and XPath is very similar to the DOM. This data model is not meant to be used in an object oriented programming languages, but in a query language. For this reason the nodes are not regarded as objects, and it does not use “methods” but “functions” and “operations”.

The data model has the following kinds of nodes:

**document** Represents a whole XML document.

**element** Represents an XML element.

**attribute** Represents an XML attribute.

**text** Represents a text string.

**namespace** Represents an XML namespace.

**processing-instruction** Represents a processing instruction. The “special” tags that has the structure `<? ?>` are processing instructions.

**comment** Represents a comment in an XML document. Comments are enclosed within `<!-- -->`.

Except for the namespace node all the node types in the above list are also present in the DOM, although the names are a little different. Figure 2.4 on the page before is actually also a representation of the CD-catalog as an XQuery and XPath data model.

## 2.2 Schemas for Semistructured Data

As mentioned earlier, it is hard to define a common schema for the World Heritage site data, and when considering semistructured data it is actually possible to use the data without a schema.

However having knowledge about the structure of the data gives some advantages:

- Complex queries can be created.
- Queries can be accelerated with the use of indexes.
- High degree of control over the presentation format.
- Data storage can be optimized.

Basically the problem is to determine a schema that makes the data usable without losing too much flexibility. It may seem a bit odd to require a schema for

semistructured data, but in order to be able to take advantage of semistructured data, it is necessary to have some kind of basic knowledge about the structure of the data.

There exist several formalisms for describing the structure of semistructured data, and some of them will be mentioned in this section.

Since semistructured data is self-describing, it must be possible to obtain a schema from the data itself. Basically two schema types exist, *upper-bound* and *lower-bound*. An upper-bound schema includes information about all the elements that the data documents **may** include, while the lower-bound schema specifies the elements that all documents **must** include.

### 2.2.1 Schema formalisms

At present time there is no formalism that is *the* right way of describing SSD schemas, so a couple of the simple ones will be illustrated in this section.

#### Logic

Logic can be used to describe schemas. The idea is that a set of rules that describe the properties of the different elements is declared. Several branches of logic exist and `Datalog` is a somewhat simple language that can be used to describe and validate a schema. Another possibility is to use `Description logic` that is able to support even more complex constructs.

An example of how a simple set of `Datalog` rules can be used to describe a schema for the SSD presented in the CD Catalog example in section 2.1 on page 7, could look like this:

```

CD-Catalog(X)      :-  ref(X,album,Y), Album(Y)
Album(X)           :-  ref(X,artist,Y1), String(Y1),
                       ref(X,title,Y2), String(Y2),
                       ref(X,recordLabel,Y3), String(Y3),
                       ref(X,year,Y4), String(Y4),
                       ref(X,genre,Y5), String(Y5),
                       ref(X,purchaseInfo,Z), PurchaseInfo(Z),
                       ref(X,track,Z1), Track(Z1)
PurchaseInfo(X)    :-  ref(X,currency,Y), String(Y)
PurchaseInfo(X)    :-  ref(X,currency,Y), String(Y),
                       ref(X,price,Z), Float(Z)
Track(X)           :-  ref(X,name,Y),String(Y),
                       ref(X,duration,Z), Float(Z)

```

The simple rules express information about which relations that **must** exist between elements. The two

`PurchaseInfo(X)` rules express that two different `PurchaseInfo` constructs exist and at least one must be included.

`Datalog` is excellent for describing *lower bound* schemas, but it can be hard to describe an *upper bound* schema using `Datalog` because multiplicities and complex sub-elements require a lot of extra rules to be added. `Datalog` can easily express the typing of schemas though.

### Schema graphs

Another way to describe schemas is to use a schema graph obtained by simulation. This approach builds upon the fact that SSD can be thought of as being a graph. Schema graphs usually express which relations that **may** exist, so schema graphs define *upper bound* schemas.

The schema graph for the CD-Catalog data-graph shown in figure 2.4 on page 19 would look like this:

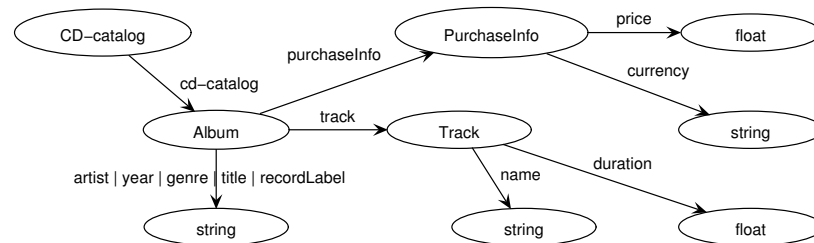


Figure 2.5: The schema graph for the CD-Catalog example.

The simple types in figure 2.5 appears in several ellipses. Usually they should be merged into the same, but in order to keep the schema graph on a form that is easy to view, they have been split up.

Schema graphs gives a better overview over the data because they describe upper bound schemas naturally. However as this small example already has shown, the graphs tend to grow huge when the data they describe is complex. The shown schema graph does not include multiplicities, but they could be added to the edges, if necessary.

### XML Schema

The XML Schema format from [w3c] has been a recommendation since 2. May 2001. XML Schemas can be used to describe and validate XML data, and they support structure information and typing- and ordering of elements.

XML Schemas use a less compact format compared to `Datalog`, but they have some constructs that are closer to the actual data format (XML Schemas are actually written in XML themselves).

The XML Schema describing the CD-Catalog, defined in section 2.1.2 on page 13, could be described by the following schema (please note that the part of the schema that describes the `Compilation` has been removed because it looks quite like the `Album` construct and it is not necessary in order to illustrate what a schema might look like):

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <schema xmlns='http://www.w3.org/2001/XMLSchema'>
3 <element name='CD-catalog'>
4   <complexType>
5     <sequence maxOccurs='unbounded'>
6       <element name='Album'>
7         <complexType>
8           <sequence>
9             <element name='PurchaseInfo'>
10              <complexType>
11                <attribute name='price' type='decimal' minOccurs='
12                  0' />
13                <attribute name='currency' type='string' />
14              </complexType>
15            </element>
16            <element name='Artist' type='string' />
17            <element name='Title' type='string' />
18            <element name='RecordLabel' type='string' />
19            <element name='Year' type='string' />
20            <element name='Genre' type='string' />
21            <element name='Track' maxOccurs='unbounded'>
22              <complexType>
23                <sequence>
24                  <element name='Name' type='string' />
25                  <element name='Duration' type='decimal' />
26                </sequence>
27              </complexType>
28            </element>
29          </sequence>
30        </complexType>
31      </element>
32    </sequence>
33  </complexType>
34 </schema>
```

The full schema can be found in appendix B on page 119.

XML Schemas are very good for describing and typing XML data, but like the other formalisms mentioned in this section, it is hard to get a quick overview over complex data and the relations that exist within the data, when using XML Schemas alone.

### Our schema format

Since `Datalog`, Schema graphs and XML Schemas tend to get so complex that it is hard to get a good overview of the structure of the data, we have decided to create our own notation to give an overview over the structure and relations. The notation that we have used is strongly inspired from UML and it describes structure and cardinalities but does not contain type information. The typing could have been included but as it does not contribute noteworthy to the understanding of the structure of the data, it has been left out.

The format is supposed to be used in conjunction with XML Schemas and it is really usable when designing data from “scratch”. When the structure is in place an XML schema can be created and type information added to the XML Schema.

A legend, created using the notation itself and describing the notation, can be seen in figure 2.6. The contents of the legend looks like the things known from UML.

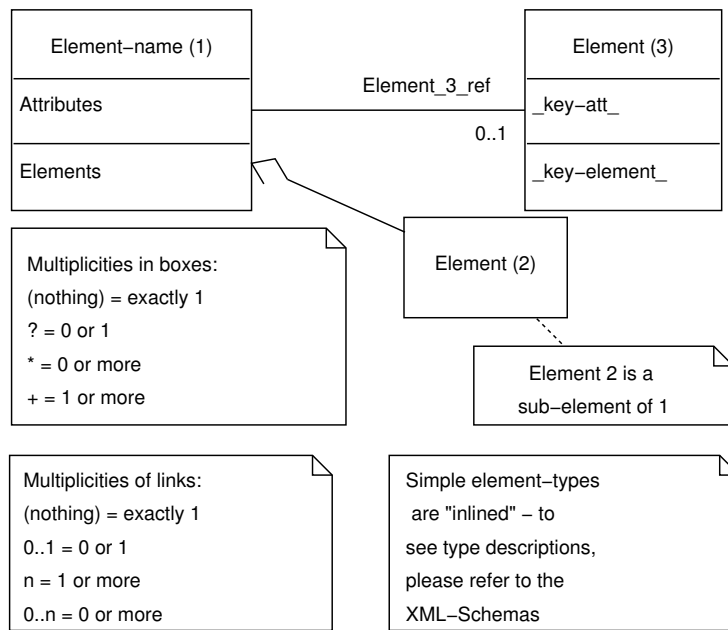


Figure 2.6: Legend for the diagram notation

Notes are illustrated as boxes with a small fold in the upper right corner.

Complex elements are drawn as boxes, simple elements<sup>5</sup> are put in the lower part of the box, attributes in the middle part and the element name in the upper part. Multiplicities on elements are like the ones known from path expressions (also described in the note “Element multiplicities”.) Unique identifiers are indicated by surrounding the name with “\_” e.g. `_ID_`.

<sup>5</sup>Elements with no sub elements

An association with a *role-name* (e.g. `Element_3_ref`) indicates that there will be an `Element_3_ref` element in element 1 and that the element contains something that can be used to identify the actual element 3 (a sort of foreign key from the referenced element – usually an ID-attribute or a sub element, unique identifier for the element, marked with “\_<name>\_”).

### 2.2.2 Obtaining a schema

Several possibilities for obtaining a schema exist:

- Extraction
- Inference
- Specification

In the WH system the data did not exist on semistructured form on beforehand, so specification is the only technique that is really used in this context.

#### Extraction of schemas

Since SSD is self describing it is possible to extract the schema from the data itself. Simple algorithms for extracting schemas in some of the aforementioned formats exist. Of course there exist no tools for generating schemas in our own format, but the primary goal of that format, is to be used when designing documents.

One of the pitfalls when obtaining schemas from extraction is that the schemas always fit the data, if an error exist in the structure of the data it will be included in the schema definition. For instance a typo in an element name. But extraction is certainly a usable tool when the data documents come from a third party without an associated schema.

In order to get a quick overview of a data format extraction can be used to generate a dataguide from the data, this would enable the user to navigate through the schema using a tree structure. However in the case of complex data constructs the dataguide may be bloated with lots of information, thus making it hard to make any sense from it.

Please note that we rely heavily on dataguides for adjusting search scopes and presenting results in the following sections, but those dataguides are generated from a well specified structure in order to be able to define a very general approach for making classifications in XML.

#### Inference from queries

Sometimes SSD come from legacy systems, and it is more convenient to construct the schema from the query generating the data, rather than extracting the schema from the data. But it is not always possible to infer a usable schema in this way.

This technique is just mentioned for completeness, it is not used in the project.

### Specification of schemas

Software engineers usually have very free hands when designing the internals of software systems. This includes the data modeling. When no data is specified on beforehand, the task is to design a schema that can handle all the necessary constructs. It is in this case that our own UML like schema notation is really usable. It provides a nice overview of the structure of the data, and it is a trivial task to generate an XML Schema that extends the graphical representation with type information.

## 2.3 Querying SSD

In order to use semistructured data it is obviously necessary to find a way to fetch data from an SSD document. In traditional RDBMS the query language SQL is used for getting the data, but because the semistructured data model is fundamentally different from the relational data model, a very different query language must be specified. The list below describes some important differences between SSD and relational data. These differences makes it impossible to use a simple query language, like SQL, for semistructured data.

1. Semistructured data allows nesting of structures in arbitrary depth.
2. SSD documents can contain attributes, that are unknown to the system/people interacting with the document – in other words, there is no fixed database schema when using SSD.
3. Attribute names does not need to be unique within a structure.

Because of the first property a query language for SSD must be able to handle hierarchical structures. When looking at figure 2.2 on page 11 this corresponds to vertical navigation in the tree – that is, navigation from a node to its children (or from a node to its parent). Most<sup>6</sup> existing query languages uses *path expressions* to solve this problem. Path expressions are described in section 2.3.1 on the next page.

The second and third property makes similar demands to the query language. Because the attribute names (the “labels”) are unknown, or because some attributes have the same name, the query language must be able to select a collection of attributes – even without knowing their names – iterate through the collection and process each attribute of the collection individually.

The following sections will describe a query language fulfilling the mentioned requirements<sup>7</sup>.

---

<sup>6</sup>Probably all

<sup>7</sup>This query language is proposed in [dotw]



### 2.3.1 Path expressions

Path expressions works by matching attribute names – it is best illustrated with some examples. The examples below use the CD-catalog data described earlier and illustrated in figure 2.2 on page 11.

- This example shows how to make simple selections of attributes, when the attribute names are known. All the album titles are selected.

```
CD-catalog.Album.Title
```

Result:

```
{Title: "Blood Sugar Sex Magic",Title: "Ten",Title: "By The Way"}
```

Notice that the result includes the labels (`Title`). This is slightly different from the query language proposed in [dotw], but it seems that the path expression is more powerful, when the labels are included in the results.

- Suppose that the CD-catalog also included compilations in addition to the existing albums. The following example shows how to select titles of all albums and all compilations:

```
CD-catalog.(Album|Compilation).Title
```

The result is the same as before, because there are no compilations, but notice it is possible to match either `Album` or `Compilation` with `|`.

- If the name of an attribute is not known, it can be matched with the *wild card* “\_”

```
CD-catalog...Title
```

This gives the same result as above.

- In order to fetch all tracks in any depth in the data document, the following expression can be used:

```
CD-catalog..*.Track
```

The `*` specifies any number of repetitions of a label – here any label because of the wild card. The query language has the following operations for specifying cardinality constraints:

- \* any number (including zero)
- + one or more
- ? “optional”, meaning zero or one

- The query language also allows matching of labels using regular expressions. This example selects all attributes in any depth below `CD-catalog` with a name starting with “C” or “G”:

```
CD-catalog..*.'[CG].*'
```

Result:

```
{Genre: "rock", Currency: "DKK", Genre: "Rock",
Currency: "DKK", Genre: "Rock", Currency: "GBP"}
```

- All the examples above return labels as part of the results, but what if the labels should be left out? This problem can be solved by introducing a func-

tion `value()`, that returns the value of a given attribute:

```
CD-catalog.Album.Title.value()
```

Result:

```
{"Blood Sugar Sex Magic", "Ten", "By The Way"}
```

The result is identical to that of the first example, except that the labels are left out.

The path expressions provides a powerful way of retrieving data from a single document, but a complete query language should be able to retrieve data from several documents and make transformations on the results of path expressions. The next section describes a “generic” query language, which closely resembles existing query languages for semistructured data<sup>8</sup>.

### 2.3.2 The Generic Query Language

The structure of the query language is to some extent similar to SQL – it is also based on SELECT-FROM-WHERE expressions. The query language is described below with a single example – the readers who are more interested in query languages for SSD should take a look at [dotw]<sup>9</sup>

Consider the following query:

```
select  BigTrackName: name
from    CD-catalog.Album.Track track,
        track.Duration dur,
        track.Name.value() name
where   dur.value() > 4.00
```

The `from` statements specifies an iteration through all Albums in the CD-catalog shown earlier. Each track is stored in the variable `track`, the variable `dur` holds the duration of the track, e.g. `Duration: 3.51`, and `name` holds the name of the track *without* the label e.g. "Once".

The `where` statement specifies that only tracks with a duration larger than 4 minutes must be chosen.

The `select` statement specifies that `name` attribute is returned with the label `BigTrackName`. The result is a collection of names of tracks longer than 4 minutes, where the label `Name` has been renamed `BigTrackName`:

```
{
BigTrackName: "If you have to Ask",
BigTrackName: "Even Flow",
BigTrackName: "Universally Speaking"
}
```

<sup>8</sup>The query language is identical to the one proposed in [dotw] sections 4.2, except, of course, that the path expressions used are modified as explained earlier

<sup>9</sup>But still remember that the path expressions used in [dotw] is slightly different than the one used in this paper

The generic query language described above is very similar to W3C's recommendation for the XML query language *XQuery*, which can be used in practice for making applications based on semistructured data. There are differences in the syntax, but the expressive power of the languages are almost the same.

### 2.3.3 XPath

XPath is the path expressions used by the XML related technologies XSL (eXtensible Stylesheet Language) and XQuery. It is very similar to the theoretical path expressions used in the generic query language, but in some ways it is not as powerful – the greatest disadvantage is, that it does not support regular expressions for matching tag names or values. Fortunately it has some other clever features, which in other ways makes it extremely powerful. The table below shows some examples of how things are expressed using the different notations.

Generic Path Expression		XPath expression	
Choice			
	CD-catalog. (Album Compilation)/Title		/CD-catalog/Album/Title  CD-catalog/Compilation/Title
Wildcard			
-	CD-catalog...Title	*	/CD-catalog/*/Title
Arbitrary depth			
-*	CD-catalog_*.Title	//	/CD-catalog//Title
Value extraction			
value()	CD-catalog.Album.Title.value()	text()	/CD-catalog/Album/Title/text()

Below is given some examples, which explains the features of XPath. The XML document with the CD-catalog from section 2.1.1 on page 12 is used for the examples.

- Simple selection of elements is done exactly as with the generic query language, except that the separator . is replaced by /. This example selects all titles from all albums:

```
/CD-catalog/Album/Title
```

Result:

```
<Title>Blood Sugar Sex Magic</Title>
<Title>Ten</Title>
<Title>By The Way</Title>
<Title>Riskin' It All</Title>
```

The XPath expression / always returns the root node, so in this example the expressions / and /CD-catalog will return the same result – the entire document.

- Selection of attributes in XPath is very similar to selecting elements, but all attribute names are preceded by “@”. For instance, the expression:

```
/CD-catalog/*/PurchaseInfo/@price
```

Will select the price attribute of the <PurchaseInfo> element.

- A nice feature of XPath is the ability to make conditional select statements. Conditions are enclosed within [ . . ]. The expression

```
//Album[PurchaseInfo/@price]/Title
```

Selects all <Title> elements of albums where the price attribute is specified. The example below is a little more complicated. It selects all albums which have some descendant element with the value “Once”. This particular query applied to the CD-catalog will return the album “Ten” by the band “Pearl Jam”, because it has a track called “Once”.

```
//Album[.//*=‘Once’]
```

### 2.3.4 XQuery

XQuery is a query language for XML specified by W3C. The specifications for XQuery are still just work in progress, but the specifications are no longer subject for frequent changes. Any changes that are made to the XQuery language are now minor, and they will probably not affect the validity of information in this section.

Syntactically XQuery seems like a mixture of an ordinary query language like SQL, and a functional programming language. It uses a *FOR-LET-WHERE-RETURN* structure similar to the *SELECT-FROM-WHERE* structure in SQL, but additionally XQuery allows the use of several decision statements e.g. *if-then-else* statements. One of the strongest features of XQuery is the support of variables and functions, which makes it easy to separate queries into several chunks of code – this makes it a lot easier to write queries that are easy to understand and debug.

#### XQuery Basics

Consider the small XQuery below:

```
1 for $track in doc("cdcatalog.xml")/CD-catalog/Album/Track
2 let $dur := $track/Duration
3 let $name := $track/Name/text()
4 where $dur > 6.00
5 return <BigTrackName>{$name}</BigTrackName>
```

The query does the exact same thing as the “big track query” from section 2.3.2 on page 28, except it queries all data in the CD-catalog (appendix A on page 111) and it selects tracks that are longer than 6 minutes instead of 4. The XQuery is remarkably similar to the generic query – the main difference is, that the XQuery `return`

statement is in the end of the query, while the corresponding `select` statement in the generic query language is in the top. Iteration through a sequence is done using the `for` statement, `let` expressions are used for assigning values to variables.

The XQuery above returns this result:

```
<BigTrackName>Sir Psycho Sexy</BigTrackName>
<BigTrackName>Release</BigTrackName>
<BigTrackName>Venice Queen</BigTrackName>
```

which are the only tracks in the CD-catalog longer than 6 minutes.

### Advanced Features

The XQuery in this example uses some of the more advanced features of XQuery. This query can be used to filter the CD catalog more refined than the above query, which just was able to get tracks that were longer than 6 minutes. The below query can find tracks that are longer than some user specified number of minutes, excluding all tracks that contains some user specified word.

```
1 declare namespace wh="WorldHeritage"
2
3 define variable $cdcatalog as node()
4     {doc("cdcatalog.xml")/CD-catalog}
5
6 define function wh:filterTracks($minLength as xs:decimal,
7     $censureWord as xs:string) as node()? {
8     let $filteredTracks :=
9         <FilteredTracks>
10        {
11            for $track in $cdcatalog/Album/Track
12            let $dur := $track/Duration
13            let $name := $track/Name/text()
14            where $dur>$minLength and not(contains($name,$censureWord))
15            return <BigTrackName>{$name}</BigTrackName>
16        }
17    </FilteredTracks>
18    return
19        if(count($filteredTracks/*)=0) then()
20        else($filteredTracks)
21 }
```

Line 1 in the query declares the namespace `wh`. Namespaces are primarily used for grouping functions, allowing function names to overlap as long that the functions belongs to different namespaces. The namespace for predefined XQuery functions is `fn`, but it is used as default and need not be used.

Line 3 defines a global variable `cdcatalog` of type `node()`. Recall that in the XQuery/XPath data model (described in section 2.1.4 on page 18) a node is pretty much anything e.g. XML documents, elements or attributes. The XPath expression in line 4 assigns a value to the `cdcatalog` variable.

Line 6 starts a definition of the function `filterTracks`. Notice that it belongs to the namespace `wh`. The function takes two arguments: a decimal number: `minLength` and a string: `censureWord`. The predefined simple types use the namespace `xs`. Also notice that the purpose of the function is to find all tracks longer than `minLength` minutes, that does not contain the word `censureWord`. The `as node()?` part of the function signature defines that the function returns a sequence of nodes, and the sequence has cardinality `?` which means zero or one. Legal cardinality constraints are:

- Nothing specified means exactly one.
- `+` one or more.
- `*` zero or more.
- `?` “optional” – zero or one

The `where` statement in line 14 uses two standard XQuery functions: `not` that negates a truth value, and `contains` that returns true if the second argument is contained in the first.

The final return statement in line 18 uses a `if-then-else` statement. It uses the `count()` function to count if the `filteredTracks` variable contains any tracks at all. If this is not the case, then nothing is returned. If there actually are some tracks, then the `filteredTrack` variable is returned.

The result of the function call `wh:filterTracks(6.0, "Psycho")` is:

```
<FilteredTracks>
  <BigTrackName>Release</BigTrackName>
  <BigTrackName>Venice Queen</BigTrackName>
</FilteredTracks>
```

Notice that the track “Sir Psycho Sexy” now has been filtered away.

This section does obviously not give an exhaustive description of all XQuery features, but it provides an overview of some of the most important features. Readers who are more interested in XQuery should go and read the XQuery language specification [[XQ-lang](#)] and the specification of XQuery functions and operators [[XQ-funop](#)].

By now it should be clear how semistructured data can be stored as XML documents, and how these documents can be queried and transformed using the powerful XQuery language. The next section describes how it is possible to use semistructured data for storing information about sites in the World Heritage list.

## 2.4 World Heritage Classifications

Many of the sites in the World Heritage system are related in more or less obvious ways. An important question is: How can these relations be expressed in a way that can be exploited, and that a computer can handle?

The relationships are built upon one or more properties describing each site. If we take a look at the example site in section 2.1.3 on page 16, and locate the `<CategoryType>` property in the `<Justification>` element, that describes whether a site is inscribed because of its natural or cultural properties. The property indicates that some kind of categorization exists among the sites. This categorization is based on the `CategoryType` property in the `Justification`, but other categorizations based on other properties are most likely to exist. The classification based on `Location` has also been mentioned earlier.

The categorization by `CategoryType` divides the sites into two classes, a class containing the set of sites with a natural `CategoryType` and a class with cultural `CategoryType` sites. It is possible to inscribe the same site several times using different criteria – the result is that some sites have both a natural and cultural `CategoryType` property. These sites can be thought of as specializations of both the natural and cultural classes. The class that these *hybrid* sites form can be called *mixed*.

The structure that the classification based on the `CategoryType` property forms is denoted a lattice.

### 2.4.1 Partial Orders and Lattices

This section contains a brief introduction to lattices. The lattices are important in this system, because they can be used to express classifications. Lattices are a specialization of partially ordered sets.

#### Partial Orders

A partial ordering is a relation  $\sqsubseteq: L \times L \rightarrow \{true, false\}$  with the following properties:

- *reflexivity* (i.e.  $\forall l : l \sqsubseteq l$ )
- *transitivity* (i.e.  $\forall l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$ )
- *anti-symmetric* (i.e.  $\forall l_1, l_2 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$ )

A partially ordered set  $(L, \sqsubseteq)$  is a set  $L$  equipped with a partial ordering  $\sqsubseteq$ .

A subset  $Y$  of  $L$  has  $l \in L$  as an *upper bound* if  $\forall l' \in Y : l' \sqsubseteq l$  and as a *lower bound* if  $\forall l' \in Y : l' \sqsupseteq l$ . A *least upper bound* (LUB)  $l$  of  $Y$  is an upper bound of  $Y$  that satisfies  $l \sqsubseteq l_0$  whenever  $l_0$  is another upper bound of  $Y$ . Similarly a *greatest lower bound* (GLB) can be defined.

Partially ordered sets need not to have LUB or GLB but when they exist, they are unique because  $\sqsubseteq$  is anti-symmetric.

## Lattices

A partially ordered set where all subsets have both LUB's and GLB's is denoted a lattice. Furthermore the notion of a *least-* and *greatest element* is also introduced. When talking lattices, the least element is called “bottom” or  $\perp$  and the greatest element is called “top” or  $\top$ .

Lattices can be drawn as the example in figure 2.7, and that kind of drawings is called *Hasse diagrams*. The diagram illustrates the same specializations as men-

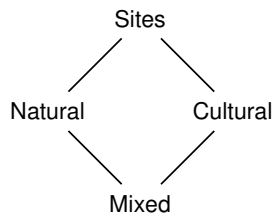


Figure 2.7: Hasse diagram

tioned in the last section. As a convention all our lattice diagrams will have the most general class on top and the most specialized in the bottom.

### 2.4.2 Classifications as Lattices

All the sites in the `CategoryType` example can be put into a general class, called top ( $\top$ ), and then the classes natural and cultural can be viewed as specializations of  $\top$ .

The structure of the classes (based on the category type specializations) form a lattice that is illustrated in figure 2.8. The most general class is on top and the most specialized in the bottom. The classifications does not need to have a greatest

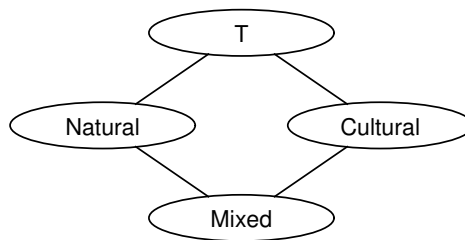


Figure 2.8: The simple representation of the category-type classification

lower bound element (GLB) specified explicitly, if some classes lack a GLB class, the bottom ( $\perp$ ) class is assumed to exist and be a specialization of these classes.



This is illustrated with the punctuated lines in figure 2.9 and it is obvious that this assumption makes the classifications fulfill the lattice properties<sup>10</sup>.

The reason that the  $\perp$  class is assumed to exist and not just defined in the classifications, is that the  $\perp$  class does not contain any sites at all. It would not make sense to put the  $\perp$  class into the dataguides that the classifications will be converted into, before being presented to the user.

Each site can belong to multiple classes. If a site for instance is a member of the Mixed class it is also member of all the classes that the Mixed class is a specialization of (Natural, Cultural and  $\top$ ).

Furthermore it is possible to refine the classes by inserting sub-categories, without changing the overall structure. The principle is illustrated in figure 2.9.

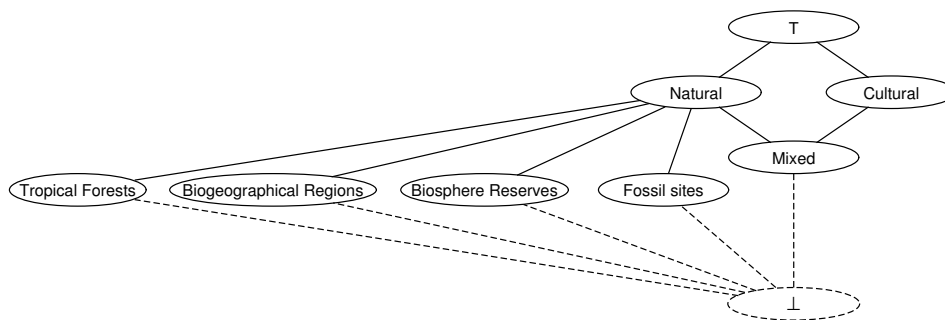


Figure 2.9: Extension of the simple category-type classification

The classifications introduce some exciting possibilities in the WH system, because they incorporate extra knowledge. In a RDBMS the classifications have to be worked out when designing the database schema, because the classifications usually introduce additional properties on the sites. This implies that if a new classification with a different depth or structure was to be embedded in the system at a later point, the schema and the applications that use the schema would need to be changed.

By using the SSD approach the natural choice would be to store the extra properties, that a new classification would impose, in the classification itself. This strategy would not call for changes in the “schema”<sup>11</sup> for the WH sites and it would be possible to create a generic piece of code to handle the presentation of the classifications.

Having introduced some of the most common concepts when doing classifications, the next step is naturally to mention ontologies. The website <http://www.whatis.com> gives the following definition of the word “ontology”:

<sup>10</sup> A lattice is a partial order where GLB- and least upper bound classes exist for all classes.

<sup>11</sup> The lower bound schema

In its general meaning, ontology is the study or concern about what kinds of things exist - what entities there are in the universe. It derives from the Greek onto (being) and logia (written or spoken discourse). It is a branch of metaphysics, the study of first principles or the essence of things.

In information technology, an ontology is the working model of entities and interactions in some particular domain of knowledge or practices, such as electronic commerce or “the activity of planning.” In artificial intelligence (AI), an ontology is, according to Tom Gruber, an AI specialist at Stanford University, “the specification of conceptualizations, used to help programs and humans share knowledge.” In this usage, an ontology is a set of concepts - such as things, events, and relations - that are specified in some way (such as specific natural language) in order to create an agreed-upon vocabulary for exchanging information.

Ontologies are important because they define a common vocabulary for the people using the system. The idea of ontologies is not a new one, but it is not getting less interesting, on the contrary. Several people are actually researching the area of ontologies actively for the time being.

Some fields already have ontologies defined, there exist an ontology for the telecommunications industry to take an example.

There is an interesting project on the WWW called “The Semantic Web” the idea is to use the Internet as a sort of database, and be able to have agents combine data from different sites into a specialized result. Viewing the WWW as a database raises almost the same issues as the “SSD vs. RDBMS” discussion, the “schema” for the data on the web is undefined, but in order to be able to have for instance agents collect and combine data, there must be some common ground. This common ground can be established by defining ontologies for the relevant areas.

### 2.4.3 Representing Ontologies

As mentioned above, the problem with representing classifications in a way that computers can handle, is essential if ontologies are to be incorporated in the system.

The *is-a* relation is a fundamental part of the classifications, it expresses the relationships between the classes in the classification. Each classification have a top element ( $\top$ ) that is the most general class, this class covers all the other classes, meaning that all the other classes are related either directly or indirectly to  $\top$  by the *is-a* relation.

Some ontologies have a tree structure, they are trivial to represent in a computer, while other have lattice structure. A lattice structure (multiple parents / inheritance) introduces a little more complexity, but not anything alarming.

The easiest way to solve the problem with the classification structure would be to restrict the classifications to a tree structure, but by doing so a lot of flexibility is lost. Tree structures also tend to be bigger than lattice structures when representing structures with natural lattice-like structure.

As mentioned before the `CategoryType` property imposes a lattice-like structure, if it should be restricted to a tree structure it would look like figure 2.10. If a site belongs to the “Mixed” class it would be present in both of the mixed groups. While using the lattice structure yields a representation like the one already shown

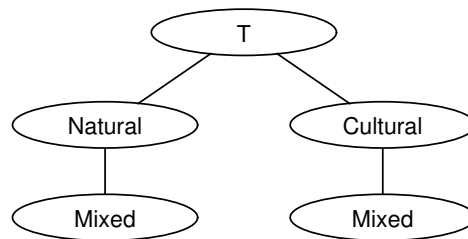


Figure 2.10: The tree representation of the classification based on `CategoryType`.

in figure 2.8 on page 34.

As the lattice structure already is present in the natural relations and we would like to reduce redundancy in the data, we would like our classifications to be able to have lattice structure.

As XML imposes a tree structure, representing the ontologies in XML is a natural choice, the problem being how to solve the issue with the multiple inheritance.

There exist two possibilities to handle the cases that would otherwise violate the tree constraints in XML:

- Duplication of information – like mentioned earlier, the data with multiple parents can be duplicated and a copy placed under each parent. This introduces some redundancy.
- The use of in-document references – it is possible to make references within an XML document by utilizing the `id` and `idref` attributes. Some XML processors<sup>12</sup> can resolve the references automatically, whereas other processors treat them as normal attributes.

Lattice structure can be expressed in XML, by using references instead of “inlining” elements multiple times.

#### 2.4.4 Taking Advantage of Ontologies

Ontologies may seem like a complex construct to use in the system, but they can be implemented in a way, so users will not even notice that they are using a system

<sup>12</sup>XSLT or XQuery processor.

based on ontologies. An obvious possibility would be to generate a dataguide based on the classifications.

A dataguide in this context is a tree structure that the user can navigate just as described in section 1.2 on page 2. It is possible to navigate all the way down to a specific site or to mark several classes as *interesting*. After having marked the interesting classes the user can enter a keyword and the system will return a list of the sites that belong to the selected classes and match the keyword search.

Another option is to have the system propose *similar* sites, based on different classifications. For instance similar sites based on `CategoryType`. These related sites can be found by selecting all the sites that belong to the same class as the site currently being viewed.

It is also possible to offer the user to “expand” his query-scope, if the user has selected `Tropical Forests` and entered a keyword that resulted in zero or very few hits, the system could suggest that the user expands his search scope, this would in this particular case be done by extending the search scope to the “least upper bound” class. If the user only is using the classification based on `CategoryType` the least upper bound class would be “natural site”, but in a more detailed classification there might exist a more fine-grained categorization, allowing the user to expand his search scope without getting all the natural sites back.

Another possibility is to combine search scopes from different classifications, maybe the user selects that he is interested in `Tropical Forests` in the `CategoryType` classification and `Southern America` in the `Location` classification, then an intersection between the 2 selected search scopes would yield a very precise result, while the union of the 2 scopes would provide a much larger (but maybe still relevant) scope.

The advantage of using classifications is easy to spot, the task is to incorporate the classifications without exposing the users to the complexity.

By using SSD / XML it is possible to create a generic application to handle and present the knowledge in the classifications, this would be a hard task to implement in a RDBMS, but it is possible to get away with a relative simple application when using XML.

For instance a simple XML file describing the `CategoryType` classification using the approach with the `id` and `idref` attributes, could look like this<sup>13</sup>:

```
<Classification>
  <Class id="1">
    <Name>Natural</Name>
    <Class id="11">
      <Name>Tropical Forest</Name>
      sites...
    </Class>
```

<sup>13</sup>Please refer to the design description in section 3.2.1 on page 47, where the general format is discussed.

```
<Class id="12">
  <Name>Biogeographical Regions</Name>
  sites...
</Class>
<Class id="2">
  <Name>Mixed</Name>
  ...
</Class>
<Class id="3">
  <Name>Cultural</Name>
  <Class idref="2"/>
  ...
</Class>
  sites...
  .
  .
</Class>
</Classification>
```

## 2.5 Querying the Classifications

The previous sections described how categorizations of sites in the WH site list can be represented more formally as *classifications* using semistructured data. The purpose of the classification documents is to generate *dataguides*, which can be presented on a web page and used to help users to find sites they find interesting.

Previous sections also described how semistructured data can be queried. This section describes how an SSD query language can be used together with the classifications to provide some very convenient search facilities, that will help the visitors to the WH website to find sites.

Because the classifications will be implemented in a semistructured data format (XML), there will be several basic operations available on the classifications, e.g. operations to find children to a node, find all successors to a node or all successors of a given type. The following queries in the classifications are therefore all feasible to implement using standard technologies<sup>14</sup>.

### 2.5.1 Searching Marked Categories

Suppose a visitor to the WH website is presented with a dataguide like the one shown in figure 1.2 on page 4. The user can mark the categories, which he wants to search, by checking the check boxes next to the category names. Then he can enter a keyword in the form in the bottom of the figure and press “Search in selected subjects”. Each of the categories shown in the dataguide, will have a unique id

---

<sup>14</sup>All XML related technologies

associated with it. The query that searches the marked categories, has the following information available from the user:

- A list of unique ids of the marked categories.
- A list of keywords.

This information, together with the classifications and all the WH site-data, is enough to execute the query.

Figure 2.11 shows how a classification might look like. The ellipses denotes the “categories” or “classes” of the classification, and the circles represents the sites. The figure illustrates a scenario, where two categories *C2* and *C6* has been marked by a user – the categories are shown with bold line in the figure. The punctuated bold lines illustrate the part of the classification, that is relevant for the keyword search in the marked categories – the sites *S1*, *S2*, *S3*, *S4* and *S5* are the only sites, that should be included in the search.

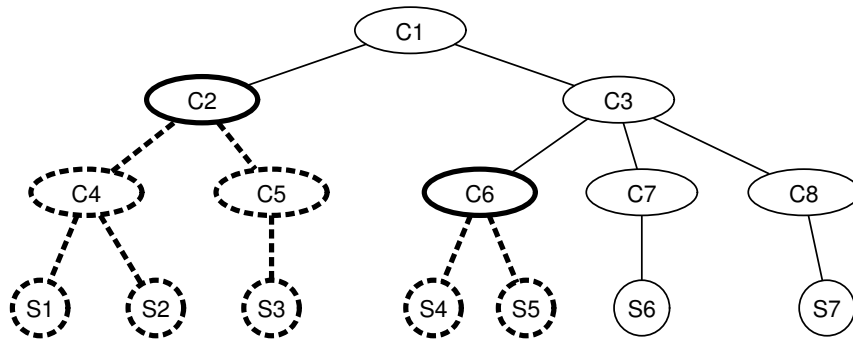


Figure 2.11: Classification illustrating a search in marked categories

Obviously it is easy enough to find all the relevant sites to search, when using a semistructured data model. When the unique ids of *C2* and *C6* are known, then all the successors, hence the relevant sites, can be located simply by using path expressions.

## 2.5.2 Finding Related Sites

In this report the term *related sites* or *similar sites* is used in the following way:

A site *site A* is *similar* or *related* to another site *site B* if they both belong to the same category.

For example all danish sites are *similar sites* to “Kronborg Castle”, because they all belong to the category “Danish sites”.

When a user has found a site that he finds interesting, he might want to find some similar sites, because the site has got some properties he finds interesting. It would

be a nice feature to be able to find other sites that belong to the same category as that site. Suppose the user has located a site  $S1$  that he finds interesting. He then asks for a list of similar sites.

Figure 2.12 and figure 2.13 illustrates two classifications. The interesting site  $S1$  has been marked with bold line in both classifications. The bold punctuated lines

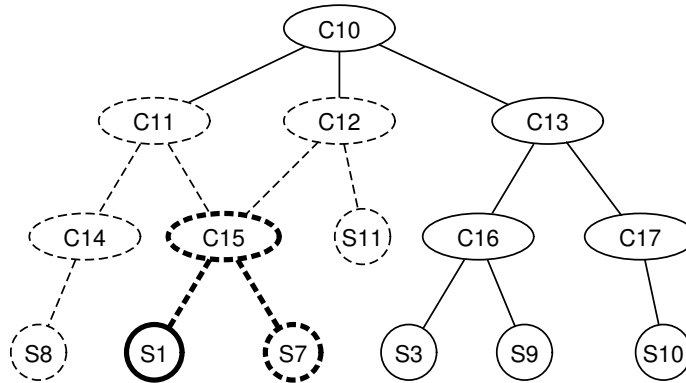


Figure 2.12: Classification showing how to find similar sites

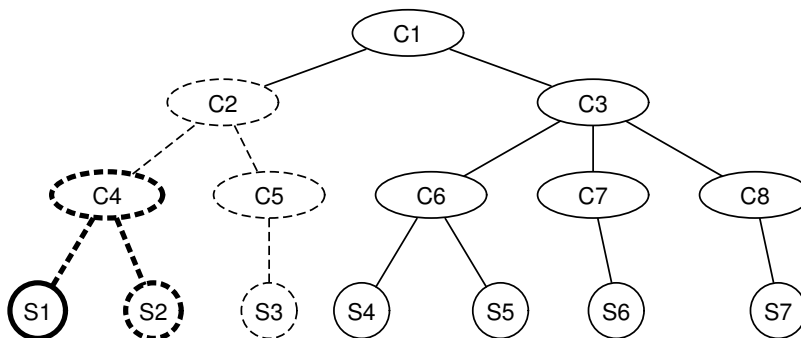


Figure 2.13: Classification showing how to find similar sites

show the part of the classifications which are interesting, when looking for similar sites. Basically all that must be done in order to find similar sites is to “find all the children of the parent of the interesting node (site)”, which is easily done using SSD representations of the classifications. In this case the similar sites are the children of the category nodes  $C15$  and  $C4$  – that is, the site nodes  $S2$  and  $S7$ .

If the user is willing to accept similar sites that are not directly in the same category as the site he finds interesting, then the query for similar sites can be expanded to include the parent nodes of the category nodes  $C15$  and  $C4$ . This is illustrated with the thin punctuated lines in the two figures.

### 2.5.3 Finding the Best Match

Naturally the visitors at the WH website should have the possibility to mark interesting categories in more than one classification and make a keyword search in all the classifications. Such a query can be executed by searching each of the classification one at a time, but the same site will probably often appear as a match in several of the classifications. This imposes a need to order the matching sites by how good a match they are.

There are several issues that should be considered when ordering the matching sites – suppose a user has marked categories in several classifications and entered several keywords to search for. The following statements seems obvious:

- A site that matches two keywords, is better than a site that only matches one keyword.
- A site that is matched in two categories, is a better than a site which is only matched in a single category. This is true regardless of which classification the categories belong to. A site might be a member of two categories in the same classification.

It seems like a good idea to apply some “hit points” to each site while performing the queries. The pseudo code below defines how such a hit point system can work. The search function takes a list of keywords, a list of ids for the categories that the user marked and a list of all the classification as arguments:

```
function search(keyword-list,markedCategoryID-list,classification-list) {
  for each classification in classification-list
    for each category in classification.categories
      for each markedCategoryID in markedCategoryID-list
        if category.ID=markedCategoryID
          then
            for each site in category.sites
              for each keyword in keyword-list
                if site contains keyword
                  then assign 1 point to site
                else do nothing
            else do nothing
}
```

The best hits are then the sites with most points.

### 2.5.4 Presenting the Query Results

When the WH system has finished making a query it has to present the result to the user. One possible way to present the result is in the form of “result dataguide”. Suppose the result of the “find similar sites” query (see the figures 2.12 on the preceding page and 2.13 on the page before) should be presented to the user. The result can be expressed as a lattice as shown in figure 2.14 on the facing page. This lattice can be transformed to a dataguide the exact same way, as the classification



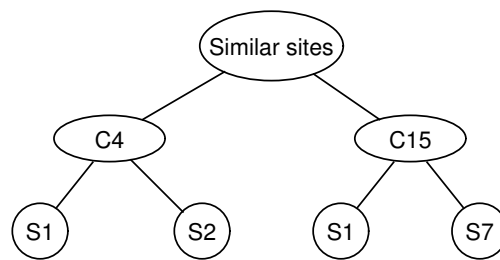


Figure 2.14: Lattice representing a search result

lattices are transformed to a dataguides. There are several advantages to this result representation as opposed to simply presenting the result as a list of sites:

- If the user gets too many results from a query, he might want to make another query that only searches sites which were results of the original query. The lattice in figure 2.14 can be used directly for such a query, because it has the same structure as the original classifications.
- A “result dataguide” contains useful information about which categories a site belongs to – a simple list of matching sites does not. Hence visitors to the WH site would probably appreciate the dataguide structure.
- The functionality used to generate the “search dataguides” (as shown in figure 1.2 on page 4) and the “result dataguides” are the same. This can make the implementation a little simpler and more elegant.

The main disadvantage of displaying the query results as dataguides is, that it is more complicated to build the result lattice, hence it requires more processing power. This issue should be considered, when deciding on how to present query results to the user.

## 2.6 Summary

- The advantages that semistructured data has over traditional relational data are:
  - SSD allows new attributes to be added to existing structures.
  - SSD allows structures to be nested into arbitrary depth.

Note that this is only an advantage, if the data in question requires some of these properties. If the exact structure of the data is known, and the types of data types of attributes are known, there is no reason for choosing SSD over traditional relational data.

- XML is an excellent format for storage of SSD.
- *Classifications* can be stored as XML data – it is not trivial to store classifications in traditional relational data, because the classifications are lattice structures which require nesting.

- Information about the sites in the WH list are also well suited for storage in XML, because it is convenient that new information about sites can easily be added.
- XML Schemas can be used for putting some constraints on XML data. XML Schemas can be used to validate if an XML document complies with a given structure, and whether elements/attributes in the document has some required type.
- XPath and XQuery can be used for querying XML data. There are several programs available that implements these technologies.
- When classifications are stored as SSD they can be used for making convenient search facilities, which can help people find sites that they find interesting.

## Chapter 3

# Application Modeling and Design

*This chapter contains a description of the WH system that needs to be created, in order to be able to take advantage of classifications for querying the WH data.*

*First section is a description of the system that need to be created. It turns out that two components are needed:*

- 1. A system running on the server, handling queries, presentation of the data and the presentation of query results.*
- 2. A client application for creating classifications.*

*The next sections describes the modeling of the XML documents describing the WH data and the classifications.*

*Finally the modeling and design of the WH System, followed by the modeling and design of the application for creating classifications.*

### 3.1 System Description

The system, that is to be implemented, should be able to help users narrow down World Heritage Sites that they find interesting, or maybe suggest other (relevant) sites, that might also be of interest to the user. Furthermore it should be possible to create classifications in the application.

In order to reach as many users as possible, the system to query classifications for locating interesting WH sites, should be a WWW system like the existing WH Web Site is.

Creation of classifications needs some graphical support, in order to give a good overview of the classification being created. This is not easy to accomplish in a web interface. Something could be done using applets, but a lot of trouble would occur. Instead a client side application should be created, and it should be able to interact with the WH System server over the Internet.

A simple diagram describing the different software systems can be seen in figure 3.1.

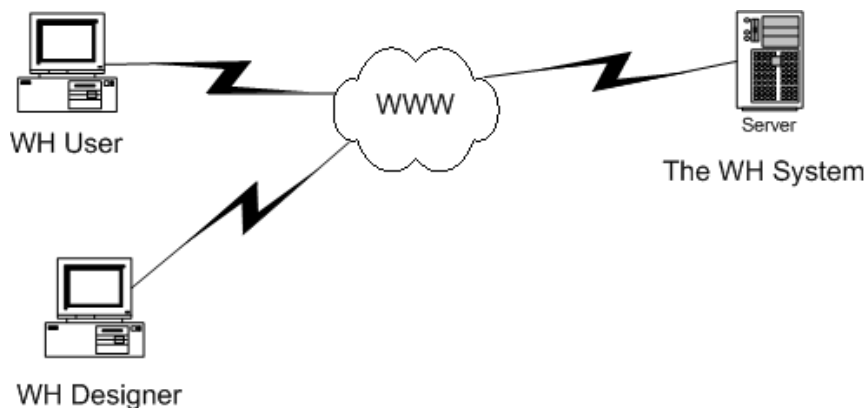


Figure 3.1: System diagram

The diagram shows the different software systems that need to interact in order to have a working WH System.

**WH User** A visitor on the WH Web site, looking for interesting sites. The only piece of software that this user need is a WWW browser.

**WH Designer** A person with relation to WH, who has credentials to alter classifications on the server. The designer creates classifications locally, and uploads them to the WH System through the application used for creating classifications.

**WH System** A Web server and a database for storing WH data. This is the system that generates dataguides based on classifications, performs queries in the classifications and presents the query results to the **WH User**.

The backbone of the system will be classifications implemented in XML. There are several possibilities for storing XML data, it can be stored in ordinary files, in a native XML database e.g. Xindice or in a RDBMS. Since the data will be XML data, an XML database is used.

The format of the data has to be specified, this is to be done by creating a model of the data, and extracting the schemas to be used in the implementation.

The XML data need processing before it can be presented to the user. Some of this processing may take quite a while, and this is not needed to be done when the data is requested from the user. It is therefore necessary to determine which data views that can be extracted beforehand, and which views that has to be processed dynamically.

Another issue with XML data is, that some of the aforementioned storage forms are not very good at handling updates, so this issue will need to be looked into. Along with the problem of updating data, the maintenance of the extracted views also has to be treated, which views need to be updated and how often.

It is possible to associate binary files to sites, the files could be sound, video or pictures describing the site in some way. The system needs to be able to handle this kind of data also.

A tool for creating classifications is modeled in this section. The tool should help people design classifications, administer the classification and data documents on the WH system server. Administration of the classifications and data documents include common actions as upload, download and removal of documents in the system.

## 3.2 Specification of the XML Documents

### 3.2.1 Modeling Classifications

As described in the previous sections, classifications play a major role in the system. In the detailed problem description a couple of possible classifications were shown on a screen shot (as dataguides), but the question is: Can we find a general classification format, that enables us to cover both existing and future classifications?

By having a general classification schema, it is possible to create a generic presentation layer that can show all the classifications, that may be incorporated in the system. This will result in a flexible system that is easy to maintain.

Even though this project tries to take advantage of semistructured data, the classifications need to be put into a quite strict schema. This constraint is necessary, because we would like to be able to take as much advantage of the classifications as possible. If we have no knowledge about the structure of classifications, it is hard to make a really usable user interface for querying, hence the schema for the classifications is fixed.

What are XML classifications and how are they related to the WH system?

*A classification is a lattice structure expressing generalization / specialization relationships. Each "Node" contains a name, description and some keywords describing it. Additionally each node can contain "references" to entities in a data document. For performance issues each "reference" contains a `DisplayName` that gives a short description of the entity referenced. This enables the classification to "stand alone" ( it is not necessary to look up the referenced entities in the data document in order to show a dataguide).*

Section 3.2.4 on page 54 shows an example of how a classification and a data document is linked.

The schema is shown in figure 3.2 on the next page, and the entities in the diagram are described in detail in the following sections. The XML Schema for the classification documents can be found in appendix G on page 143.

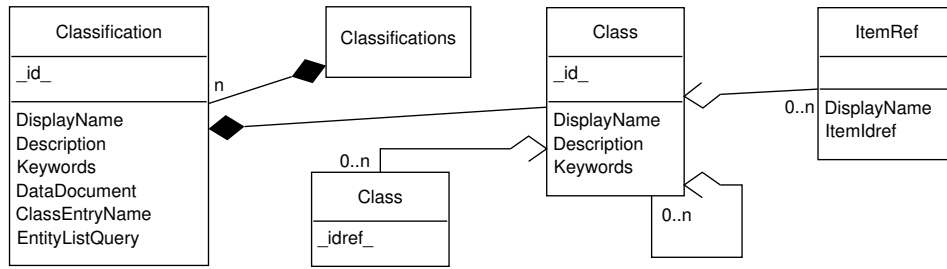


Figure 3.2: Lower-bound schema for the classifications

### The Classifications Element

This element contains all the classifications in the system. This element is not included in a classification definition, but is used to represent the collection of all the classifications in the system.

It is included in order to show how the WH system keeps track of its classifications.

### The Classification Element

*Root* element of a classification. This element contains the information that is necessary for describing a classification.

In order to handle the classifications in a uniform way, on the server and in the ClassificationDesigner, it is necessary to define the following elements:

**DisplayName** - The name that will show up in the presentation layer on the server.

**Description** - A short description of the classification, used to describe the classification subtree in the dataguides. Could also be used to locate a classification in a system with a large number of classifications.

**Keywords** - Keywords describing the classification, could also be used to locate interesting classifications in systems with many classifications.

**DataDocument** - The URI of the document containing the elements to be classified.

**ClassEntryName** - The name of the elements in the data document that is going to be classified - in the WH case the elements are called `<Site>`. The elements must contain an `id` attribute, enabling us to look them up easily.

**EntityListQuery** - The XQuery used to query the `<DataDocument>` document in order to get the `<ClassEntryName>` items that the classification is indexing.

**Class** - The root ( $\top$ ) node of the classification structure, the format of this element is described in the next subsection.

`<DataDocument>`, `<ClassEntryName>` and `<EntityListQuery>` are not needed in the WH system, but the tool for generating classifications need these values for editing existing classifications.

### The Class Element

Class elements contain information about the classes and their lists of references to items (sites in the WH system) belonging to the respective classes. Classes can contain other classes or references to classes – this is the way that the structure of classifications is defined.

**DisplayName** - The name that will show up in the presentation layer.

**Description** - A description of the class.

**Keywords** - Keywords describing the class.

**ItemRef** - A reference to the `id` of a class entry, name of the element to be referenced is specified with the `<ClassEntryName>` in the `<Classification>`.

### The ItemRef Element

The element containing the name of the reference and the `id` used to locate the referenced item.

**DisplayName** - The name that will be displayed in the presentation layer, is constructed from one or more simple elements in the `<ClassEntryName>` elements in the data document.

**ItemIdref** - The value is a foreign key in the data document described in the `<Data-document>` element.

For example could this element contain the value extracted from:

`<Site><Name>...</Name>...</Site>` in the WH system. An `<ItemRef>` referencing the example site in section 2.1.3 on page 16 could look like this:

```
<ItemRef>
  <DisplayName>Roskilde Cathedral</DisplayName>
  <ItemIdref>site259</ItemIdref>
</ItemRef>
```

### 3.2.2 Modeling the Site Document

Figure 3.3 on the following page shows the schema for the site document containing all the information about the different sites. The schema is inspired by the schema to the relational database currently used in the official World Heritage prototype application can be found in appendix H on page 145. The different “entities” or “complex elements” are explained in the following subsections.

#### DataCollection

This is the root element, which is used as a wrapper for the other elements.

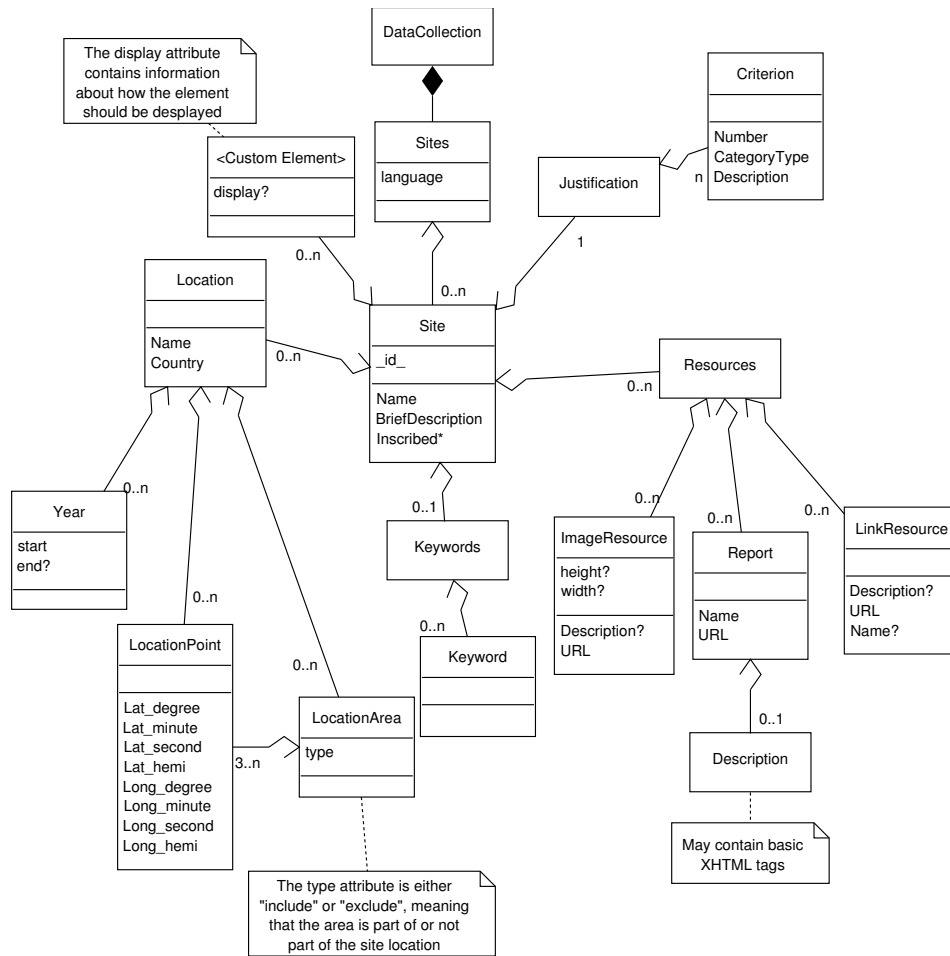


Figure 3.3: Schema for the site document

## Sites

For each language there is a `<Sites>` element containing all the site information for that language.

**language** – This attribute describes which language is used in the site information. Currently there is only site information in English.

## Site

Contains all the information about a single site.

**id** – unique id of this site.

**Name** – The name of the site.



**Number** – The number of a site. In the original WH website, all the sites have a number that may be used to link to old documents containing information about the site.

**BriefDescription** – A brief description of the site.

**Inscribed** – The year(s) this site was inscribed in the WH list.

### Resources

This element is very important. It contains information about all the internal and external resources linked to a site.

### ImageResource

Contains information about an image that is linked to a site.

**height** – The height of the image, when it is displayed to the user.

**width** – The width of the image, when it is displayed to the user.

**Description** – A description of this image.

**URL** – Describes the location of the image. The image may exist only on an external website or on the local web server.

### Report

All the sites have at least one report associated with them. The report contains information about the site, e.g. why the site has been inscribed in World Heritage.

**Name** – The name of this report. This name will be displayed on links to the report.

**URL** – The location of the report.

### Description

A description of a report – it may contain basic XHTML elements for formatting the text.

### LinkResource

This is the most general resource and it is simply a hyper link to something, e.g. a link to the homepage of Roskilde Cathedral.

**Description** – A description of this link resource.

**URL** – The location of the resource.

**LinkName** – A name of the link. This name will be displayed on hyper links seen by the user.

**Location**

Describes the location of a site. The location of a site is described by a collection of points and areas.

**Name** – Name of this location.

**Country** – Location is in this country.

**Year**

Each location of a site may be created in different years. This element describes when this location was created and possibly ended.

**start** – Start year.

**end** – End year.

**LocationPoint**

Represents a single geographical point.

**Lat\_degree, Lat\_minute, Lat\_second, Lat\_hemi** – Describes the exact latitude of this point.

**Long\_degree, Long\_minute, Long\_second, Long\_hemi** – Describes the exact longitude of this point.

**LocationArea**

Describes a geographical area. An area must contain at least three `LocationPoints` elements in order to describe an area.

**type** – The type is either “include” or “exclude”. The location of a site is found by taking the union of all “include” areas and then removing the “exclude” area and finally adding all the single points which are part of the location.

*Note that the location areas are not be used for anything in the WH System. The `LocationArea` element is included, because it would be possible to describe areas this way.*

**Justification**

Contains information about why a site was inscribed in the World Heritage list.

**Criterion**

A criterion is part of the justification of why a site is inscribed in the WH list. A criterion describes one reason why the site was inscribed.

**Number** – Unique id of criterion.

**CategoryType** – “N” for natural or “C” for cultural.

**Description** – Either a general description of this criterion or a description of why the site fulfills this criteria.

**Keywords**

A site may have number of keywords associated with it. The keywords can be used when making keyword searches. A collection of keywords is enclosed within this element.

**Keyword**

Contains a single keyword.

**3.2.3 Generating XML Data from Existing Data**

In order to implement a usable WH system based on XML data, it is necessary to generate the XML data somehow. Because there is a great amount of data, it obviously cannot be generated by hand – it has to be done more or less automatically.

The “official” WH prototype website uses a *MySQL* relational database<sup>1</sup> for storing the data, and fortunately we received a copy of this database. *MySQL* is able to return the results of SQL queries as XML data. The easiest way to create XML documents, that complies with the schemas described in the previous chapters, is first to dump the content of each table in the *MySQL* database into separate XML files. The next step is to write *XQueries*, that uses these files to generate the document. This procedure was used to generate all the needed WH site data, but it was necessary to make some small adjustments to the XML documents, before they were ready for use in the WH system.

The following XML documents were created:

- A document with a *Classification by geographical location*. This classification contains all the information about geographical location, that is available in the *MySQL* database, except for the coordinates. The schema for this document type is shown in figure 3.2 on page 48.

---

<sup>1</sup>The schema is specified in appendix H on page 145

- Another classification: *Classification by Miscellaneous Categories*. This document was generated using the data from the tables `category` and `subcategory`.
- A site document (see figure 3.3 on page 50) with information about all the sites in the WH list. The document contains most of the information, which is also in the MySQL database. The keywords from the database were left out, since they are not really needed, and the precise location with coordinates is also not present in the XML document.

The MySQL database contains some additional data, but the above mentioned three document is perfectly adequate for implementing a fully functional WH system.

### 3.2.4 The Connection from Classification to Data Document

As mentioned earlier the `<ItemIdref>` in the `<ItemRef>` element is used to identify the “items” that belong under the given `<Class>`. An example of what this looks like in the actual documents, can be seen in figure 3.4. The big ellipsis indicates the item that references the actual element in the data document.

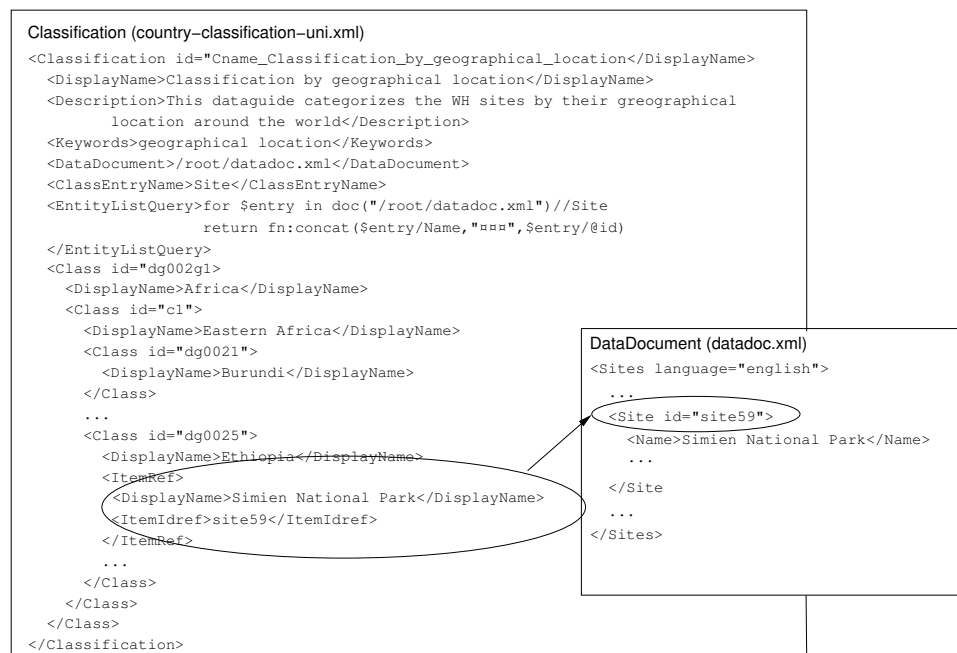


Figure 3.4: Connection between a Classification and a Data Document

## 3.3 WH System Model

The *WH System* is the server application that contains the web application, the database and all the programming logic for searching the sites and classifications.

### 3.3.1 Actors

This section describes the different actors who will interact with the WH System. To make things simple, there are only two different actors:

**User** The *user* is simply a person who is interested in the World Heritage, and browses the WH web application to find information about different sites. The user can search the site list and view info about the sites.

**Administrator** The administrator is responsible for administer any changes to the WH System. The only people who needs this kind of functionality is classification designers. The use cases for this actor are therefore described in the use case descriptions of the ClassificationDesigner in section 3.6.1 on page 68.

### 3.3.2 Use Cases

Figure 3.5 on the next page shows the use cases for the actor: *user*.

#### Use Case Descriptions

**Browse dataguide** The user used the dataguide to narrow down the search scope and ends by making a mouse click on some site, which he finds interesting.

**Search marked categories by keyword** The theoretical aspect of this use case was described in section 2.5.1 on page 39.

The user browses a dataguide and marks some classes, which he finds interesting. Then he enters some keywords in an HTML form, and pushes “search”. the system returns a list of all sites that matches the keywords. The system indicates how good a match the different sites are, e.g. by placing the best matching sites in the top of the page displayed to the user or assigning some “hit points” to each match. The sites with most hit points would then be the best matches.

**Search marked categories by keyword, combining several classifications** The user has the possibility to search several classifications at the same time. The best matching keywords are those in all the classifications. Less good matches are present in only some of the classifications.

**Search sites by keyword** The system is able to search a list of sites for some specified keywords. This functionality should not be used directly by the user.

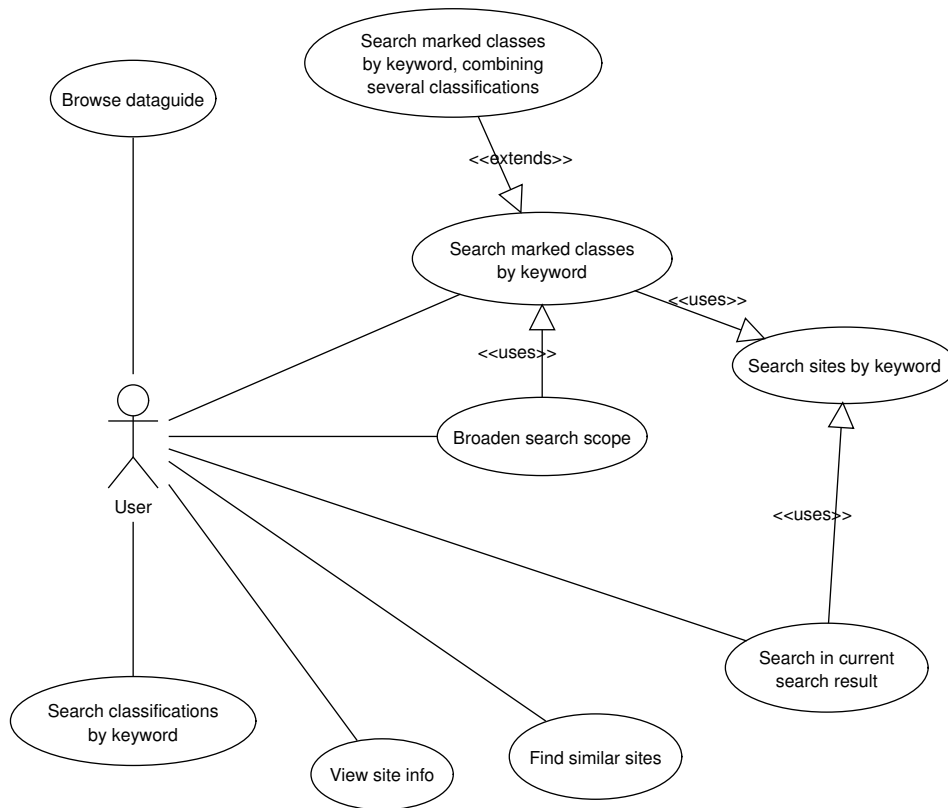


Figure 3.5: Use cases for the user

**Search in current search results** If the user has made a search that returned too many hits, he can choose to make a keyword search in the current hits.

**Find similar sites** When a user has got some search results, he can choose to find sites similar to a given site. The definition of “similar sites” was described in the top of section 2.5.2 on page 40, where the meaning of this use case was also described from a more theoretical viewpoint. This functionality searches all classifications and returns categories where the site is present.

**Search classifications by keyword** The user can choose to make a keyword search in the classification documents. This facility searches the classifications and returns all categories which matches the keyword.

**Expand search scope** When the user receives few or zero results in a search, the user can choose to expand the search scope, meaning that the system searches the parents of the categories chosen by the user.

### User Case Interactions

The following use case interactions assume, that the user has already entered a “Search” page in the WH application. This page presents a list of all the dataguides, and the user can browse these categories. There is a text field for making a keyword search in the categories which the user has marked, and a text field for making keyword search in the dataguides (without searching the site documents).

Actor Action	System response
<b>Browse dataguide</b>	
1. The user browses the dataguides he finds interesting. He finds a site that he wants to know more about and clicks the link with the mouse.	2. The user receives a new web page containing the detailed information about the chosen site.
<b>Search marked categories by keyword</b>	
1. The user browses a single dataguide, and marks interesting categories by checking a check box next to the category names. Then he enters some keyword(s) in a search form and pushes “Search”.	2. The system searches the sites under the marked categories and presents links to sites, which matches the keyword(s). The links may be presented as a simple list of sites, where the best matches are positioned in the top of the list, or as another dataguide, which can be used for additional searching.
3. The user makes a mouse click on a site that he finds interesting.	4. The user receives a new web page in his browser. It contains the detailed information about the chosen site.
<b>Search marked categories by keyword, combining several classifications at the same time</b>	
1. The user browses several dataguides and marks interesting categories by checking some check boxes. He enters some keywords in the search form, and pushes “Search”.	2. The system registers that categories in different dataguides have been chosen. It searches categories in all the dataguides as described above in “Search marked categories by keyword”

**Search in current search results**

1. The user has already searched the system for sites, but he thinks that he received too many hits. He decides to search the current search results by keyword. He enters some keyword(s) in a search form, and pushes “Search”
2. The system makes another keyword search, but only in the list of sites from the previous search. The result is presented to the user as previously explained.

**Find similar sites**

1. The user has chosen to view the details about some specific site. He chooses to “find similar sites” (or “find sites in same categories as current site”) by pushing a link or a button.
2. The system locates the specific site in all the different dataguides. It finds all the categories, where the site is present and creates a list of these categories and all the sites that belong to these categories. This list is presented to the user – the site names are links which the user can click in order to see the details about the sites.

**Search classifications by keyword**

1. The user enters some keyword(s) in a form for searching the classifications (dataguides). He pushes “Search dataguides”.
2. The system searches the description and keyword fields of all the categories in all the dataguides. All the categories that matches the keywords are presented to the user as a kind of “mini dataguides” that the user can fold and unfold
3. The user browses the “mini dataguides” and clicks a site that he finds exciting
4. As explained earlier a new web page with the site details is displayed.

**Expand search scope**

1. The user has already made a keyword search in some marked categories, but thinks that he received too few hits. He clicks a link or button saying “expand search scope”
2. The system finds the parents of all the categories that the user chose in his previous search. Then the system uses the keyword from the user’s previous search to search all the parents. The search result is presented to the user as already explained.

**3.4 WH System Design**

The WH system is a J2EE application and this has some major advantages when it comes to making the design. It is possible to make use of a design pattern: The



*Model View Controller* (MVC) pattern, which is very suitable for J2EE applications – especially web applications. The use of this design pattern eliminates the need for an in-depth class specification, because many of the classes needed to make the application will have a “standard” structure, and the different classes will be connected in a “standard” way. The next section briefly describes how the MVC can be used in a J2EE application.

### 3.4.1 The Model View Controller design pattern

The purpose of this pattern is to separate business logic from data presentation. The pattern consists of three components:

- **The Model** contains the “business logic” for the application, which means that it contains programming logic for interacting with any database back end, and it generates data for the view. It does *not* define how the data should be used or presented for users.  
In the J2EE environment the model consists of *Enterprise JavaBeans* (EJB) containing the logics and *JavaBeans* containing data<sup>2</sup>.
- **The View** is responsible for presenting the model data to the client application. In the J2EE environment the views are JSP pages which use the model data for generating HTML.
- **The Controller** intercepts service requests from clients. It determines which changes should be made to the model depending on what kind of request it receives.  
J2EE web applications typically implements the controller as a *Servlet*.

Figure 3.6 on the following page illustrates the principle of the model view controller. The figure shows the different stages that a web application goes through while processing some service request:

1. The client sends a service request to the application server. In the WH system, the “service request” is a HTTP request.
2. The controller intercepts the service request and determines what actions to take. It calls the appropriate business methods in the model, which updates the model data.
3. The controller then forwards the request to a view, that is associated with this particular request.
4. The view generates a presentation of the available model data. This is typically a HTML page.
5. The generated presentation is sent back to the client in a service response.

It is important to notice that the model shown in the figure, is an abstract model, meaning that an implementation using the MVC design pattern does not necessarily contain components which are equivalent to the components the figure. The

<sup>2</sup>“Enterprise JavaBeans” and “JavaBeans” are two completely different type of components that unfortunately have very similar names

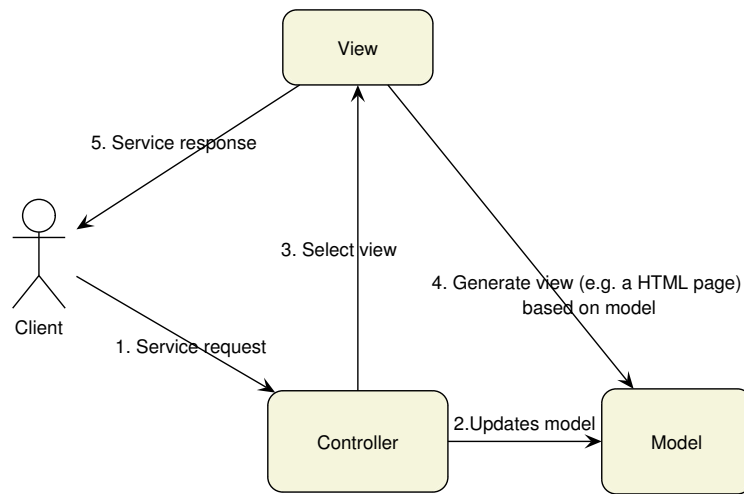


Figure 3.6: Model View Controller design pattern

abstract components will typically be implemented as several different software components.

It is quite simple to make a design for the WH system based on this general design pattern, when using the J2EE technologies.

### 3.4.2 Design of the WH System

Figure 3.7 on the next page illustrates the design of the WH System. The easiest way to explain how this works is by example – the numbered arrows in the figure illustrates the different states the application goes through while processing a request from a user<sup>3</sup>:

1. The client has opened the search page of the WH web application. He enters some keywords in a search form and pushes the search button. This action launches a HTTP request for the URL *http://www.world-heritage.org/result.html*. The keywords are included as POST variables in the request.
2. The *Front Controller Servlet* intercepts the HTTP request. The front controller has a map which maps each possible requests into a *Request Handler*. A request handler is a Java class, that is responsible for handling one specific request. The request for “result.html” is mapped to the request handler “ResultRequestHandler”, hence the front controller forwards the HTTP request to this request handler.
3. Because the request handler is responsible for only one HTTP request, it

<sup>3</sup>This is just an example – the URLs and filenames do not necessarily match those in the real application

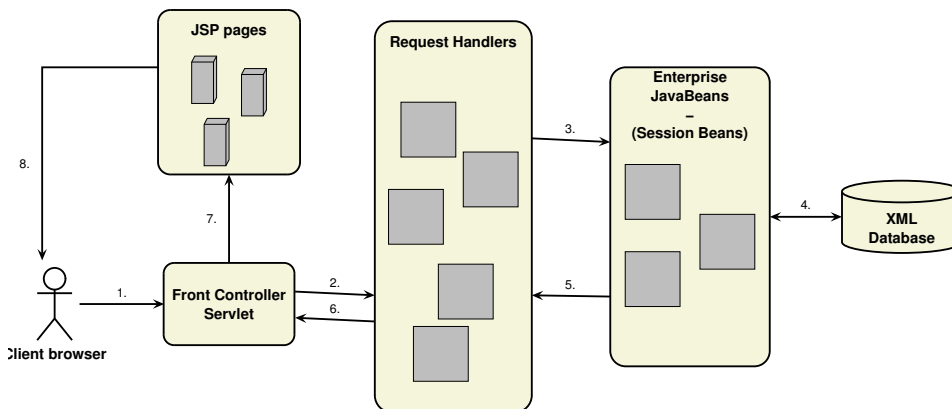


Figure 3.7: Design of the WH web application

knows exactly what to do. It makes a JNDI<sup>4</sup> lookup for an Enterprise JavaBean (EJB). Then it calls the appropriate search method in the EJB.

4. The EJB contains the *business logic* which in this case means, that it implements the functionality for performing the keyword search requested by the client. The EJB searches the XML database and generates data (probably information about some sites that matched the keywords), that should be returned to the user. Notice that the EJB does *not* specify *how* the data should be presented to the user.

Details about how the XML database can be searched is explained in a section later on.

5. The result is returned to the request handler – possibly as a JavaBean<sup>5</sup>.
6. The request handler now prepares the result for a JSP page. If the EJB did not create a JavaBean with data, it is created now. The JavaBean is added to the HTTP request as an attribute. The request handler returns the name “result.jsp” to the front controller.

Typically a request handler can return with two possible outcomes – it can return the name of whatever JSP page is responsible for presenting a result to the client, or it might return the name of an error page if something went wrong while handling the request.

7. The front controller forwards the HTTP request to the JSP page “result.jsp”, which is responsible for presenting the data.
8. The JSP page extracts the JavaBean from the request. Then it generates an HTML page based on the data. Finally the resulting page is returned to the user.

<sup>4</sup>JNDI is a service for looking up Java objects based on some unique name, quite similar to how one can use DNS to translate a domain name into an IP address

<sup>5</sup>Again it should be emphasized, that a JavaBean is simply a class instance containing data and *no logics*, so do not confuse it with an EJB

The experienced J2EE developer has probably noticed that the EJB's are so called *Session Beans*, which are directly connected to the database. This is normally not the best way to interact with a database. Normally one would put in an extra layer in the model between the session beans and the database. This layer would consist of *Entity Beans*, so that the session beans would communicate with the entity beans and the entity beans would be the only interface to the underlying database.

The problem with entity beans is, that they are designed as representations of relations in a relational database. This is a useful feature in applications using RDBMS for data storage, but it is completely useless when using an XML database, and this is why the session beans have to communicate directly with the database.

It is easy to recognize the MVC pattern in the design. The controller consists of the *front controller servlet* and the *request handlers*. The model is the EJB's and the JavaBeans. Notice that the JavaBeans are not components the same way that the front controller and the EJB's are – they are simply temporary containers for temporary data. Finally the views are the JSP pages.

An excellent quality in the design shown in figure 3.7 on the preceding page, is the very clear division into components, where each component is responsible for a restricted part of functionality. Because of this, it is very easy to understand how the program works and what “kind” of code to expect in the different components. It also makes it extremely easy to expand the application with additional functionality. Suppose that the application should be extended with a new search facility of some kind. The programmer would have to go through the following steps (not necessarily in this order):

1. The programmer creates a new static JSP/HTML page containing a search form, where the user can provide his search criteria.
2. If the result of the new search fits into an existing data model (JavaBean), then this is used for the result. Otherwise a new JavaBean is created for holding the data.
3. The new search method (the business logic) is implemented in an existing EJB.
4. A request handler is created for extracting the user's search criteria from the HTTP request and calling the appropriate search method in the EJB.
5. If the result fits an existing data model, it will also fit an existing view (JSP page) and this can be used for presenting the result. Otherwise a new JSP page is created for presenting the result for the user.
6. Finally the new type of request has to be mapped into the correct request handler in the front controller.

This may seem like a lot of things to go through, just to add a little functionality, but it really is not. All the steps, except the implementation of the business logic in the EJB, are trivial. The beauty of it is, that it is not necessary to alter any existing code to make it work. It is only necessary to add some code. It is especially nice,

that the request handler is implemented as a whole new Java class, so you can easily find and get an overview of the code, that handles a specific request.

### 3.4.3 Introducing XQueries in the Application

An essential part of this project is the use of XQuery for making queries in the XML data. The evaluation of XQueries will be made by an XQuery engine implemented as some Java packages, and these packages will be included in the EJB, that is responsible for all the search facilities. But where should the XQueries reside? Normally when using XQuery, one would write the queries in a file and store it on a hard drive together with the XML documents, that should be queried. Obviously this is not possible, because EJB's do not have access to the hard drive, and even if it was possible, it would not be a desirable solution – it would be “messy” to use a lot of files on the hard drive.

Another solution is to code the XQueries directly into Java classes as text strings. This is definitely possible, but it would be even more messy, because there are many XQueries. In order to make it work, all characters that Java consider to be “special characters” should be escaped. For example all line breaks had to be replaced by: `\n`. Another disadvantage is that it would be much harder to effectuate any changes in the XQueries. If the XQuery is inside Java classes, then the entire Java application must be compiled and redeployed before the changes take effect.

The best possible solution seems to be to put the XQueries in the XML database together with all the XML documents. This ensures that the XQueries are stored in a structured manner, and they are easily available for the Java classes, that needs them. This does, however, mess up the elegant MVC design explained in the previous section, because most of the business logic will reside in the database and not in the EJB. The database will actually also be responsible for some of the *view* part of the MVC pattern, because it will contain XSLT style sheets used for transforming XML data into HTML used in the web pages. These facts might take away some of the “beauty” of the design, but when using new and non (yet) standard technologies such as XQuery, it is necessary to compromise.

### 3.4.4 Database Design

The design of the XML database is quite simple. The database must store the classifications, the site data, the XQueries and some style sheets for transforming XML data into HTML. Figure 3.8 on the next page illustrates the database design. The figure does not show the structure of the classifications and the site data, this was explained earlier and is illustrated in figure 3.2 on page 48 and 3.3 on page 50 respectively. The XML structures containing XQueries and XSLT style sheets are very similar and very simple. The `<Name>` elements are used as unique identifiers and the `<Data>` elements contains the XQueries and style sheets inside a `CDATA`

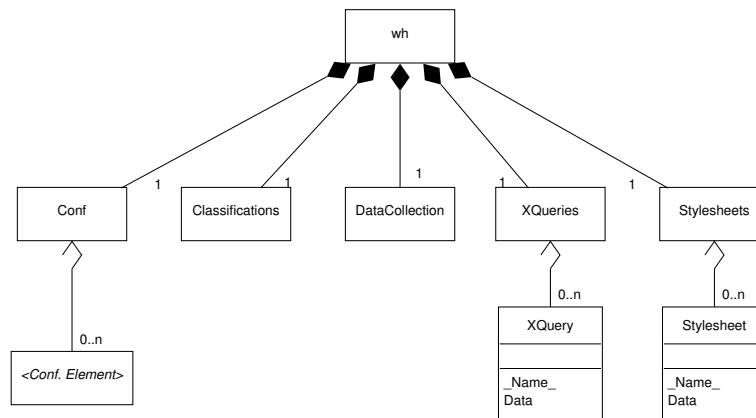


Figure 3.8: Design of the WH XML database

section. The `CDATA` section is necessary because the XML parser should not attempt to parse the XQueries or the style sheets.

The database also has a collection called `conf`. This collection can be used for any configuration parameters.

### 3.5 WH System Class Specification

This section describes the Java classes and JSP pages that makes up the WH system. This is not an in-depth specification of all classes and their methods and attributes available in the system, but it gives an outline of all classes that are important for the desired functionality described in the use cases earlier.

The classes are described from top to bottom, meaning that the classes closest to the users of the WH system are described first.

#### 3.5.1 JavaBeans and JSP Pages

The system should contain the following JSP pages, that are important for the functionality:

- **search.jsp**. Contains the search form for making an advanced search in the WH list. This is where the user can browse all the dataguides, select some interesting categories and search by keyword.
- **simpleSearch.jsp**. Contains the search form for making simple searches in the database. This page is basically just a text field, where users can enter some keywords, and a search button.
- **dgSearch.jsp**. Page for searching the classifications for categories e.g. *Northern Europe* or *Asia*.

- **simpleSearchResult.jsp**. Displays the result of a simple search. It is simply a list of matching sites, with links to more information about each site.
- **siteinfo.jsp**. Shows information about a chosen site. Shows links for “sites in same category as chosen site”.
- **similarSitesResult.jsp**. Shows sites which are in the same categories as a previously chosen site. The sites are displayed as a dataguide. The user can choose to search this dataguide by pressing a link.
- **result.jsp**. Displays the result of an advanced search. Results are displayed as either a dataguide or a list. The page also contains a form for searching the current result. The *current result* may be the result of a search for similar sites, a search for categories in a classification or, of course, an advanced search.

In addition to the above JSP pages, there are a few other uninteresting pages – e.g. a welcome page.

The JSP pages are strongly connected to JavaBeans, because JavaBeans are the JSP pages only source to data. Each JSP page, except the static JSP pages, uses one or more JavaBeans for generating the HTML pages that are returned to the user. The important JavaBeans in the WH system are:

- **DataGuideBean**. Represents a single dataguide. It is used by *search.jsp* for creating the advanced search form.
- **SearchResultBean**. Represents a search result. The search result should be presented both as a dataguide and as a simple list, when available. The bean is used by *result.jsp*.
- **SimilarSitesBean**. Represents the result of a search for similar sites. Used by *similarSitesResult.jsp*
- **SimpleDataBean**. Represents the result of a simple search in the WH site list. It contains a simple list with the search results. It is used by *simple-SearchResult.jsp*.
- **SiteBean**. Holds detailed information about a single site. Used by *site-info.jsp*.

### 3.5.2 Front Controller Servlet

The front controller servlet is a single Java class called *FrontController*. Its only responsibility is to pass HTTP requests on to the correct request handlers, and to forward requests to JSP pages.

### 3.5.3 Request Handlers

The request handlers are not that important for the class specification. Which request handler that are required, are implicitly given by the types of requests that are received in the front controller.

### 3.5.4 Enterprise JavaBeans

The EJB's are an essential part of the WH system, because they execute the programming logic for all search facilities. There are two types of clients to the WH System EJB's: ordinary users, who use the WH website to search for sites, and the classification designer program, which must be able to update classifications in the database.

It seems obvious to make two EJB's – one for the “web users” and one for the classification designer client application.

#### SearchSessionEJB

This bean implements all the search facilities, hence it should have methods to:

- Perform an advanced search based on keywords and categories, that the user selected in the dataguides.
- Perform a simple search based on keywords alone.
- Search the classifications for categories, based on keywords.
- Finding sites that are in same categories as a given site – the search uses the id of the given site to find the other sites.
- Getting information about a site based on the site's id.
- Perform a search in a “result dataguide”. This is the search facility that the user of the WH system sees as “search current result by keyword” or something similar.
- Expand search scope.

*Note that this functionality is not implemented*

#### DBUpdateSessionEJB

This EJB implements the facilities required by the classification designer client. It is required to have the following methods for managing XML documents in the database:

- Insert `Classification` in database.
- Insert `DataCollection` in database.
- Remove `Classification` from database.
- Remove `DataCollection` from database.
- Get list of all `Classifications` in the database.
- Get list of all `DataCollections` in the database.
- Setting “update” flag in XML database. This flag indicates that changes have been made to classifications in the XML database.

Additionally there should be some method for authenticating users of the classification designer.



## 3.6 ClassificationDesigner Model

Having a system that is able to show and query classifications is nice, but creating the classification definitions by hand would be an overwhelming task. Hence a tool allowing advanced users<sup>6</sup> to create classifications is needed.

An intuitive way of creating a classification would be the following steps:

1. Determine what the classification should describe (Planning).
2. Create, connect and name/describe the classes (Create structure).
3. Assign item references to the classes (Index data document items).

We have named the application to generate classifications: ClassificationDesigner (CD). The work flow in the CD application should support step 2-3.

As mentioned in the section describing the schema for the classifications, each classification contains some properties that is needed for the CD only. For example is it necessary that the CD knows the location of the data document containing the data, that a given classification references.

After a classification has been created or edited, it should be put into the WH system, thereby enabling the visitors on the site to take use of the classification. There are 2 steps in this process:

1. Insert the XML document describing the classification into the XML storage in the WH system.
2. Make the WH system reload its cache.

Step 1 is trivial, while step 2 may seem a bit odd.

However since classifications may be large structures, it can take a while for the presentation layer to generate the “fold-able” dataguides. Considering the possible long processing time, and the fact that classifications does not change that often, the WH system should generate the dataguides once and then cache them for quick access.

In order to make system administration of the WH system easier the CD should also enable privileged users to remove classifications and download them for further editing. It would also come in handy if the CD is able to upload, download and delete data documents in the WH database – basically administering the WH System.

Since it should be possible to edit an existing classification, the CD must be able to import an existing classification, generate the graph structure and handle existing item assignments to the classes.

The application should be able to validate classifications, in order to help the users design usable classifications that does not break the system. The classifications

---

<sup>6</sup>Users with privileges to work on the WH system, not *ordinary* users (visitors).

could be validated using XML Schemas and maybe some tool to check the lattice structure.

Finally the application should be able to load and save classification drafts locally, allowing users to make a valid classification over several sessions.

### 3.6.1 Actors

This is a description of the actor that is going to interact with the Classification-Designer. The application is intended to help the WH system administrators create and edit classifications, so no *ordinary*<sup>7</sup> users will ever get to see this application.

**Designer** - A person that has the necessary knowledge to design classifications for the system. Designers are also able to administer the contents of the WH system, for instance by uploading new classifications or by removing obsolete or faulty classifications.

### 3.6.2 Use Cases

Descriptions of the actions that the designers should be able to perform in the ClassificationDesigner. The use cases are illustrated on figure 3.9 and described in detail in the following section.

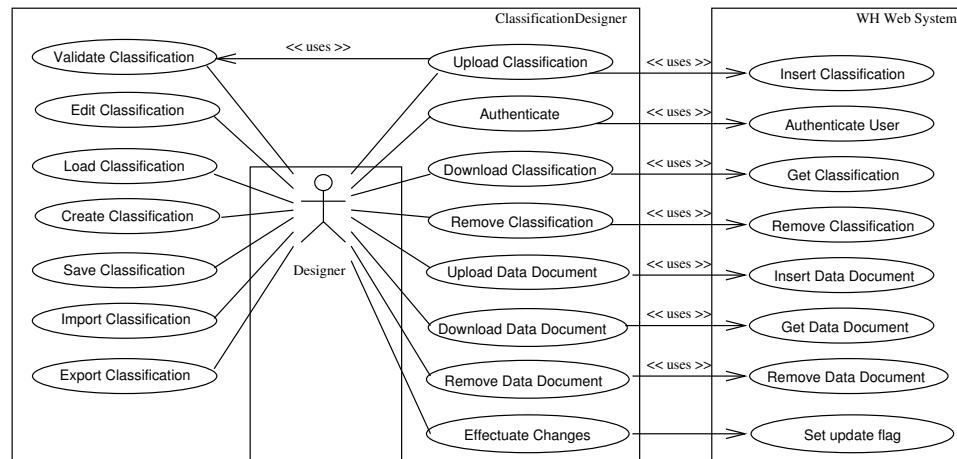


Figure 3.9: Use Cases for the Classification Designer

### Use Case Descriptions

The use cases for the ClassificationDesigner can be split up into two groups - the use cases that are related to creating and altering classifications, and a group of

<sup>7</sup>The normal users that browse the WH sites.

administrative use cases. The latter being the use cases on the right side of the actor in the diagram. The WH system use cases are not described, as they are simple methods and their meaning can be deduced from the use cases that << uses >> them.

**Create Classification** Enables the designer to create new classifications. When the designer chooses to create a new classification, he is guided through a sequence of dialogs, making sure all the necessary information for the CD is in place, before he starts drawing the structure.

The drawing of the classification structure and the assignment of item references is covered in next use case - “Edit Classification”. It is possible for the designer to assign sites to classes while drawing

**Edit Classification** This use case covers the actual process of creating the classification graph (showing the structure) and assigning item references to the class nodes.

Creation of the structure is done by inserting class nodes into a canvas and then connecting them with arrows.

After a node is created, the properties of the class, that the node represents in the classification, can be defined and item references to the data document can be inserted.

**Save Classification** Used to save a complete- or draft classification locally, in a specialized ClassificationDesigner format. This use case is used when developing classifications. A “save” preserves everything, even the size/location of the nodes in the graph-area (the XML representation of a classification only contains structure information, no layout).

**Load Classification** The process of loading a classification formerly saved in the “Save Classification” use case. This use case restores state completely.

**Validate Classification** An action used to check whether the loaded classification is valid, this validation checks to see if all required data is entered and whether the classification graph has a legal structure (e.g. one root and no cycles). Another obvious possibility for validation is validation against an XML Schema, but this option is covered in the “Import Classification” use case.

**Import Classification** Import a classification stored in an XML file. This use case is utilized when the user chooses to load an existing classification from the hard-disk. When the ClassificationDesigner imports the XML, it is validated using an XML Schema.

A very simple layout algorithm is implemented, in order to “spread” out the nodes in the graph, without the layout algorithm all the nodes would be inserted in the same spot, and the designer would have to move them around manually.

**Export Classification** Upon completion of a classification the user chooses to export it to an XML file. This file can then be uploaded to the WH system using the “Upload Classification” use case. The classification data is vali-

dated prior to export.

**Upload Classification** The process of uploading a classification to the WH system. The WH system relies on this action to upload valid data only, so some validation of the data must be performed prior to upload. The validation is done by validating the classification against an XML Schema.

**Download Classification** The process of fetching a classification in XML format from the server, and writing it out into an XML file that can be used in the “Import Classification”.

**Remove Classification** The action of removing a classification from the WH server. This action is provided in order to keep users away from the database management system, thereby trying to prevent some of the mistakes that may happen in such a situation.

**Upload Data Document** In order to provide users a simple interface to the WH database, it is possible to upload an XML document containing the data, that the classifications in the system are indexing.

**Download Data Document** Used for downloading an XML document containing the data that is present in the WH system. The Data Document is used in relation to many of the editing features.

**Remove Data Document** Used to remove unwanted Data Documents from the server.

**Authenticate** This use case enables the user to actually connect to the database. It must be called prior to making changes to the WH system, or else the server will reject the changes.

**Effectuate Changes** Because the WH system caches views of the classifications, the system needs to know when the data is changed and the views should be updated.

### 3.7 ClassificationDesigner Design

As mentioned earlier the ClassificationDesigner should provide a canvas, where the user is able to draw the classification structure with boxes as nodes and arrows connecting the boxes, indicating the specialization/generalization relationships.

Creating a drawing application from scratch would take a lot of time and as a stable, open source component is available, it was decided to base the drawing part on the JGraph [[jgraph](#)] component.

The JGraph component (and the CD application) is based on the *Model View Controller* pattern, a clean separation of model and presentation. The JGraph component contains the model (data) and the application provides a view to the model.

Since the JGraph component is the “heart” of the application, most of the classes needed in the application, are just views to the underlying data model, classes that change the default behavior of the JGraph component or graphical dialogs helping the designer to enter valid data for the classification.

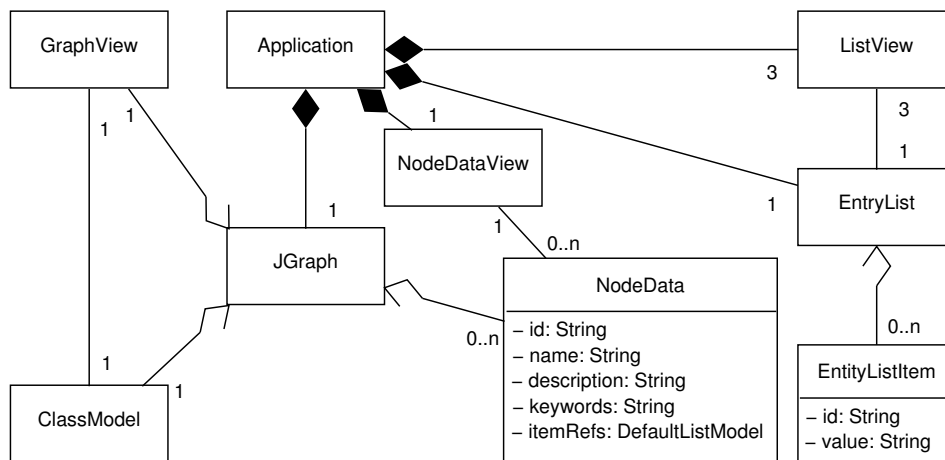


Figure 3.10: Abstract class diagram showing the connection between the models, views and the application.

Most of the *helper-classes* are not interesting at this stage, but a few key-classes can be identified:

**ClassModel** Defines which Node/Edge relationships are legal, thus defining how the JGraph data model structure should be. In order to maintain the lattice-like structure, it is necessary to only allow Edges between 2 different Nodes and not allowing *loose* Edges to be created.

**NodeData** A representation of the information stored in each graph node. This includes name, description, keywords and a list of references to the items in the data document. The `NodeData` corresponds to the data in the `<Class>` elements in the classification schema.

**EntryList** A list of references to the data document items, that can be indexed by the classification.

An abstract class diagram showing how the *key-classes* conceptually are related can be seen in figure 3.10. The application contains the JGraph component and the component contains its model, view and the `NodeData` objects attached to each class node. The application has a view to show / edit the `NodeData` values for a single node at the time. The three `ListView` classes indicates that there will exist three different lists of item references in the system. The first list will contain all the item references, the second all the assigned references and the last one all the unassigned references, making it a bit easier for designers to locate unassigned item references.

The ClassificationDesigner application will consist of a frame containing three areas that will contain a model-view each. Figure 3.11 on the following page shows a picture of what the CD application would look like. The hand drawn circles with class-names in, illustrate the three classes that makes up the views.

**Grapher.java** This is the view to the graph component, the drawing will be made in this part of the application.

**EntryList.java** This panel will contain different list views containing the item references (to the data document items) that will be added to the classes.

**ClassProperties.java** This panel is a view for the `NodeData`. It contains fields where the user can change the information in the `NodeData` objects. When the user selects a node in the Grapher panel, the content of the selected nodes `NodeData` will be shown in this panel for editing.

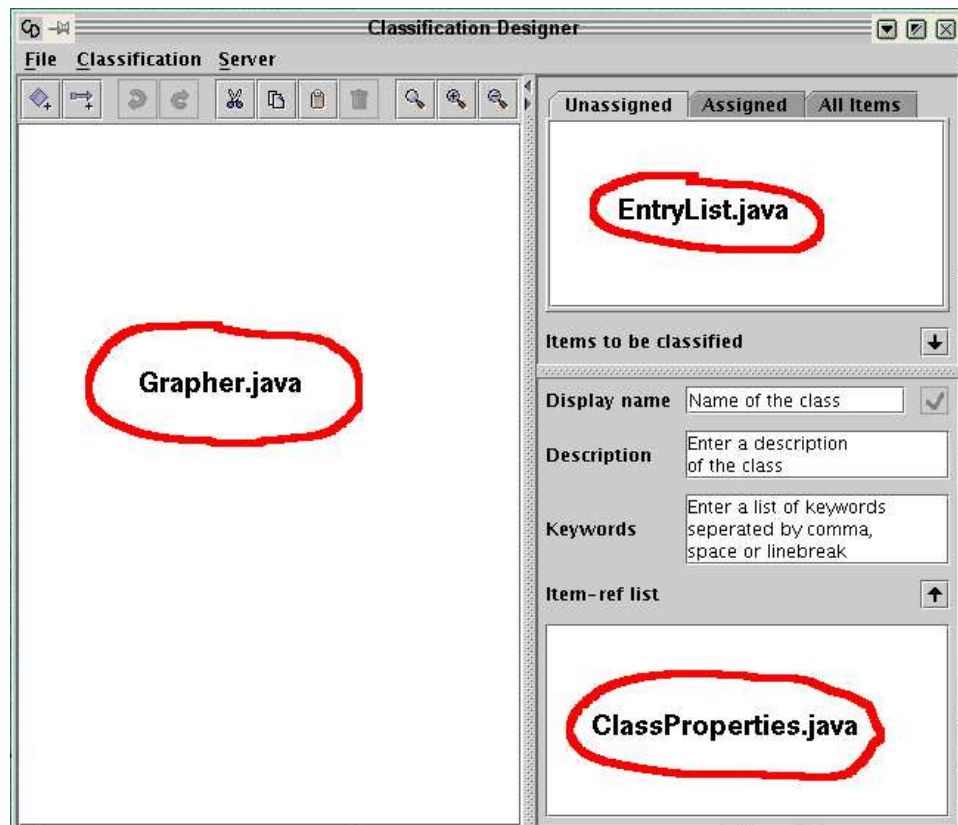


Figure 3.11: Overview of the CD application

Please refer to the description of the implementation in section 4.3 on page 89 for more detailed information.

### 3.8 Summary

The system description section provides the “big picture” of how the WH System and the ClassificationDesigner interact.

After the description of the systems, the XML documents that constitute the “backbone” of the systems is modeled. The classification document turns out to be a very strict specification, while the site document is more “in the spirit” of semistructured data, allowing much more freedom in the schema.

Having specified the two document types that the systems shall build upon, the next step is to create an UML model of the WH System. Actors and use cases are identified and described, in order to get a good overview of the features that the system should have.

Based on the actor and use case descriptions, a design of the WH system is created. Finally the class description is written, providing a decent picture of what that needs to be implemented and how.

Knowing how the system, to query the classifications, is going to work, it is time to make a model of the ClassificationDesigner. First is the identification of actors and use cases. Next is the design. It turns out that most of the functionality in the classification designer can be provided by a graph component, thus making most of the design phase unnecessary.





## Chapter 4

# Implementation and Testing

*This chapter documents the implementation of the WH System and the ClassificationDesigner, and explains how the implementations were tested. The first part of the chapter explains which software, that was chosen for the applications and why. Then the implementations of the WH System and the ClassificationDesigner are described. Finally the tests of the implementations are described.*

*We recommend the reader to read the sections “Choice of Software” and “Implementation of the WH System” carefully, because it contains some interesting information about the new XML technologies that are used.*

*The section “Implementation of the ClassificationDesigner” contains descriptions of all Java classes. This section is probably most interesting to people who want to know about all the programming details, or programmers who want to make changes to the ClassificationDesigner application. The “normal” reader who just needs to get an overview of the implementation, should not waste too much time on this section. Test strategies are explained in the “Tests” section. This section should be read by all readers.*

### 4.1 Choice of Software

The WH System and the ClassificationDesigner uses many different technologies. Some of these are well proven technologies like J2EE and XSLT, which should not cause any problems during implementation. Other technologies, like XQuery and XML databases are very new, and therefore all existing implementations of these technologies, do contain some bugs and missing functionality.

In order to make the implementation as painless as possible, and to end up with some robust applications, it is necessary to choose the best implementations of

these new technologies. The next section describes the experiences we had with the different technologies, and what considerations we made.

### 4.1.1 Choice of Software for the WH System

#### Choice of J2EE Application Server

There are several different free J2EE application servers available. Two of the most famous are *Jakarta Tomcat* and *JBoss*. The Tomcat server is the one Sun includes in their own J2EE reference implementation. JBoss has more features than the Tomcat server, and this make it seem more professional. JBoss is partially commercial in the sense that you have to pay for the documentation of the JBoss server – the server itself is free open source software. Fortunately it is not necessary to buy any documentation to set up a basic server configuration without any fancy extensions.

The choice of the application server is really not that important, because most J2EE applications can be moved from one J2EE server to another with minimum effort. J2EE applications uses standard configuration files for describing how the application should be deployed on the application server, but there usually is a single configuration file proprietary for an application server. Naturally this configuration file must be changed when moving from one application server to another, but this is usually the only change needed.

We decided to go with JBoss, because it is easy to setup and use.

#### Choice of XML Database

The choice of XML database is far more interesting. XML is still fairly new and many related technologies for working with XML data, has not even reached the *recommendation*<sup>1</sup> status at W3C. Because of this, there are only few XML database systems available, and only few of these are free. Unfortunately none of the free XML databases implements any XQuery functionality, and the few commercial XML database system that does implement XQuery are very expensive. The XML database that seems to have the most complete implementation of XQuery is called *IPedo*. In a review in *PC Magazine* [[ipedo-rev](#)] it is reported to cost about \$29000 per CPU.

We decided to go with a cheaper solution, and initially tried out *Xindice* [[xindice](#)] from the Apache Group. Xindice supports XPath expressions for querying the data, but unfortunately it does not support it very well. Very important parts of XPath is

---

<sup>1</sup>The recommendation status, means that W3C recommends it as a standard – this is the final status of specifications at W3C

missing, and it actually seems like the whole Xindice project has come to a halt – the latest “news” in the Xindice website is from November, 25th. 2002<sup>2</sup>.

Finally we decided to try the *eXist* XML database. eXist actually have an excellent (nearly complete) implementation of XPath 2.0, and there are implemented some very convenient extensions to XPath. Another positive thing about eXist is, that it implements the same Java API as Xindice does, so eXist could easily be integrated with our existing code. Naturally we decided to use eXist for the WH application. The API used for interacting with the database is called *XML:DB* [[xmldb](#)]. The primary goal of the XML:DB project is:

Development of technology specifications for managing the data in  
XML Databases

and the current draft specification actually provides an excellent interface for XML databases.

### Choice of XQuery Processor

Choosing a good XQuery processor was also difficult, considering that the current XQuery specifications is just a working draft. The most important requirement to the XQuery processor was, that it should be able to get data from the XML database.

We tried three different XQuery processors. The first one was *Qexo* [[qexo](#)] or *Kawa XQuery*. Qexo is just a small part of a large framework: the *Kawa Scheme System* – it is written by a guy named Per Bothner. Qexo had quite a few annoying flaws, when we tried it. First of all, it only includes a limited part of the XQuery specifications. Additionally it is poorly documented, so the best way to see if it supports a certain XQuery feature is by trying it out. Qexo is able to read XML documents from files and via the HTTP protocol, but we could not find any documentation of how to use other data sources. Qexo did really not comply with our requirements, so it was discarded.

Next we tried a processor called *Ipsi* [[ipsi](#)]. Ipsi is better documented, it is a more complete XQuery implementation and it should be possible to make it use an XML database as data source, even though we did not find out how. In spite of this Ipsi was also discarded because we tried *Saxon* [[saxon](#)]. Saxon was originally an XSLT processor, but the latest release of Saxon includes an XQuery processor as well. Saxon implements most of the functionality described in the XQuery specification. It can get XML data from anywhere (the software developer has to implement the data source connection himself) and finally, it is faster than both Qexo and Ipsi. In some tests we made, Ipsi and Qexo were about equally fast, while Saxon was about 11-12 times faster.

---

<sup>2</sup>The time of writing is August 29th. 2003

## Software Overview

The software we chose for the WH System is:

- JBoss Java application server v. 3.2.1 running on Sun's J2SDK v. 1.4.2
- eXist XML database v. 0.9.2
- Saxon XSLT and XQuery processor v. 7.6.5

### 4.1.2 Choice of Software for the ClassificationDesigner

As we need a validating SAX parser for checking XML classifications against an XML Schema, the Xerces parser was embedded in the classification designer.

#### Choice of Graph Component

The choice of the JGraph component was easy, it had a high rating on Google.com and the documentation for the component looked reasonable. The component utilizes the MVC pattern like the other complex Swing components and works pretty much like one would expect, however it is important to note a few more or less intuitive points.

- An item in the graph is an instance of a *DefaultGraphCell*.
- It is not possible to connect an edge to a "node", all connections are made between edges and ports. Each port is "glued" onto a "node", thus enabling us to connect our graph nodes using edges and ports.
- The nodes in the classification designer are instances of *DefaultGraphCell*, while the edges and ports are specializations of *DefaultGraphCell*.
- The "views" in JGraph are kept in the `GraphLayoutCache` and also accessed through this.

## Software Overview

The software we chose for the ClassificationDesigner is:

- JGraph v. 2.2.2 - The "core" of the application.
- Xerces2 v. 2.5.0 - Used as a validating SAX parser.
- Saxon XSLT and XQuery processor v. 7.6.5

## 4.2 Implementation of the WH System

This section documents the most interesting and important parts of the implementation of the WH system.

### 4.2.1 Overview of Components in the WH System

The application illustrated in the design in figure 3.7 on page 61, is actually implemented as three separate components. The database is a stand-alone Java application, just as a normal relational database would be.

The Enterprise JavaBeans also makes up a separate component. This component will reside in the application server's (JBoss) *EJB container* when deployed.

The last component consists of the front controller servlet, the request handlers and the JSP pages, which are bundled in a *Web Archive*, and deployed in the application server's *web container*. Even though this component is not connected to the EJB component, both components are deployed together in an *Enterprise Archive* – this is not necessary, but it makes deployment a lot easier.

The above may sound very “fancy” for those who do not know much about the J2EE architecture, but practically it just means that all the files necessary for the EJBs (EJB class files and some configuration files) are packaged in a *jar*<sup>3</sup> file e.g. `wh ejbs.jar`. The web archive containing the JSP pages, front controller servlet and request handlers are packaged in a similar fashion in another file e.g. `whapp.jar`. Finally these two jar files are stored together in the enterprise archive, which just is a new jar file e.g. `whapp.ear`. The enterprise archive is then deployed into JBoss.

### 4.2.2 Implementing the Web Archive

The image to the right shows the front controller servlet. It has the methods that are expected in a `HttpServlet` – it implements the methods `doGet` and `doPost`, that are invoked when it receives HTTP GET and HTTP POST requests. The most interesting thing about this class is the `handlerHash` `Map`, which is responsible for mapping HTTP request into request handlers. The tiny code snippet below shows the principle of how this works:

HttpServlet <b>wh.mvc.FrontController</b>
-handlerHash:Map -logger:Logger
+init:void #doPost:void #doGet:void

```

1 ...
2 handlerHash.put("/search.html", new SearchRequestHandler());
3 handlerHash.put("/simpleSearch.html", new
    SimpleSearchRequestHandler());
4 ...

```

Imagine that a client pushes the link for “search.html”. The client should retrieve the HTML page with the form for making an advanced search in the WH site list. Recall that this page contains all the dataguides.

What happens is, that the front controller simply looks it up in the map and finds the request handler `SearchRequestHandler`.

<sup>3</sup>“jar” is a “Java archive” – a compression program very similar to zip

The request handlers all implements the same interface shown here. This allows the front controller to cast all elements of the `handlerHash` map to the same class. The front controller then invokes the only method present in the request handler: `handleRequest`. This method takes care of looking up the search EJB and invoking all the necessary business logic. The code snippet below shows what happens in the `SearchRequestHandler`:

interface <b>wh.mvc.RequestHandler</b>
<b>+handleRequest:String</b>

```

1 ...
2 SearchSessionLocalHome sslh = Lookup.lookupSearchSessionLocalHome
   ();
3 LocalSearchSession lss = sslh.create();
4 DataGuideBean[] dataguides = lss.getDataguides();
5 request.setAttribute("dataguides", dataguides);
6 return "search.jsp";
7 ...

```

The `Lookup` class is a helper class for looking up EJBs using JNDI. The `LocalSearchSession` class is the local interface for the `SearchSessionEJB` which contains all the search facilities. The request handler simply invokes the method `getDataguides()` in the EJB, this method returns an array of `DataGuideBeans` – one for each dataguide. These beans are added to the HTTP request, and finally the `handleRequest` method returns `"search.jsp"`, which is the name of the view (JSP page) that should present the result to the user.

The `DataGuideBean` itself is shown here. It is really simple – it has a `dataguide` field, that contains the HTML presentation of the dataguide, and then it has a `name` and an `id` field. The only methods in the class are getters and setters for the fields.

<i>Serializable</i> <b>wh.beans.DataGuideBean</b>
-dataguide:String -name:String -id:String
+DataGuideBean +getData:String +setData:void +getId:String +setId:void +getName:String +setName:void

The above description of the front controller and the request handlers should be adequate to get an idea of how the controller mechanism works. The rest of the request handlers and JavaBeans are very similar to the classes just described, and readers who want all the details should consult the *javadoc* documentation or the source code.

The JSP pages are naturally an important part of the web archive. Figure 4.1 on the next page is a state chart, that tries to illustrate how users of the WH system will visit the JSP pages. The states (boxes) represents the JSP pages and the arrows shows a transition from one page to another. The text at the arrows is the HTTP requests that users send. The boxes with thick line are JSP pages, that are accessible from any other state – appendix D.1 on page 123 shows a screen shot of the WH application's welcome page, and the links in the top of the page each maps to a JSP page. These 5 links are always present in the WH application, hence the 5 corresponding JSP pages are accessible from any other state. There should be transition arrows from all other states to those 5 states, but for the sake of simplicity

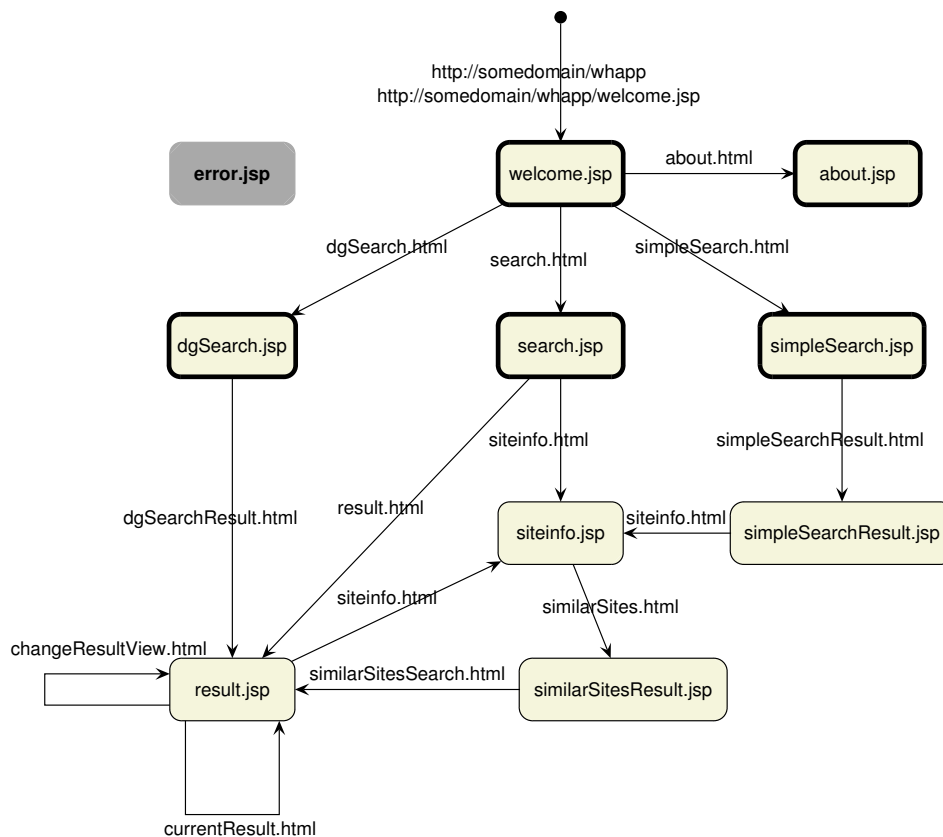


Figure 4.1: Overview of all JSP pages and how they are accessed through HTTP requests.

the 5 states are just emphasized using the thick lines. The JSP page “error.jsp” is naturally an error page and it is used for displaying error messages.

The figure should be self explanatory. The functionality of the JSP pages is as described in the class specification (section 3.5 on page 64). Notice how the page `result.jsp` is used to present search results for many different search facilities. This is because most of the search facilities present the search results as dataguides, hence they all use the same *model* for the data (recall the MVC design pattern), which is presented by the *view*: `result.jsp`.

The dataguides shown by the JSP pages are made using ordinary HTML and a small JavaScript to handle the folding/unfolding of categories.

Appendix D on page 123 gives an idea of how the search facilities are used from the WH System website. Notice, that the design of the website is “stolen” from the official WH prototype website.

### 4.2.3 Implementing the EJB Component

Most of the details about how the Enterprise JavaBeans are implemented, are not important – nor are they very interesting. This section focuses on the few interesting details there are, namely how the Saxon XQuery processor and the eXist XML database are incorporated into the WH system. Readers who are very interested in all the hairy details should look at the source code and the *javadoc* documentation.

#### Combining the XQuery Processor with XML Database

Suppose there is an XML document `datadoc.xml` stored on the hard drive. The code snippet below shows how that document is used in an XQuery evaluated with Saxon:

```

1 QueryProcessor processor = new QueryProcessor(new Configuration()
  , new StaticQueryContext());
2 DynamicQueryContext context = new DynamicQueryContext();
3 String xquery = "for $name in doc('datadoc.xml')/Sites/Site[
  contains(string(.), 'shark')]/Name return $name/text()";
4 XQueryExpression queryExpression = processor.compileQuery(xquery);
5 List result = queryExpression.evaluate(context);

```

The `QueryProcessor` at line 1 is the “base” class. It is used to create new `XQueryExpressions` (line 4), that can be evaluated (line 4). The XQuery in this example is very simple - it just returns all the names of all sites in the `datadoc.xml` document, that contains the word “shark”.

The `DynamicQueryContext` is actually the most interesting class. It can be used for setting external parameters that will be available in the XQuery at evaluation time, but an even more interesting feature is that you can use it for specifying how the XQuery function `doc()` behaves. The Java code below shows the same example as above, but now the XML document are stored in the XML database in the `DataCollection` collection (as illustrated in figure 3.8 on page 64).

```

1 QueryProcessor processor = new QueryProcessor(new Configuration()
  , new StaticQueryContext());
2 DynamicQueryContext context = new DynamicQueryContext();
3 context.setUriResolver(new WHUriResolver("xmldb:exist://localhost
  :4041/exist/xmlrpc"));
4 String xquery = "for $name in doc(\"DataCollection:/Sites/Site[
  contains(string(.), 'shark')]/Name\")/Result/Name return $name/
  text()";
5 XQueryExpression queryExpression = processor.compileQuery(xquery);
6 List result = queryExpression.evaluate(context);

```

Now the method `setUriResolver()` is called in line 3. This method tells the `XQueryExpression` which implementation of `UriResolver` to use – the `UriResolver` implements the XQuery `doc()` function. This allows the software developer to have complete control of how the XML data is fetched into the XQuery processor.



The default URI resolver simply reads the XML data from the hard drive, but the WH system uses our personal implementation – the `WHURIResolver`, which reads the XML data from the eXist database.

Notice that the `doc()` function now takes the argument:

```
DataCollection:/Sites/Site[contains(string(.), 'shark')]/Name
```

This argument obviously does not comply with the XQuery specification of the `doc()` function, but since it is not standard functionality to fetch data from an XML database, we had to invent a new convention for the use of `doc()`. The argument has two parts separated by a semicolon. The first: `DataCollection` is the name of the collection to use. The second part: `/Sites/Site[contains(string(.), 'shark')]` is simply the XPath expression that should be used on the `DataCollection` – recall that eXist supports XPath queries.

According to the specification [XQ-funop] the `doc()` function must return a document, but the above mentioned XPath expression may return a *collection* or *sequence* of XML elements. To solve this problem, the `WHURIResolver` wraps the result of the XPath expression into a document with the root element `<Result>`. This is why results of the `doc()` function *always* is followed by `/Result` e.g.

```
doc("DataCollection:/Sites/.../Name")/Result/Name
```

The `URIResolver` interface only has one method that should be implemented. The method has the signature:

```
1 public Source resolve(String href, String base) throws
   TransformerException;
```

When an XQuery expression is evaluated, it calls this method when it comes across the `doc()` function. The `href` argument in the `resolve()` method in Java is actually the argument for `doc()` in the XQuery expression.

The example below tries to make the description above a little easier to understand:

1. An instance of `XQueryExpression` with the above XQuery is created. It is evaluated and configured to use the `WHURIResolver`.
2. When the XQuery processor reaches the `doc()` function, it calls the method `resolve("DataCollection:/Sites/Site[contains(string(.), 'shark')]/Name", "")` in the `WHURIResolver`.
3. The `resolve` method separates the argument in a *collection* part: `"DataCollection"` and an *XPath* part: `"/Sites/Site[contains(string(.), 'shark')]/Name"`. The URI resolver has one connection to each collection in the database, so it can easily evaluate the XPath expression on the correct database collection. The result of the XPath expression is a sequence of `<Name>` elements, with names of all sites that contains the word “shark” in any sub element. Next a new document is created for the result with `<Result>` as root element. Each element in the sequence of site names is added as a child node to the new document.

Finally the `resolve` method returns the new document as a `DOMSource`<sup>4</sup> object.

4. The result document is now returned to the XQuery processor, which then evaluates the additional XPath expression `/Result/Name` on the result document, in order to extract the `<Name>` elements. Finally the XQuery processor returns the content of the `<Name>` elements as a `List of String` objects.

### Loading XQueries into the EJB

All the XQueries for searching the WH site list, are stored in the XML database. The `SearchSessionEJB` only has a single XQuery “hard coded” into the Java class, and that is the XQuery for fetching the other XQueries in the database. This XQuery is not much more than an XPath expression, that queries the `XQueries` collection in the database for an `<XQuery>` element with a specific name.

When the XQueries have been fetched from the database, they are compiled as shown in the previous section, so they are ready to be evaluated when needed.

### Using External Parameters

Most of the XQueries depends on external parameters such as keywords or the unique id of a site. These parameters are passed to the XQuery processor using the `DynamicQueryContext` class which also was used for specifying a custom URI resolver. It is simply done by calling a the method

```
setParameter(parametername,parametervalue) – e.g. setParameter('siteid','site127').
```

This naturally requires that the XQuery has defined an external value named “siteid”.

#### 4.2.4 Performance Considerations

It is very important to write the XQueries the “right” way to ensure fast execution of the queries. Most of the XQueries implements search facilities, that has to go through large amounts of data. Even though Saxon is fast to evaluate XQueries compared to other XQuery processors, it is *much* slower than eXist, when it comes to evaluating XPath expressions. Appendix F on page 139 shows some small scripts for testing the performance of eXist and Saxon. The result of these tests are shown in the two tables below (tested on a 600MHz AMD Athlon system). The tests made a number of keyword searches in two different WH site list – the first list contains all sites in the WH list (about 730 sites) and the other list only has a single site. These two documents were searched for 1 and 10 different keywords.

---

<sup>4</sup>This is simply DOM representation of an XML document

<b>eXist</b>			
	<b>Number of sites</b>		
<b>Number of keywords</b>	<b>0</b>	<b>1</b>	<b>730</b>
<b>1</b>	1.25	1.25	1.35
<b>10</b>		1.30	1.60

<b>Saxon</b>			
	<b>Number of sites</b>		
<b>Number of keywords</b>	<b>0</b>	<b>1</b>	<b>730</b>
<b>1</b>	1.50	1.50	3.00
<b>10</b>		1.55	3.05

Notice that one of the tests attempts to search a list with 0 WH list (actually it attempts to search a non existing list). This test is used for measuring how long time it takes Saxon and the eXist client programs to start up. The eXist client program uses about 1.25 secs. to start up, while Saxon uses about 1.5 secs. These numbers should be subtracted from all the other numbers in order to give a more realistic estimation of how fast it performs after deployment in the WH system.

eXist is obviously much faster than Saxon – the difference between searching 1 and 730 sites in eXist is just 0.1 second, while the difference in Saxon is 1.5 second. eXist searches 730 sites 10 keywords in approximately 0.35 second ( $1.60 - 1.25 = 0.35$ ), while Saxon uses 1.55 second ( $3.05 - 1.50$ ). Notice that Saxon for some reason is able to search 730 sites for 1 keyword in the same speed it can search it for 10 keywords, while eXist uses relatively more time when the number of keywords is increased. This might be due to the extended XPath functionality of eXist. eXist implements two functions `match-any(node, 'kw1', 'kw2', ...)` and `match-all(node, 'kw1', 'kw2', ...)`. The first function evaluates to true when the node (XML element) matches any of the keywords 'kw1', 'kw2', ..., while the second function requires that all the keywords match. In addition to check for matching keywords, eXist also replaces all matching words in the text with the XML element `<exist:match>someMatchingWord</exist:match>`. This feature might be used to highlight matching words in the text.

The main reason that eXist is so much faster than Saxon is, that its default behavior is to index *all* words in *all* XML elements or attributes.

The following things should be considered, when optimizing the queries:

- The eXist XML database should evaluate as much of the functionality as possible, because it is much faster than Saxon.
- There should be as few XPath queries in eXist per HTTP query, because the result of each query is first packed in a new XML document with `<Result>`

as root element – this was explained earlier. This preparation of the XPath results takes a little time.

- eXist should not evaluate `match-any()` or `match-all()` with too many keywords as argument. The tables above shows that if the number of keywords is increased from 1 to 10, when searching a list with 730 sites, eXist needs an additional 0.25 second to evaluate the expression.

These guidelines where used during the implementation of the XQueries.

### 4.2.5 The XQueries

Appendix J on page 163 is the XML document containing all the XQueries – the structure of the document is as described in section 3.4.4 on page 63. This section briefly describes each of the XQueries. There is not made any attempt to explain exactly *how* the XQueries performs the queries, since this is rather complicated.

Notice that the headlines below refer to the `<Name>` element in the XML document and not to XQuery functions.

#### **getclassifications**

Simple query which fetches all classifications in the database. More precisely it retrieves all `<Classification>` elements in the `Classifications` collection.

#### **getstylesheet**

Gets a specific XSLT style sheet from the database.

#### **getsites**

Gets a list of sites from the database.

#### **simplesearch**

Performs a simple keyword search in the WH site list. The query uses the extended functionality of eXist when searching. `match-any()` is used for making a search that requires one keyword to match, while `match-all()` is used for matching all the keywords. Notice that the XQuery itself does not use any of these functions in the XML document. This is because this information is passed to the XQuery as an external parameter from the search EJB.

This search facility performs a full text search in all the XML elements in the collection of WH site data.

**evalxpathepr**

Very general function, that simply takes one external parameter, and uses this as an argument for the `doc()` function and returns the result.

**makedataguide**

This is one of the more important XQueries. It takes one external parameter: “classification”, which is a `<Classification>` element. This classification is transformed into the HTML dataguide that is returned.

**findsimilar**

Query for finding sites that belong in the same categories as some chosen site. It takes two external parameters: “site” that is a string with the id of the chosen site, and “ancestordepth” that specifies how “high” in the classification lattice structure to look when generating the result (this may not make any sense for anybody if they have not tried the functionality in the WH system web page).

**advancedsearch**

This is largest XQuery, that contains all the functionality for performing an advanced query. It takes two external parameters. One of them is a list of `<Classification>` elements with the classifications that should be searched. The other parameter is some XML data containing all keywords and categories provided by the user. This XML data is created in the search EJB.

This query returns a list with 2 elements. The first is the search results as an XML `<Classification>` element while the other is the result as a simple list. The structure of the simple list is described in-line in the XQuery code.

Like the simple search facility, this search facility also performs a full text search in all the XML elements in the collection of WH site data.

**searchclassifications**

Searches classifications for categories that matches some specified keywords. Takes two external parameters: the keywords and the classifications. This search facility performs a full text search in the classification documents.

## 4.2.6 General Notes About Query Results

### Representation of Query Results

Notice that results of searches in the WH site list always are presented to the user either as a list or as a dataguide. The “result dataguides” are generated from “result classifications” returned by the `advancedsearch` XQuery. It is extremely convenient to return search results as classifications because the `advancedsearch` XQuery can use the result classification as an argument for a new search. Consider the following:

- A user enters the WH System website. He chooses the advanced search facility, where he selects some categories in some of the standard dataguides and enters some keywords. He pushes the search button.
- The user query is eventually handled by the `advancedsearch` XQuery. One of the arguments for the XQuery is the standard classifications. The other argument contains information about selected categories and keywords. The XQuery returns a result containing *a result classification* (and a simple list). The result classification is stored in the J2EE server in a user session variable.
- The user is now presented with a result dataguide generated from the result classification. The user thinks he received too many results, so he selects some categories in the new dataguide and enters some keywords. He pushes search.
- This time the `advancedsearch` XQuery uses the result classification from the previous request, as an argument for the search. Again the XQuery returns a result containing a new result dataguide (and a simple list).

The user can keep making new searches in a continuously shrinking dataguide, until the search does not match any sites and nothing is returned.

### The “Hit point” System

Visitors to the WH System website can choose to see the result of an advanced search as a list instead of a dataguide. The list shows how good the sites matches the search, by showing some “hit points”. The hit points are assigned the following way:

A site that matches an advanced search will as a minimum be present in one category which the user chose, and match one keyword – this gives the site 1 hit point. The site gets 1 hit point for each additional category and 1 hit point for each additional matching word. Notice that the keyword “sword” could match both words “swords” and “swordfish”, hence two points would be assigned – one for each word. A simple hit point formula can be written as:

`noOfMatchingCategories + noOfMatchingWords - 1`

## 4.3 Implementation of the ClassificationDesigner

The section on modeling and design describes how the ClassificationDesigner (CD) is put together using the JGraph component and a couple of standard views based on JList and a NodeData view.

This section offers a detailed description of the classes actually used in the application, and some notes on how the tasks that the different classes perform are carried out.

### 4.3.1 Important Notes about the ClassificationDesigner

There are a few details that should be kept in mind when reading the code for the CD. First, the *parent* frame contains a `Properties` object called `props`. The `props` object contains the *global* values in the application, this is convenient when the application state needs to be saved, restored or when values need to be shared between different parts of the system. The advantage of sharing “global” data using a `Properties` object is that the components are more loosely connected. The disadvantage is that a little more bookkeeping is needed when extracting values from `Properties`.

#### The ClassificationDesigner Properties

The following properties are used in the application:

- cdd-filename** ⇒ Filename for the last saved or loaded file (in the binary format that includes layout information).
- ccd-pathname** (c) ⇒ Absolute pathname for the last load or save location (working directory for classifications).
- classification-description** ⇒ Description of the classification being worked on.
- classification-entry-element** ⇒ The name of the XML element designating the XML entities in the data document that are to be classified.
- classification-id** ⇒ Contains the value of the `id` attribute in the `<Classification>` element of the classification imported from XML.
- classification-keywords** ⇒ A list of the keywords that describe the currently active classification.
- classification-name** ⇒ The name of the active classification.
- classification-stylesheet** ⇒ Absolute path to the presentation style sheet for the classification (not used in this implementation).
- classification-stylesheet-path** (c) ⇒ Absolute path to the last location pointed to in the style sheet file chooser (style sheet working directory).
- data-doc-path** (c) ⇒ Absolute path to the current data document working directory.

- data-document** ⇒ Absolute pathname of the currently selected data document.
- entity-list-query** ⇒ The XQuery used to extract the data document items that the classification is indexing.
- password** ⇒ The password that the user used successfully to authenticate the DBUpdateSession EJB.
- server-connection-ok** ⇒ `true/false` value indicating whether the connection test has been completed successfully.
- server-name** ⇒ The hostname / IP of the WH system server.
- server-port** ⇒ The port number where the Java Naming service is running on the WH system server (defaults to 1099).
- username** ⇒ The username used to authenticate the DBUpdateSessionEJB.
- xml-filename** ⇒ Absolute path to the XML document last loaded/saved (if any).
- xml-pathname** (c) ⇒ Absolute path to the XML import/export working directory.

Most of the properties that start with “classification” contains information that goes into the `<Classification>` construct. This is needed because the graph structure only contains information about the classes.

Most of the properties applies only to the current working session, for instance things like username/password should not be carried over between sessions. In fact the only values that are carried over between sessions are the working directory paths (indicated with a “(c)”). The “carry-over” is done by storing the values in a property file called `CDesigner.properties`, in the same location as the `ClassificationDesigner.jar` is located.

## The Dialogs

The dialogs used in the application are setup in a somewhat similar manner to `JFileChooser`s. A dialog life-cycle has the following steps:

1. The dialog is instantiated and internal initialization takes place.
2. If the dialog depends on special values from e.g. the properties object, the caller may need to query the `setupOK` attribute, to see whether the dialog was initialized properly or not.
3. If initialization went well, the `showMe()` method is invoked.
4. When the user is done using the dialog the `showMe()` method returns a value indicating the choices made in the dialog. (for instance “OK” or “Cancel”).

Step 2 is not necessary for every dialog in the application. The section describing the classes will mention if the step is necessary for a particular dialog.

## The DBUpdateSessionEJB

The interaction between the `ClassificationDesigner` and the WH system is channeled through the EJB. The bean interaction works over RMI. Once the bean is looked up, its methods are used like any other local object methods.



### Conversion between XML and ClassificationDesigner Graph

The conversion between XML and ClassificationDesigner graph representation utilizes a DOM as middle tier.

XML file  $\Leftrightarrow$  DOM  $\Leftrightarrow$  ClassificationDesigner graph

Validation of the data are always performed “on” the first arrow encountered. For example when converting from XML to graph, the data is validated during the parsing from XML into DOM using an XML Schema (`classification.xsd`). The other way around the data is validated as the DOM is constructed.

#### 4.3.2 Class Description for the ClassificationDesigner

This section contains short descriptions of the classes. It is not full descriptions of the code, but it points out important issues that need to be taken into account, if someone needs to work with the code. Readers interested in further details should consult the javadoc or the source code.

##### AuthenticateDialog

This dialog handles user authentication with respect to the DBUpdateEJB. Username/password and bean are be provided when calling the constructor. If the username string is non-empty, the constructor tries to perform a *silent authentication* based on the values supplied in the constructor.

If the silent authentication succeeds, the `statusOK` flag is set to `true` and nothing more needs to be done. The code depending on this dialog should check whether `getStatus()` returns `true` or `false` before going on. If `getStatus()` returns `true`, the bean is already authenticated using user data entered earlier in the current session.

In the case where `getStatus()` returns `false`, the method `showMe()` should be used to display the dialog.

JDialog
<b>AuthenticateDialog</b>
-parent:CDFrame
-me:JDialog
-user:String
-pass:String
-statusOK:boolean
-userTF:JTextField
-passTF:JPasswordField
-bean:DBUpdateSession
-okJB:JButton
-cancelJB:JButton
+AuthenticateDialog
-setLayout:void
+showMe:boolean
+getStatus:boolean

##### BeanTools

A simple *helper-class* that handles the lookup / creation of the DBUpdateSessionEJB and clean-up after the lookup.

BeanTools
-BeanTools
+lookupBean:DBUpdateSession
-lookupBeanHome:DBUpdateSessionHome

## CDFrame

This is the frame that contains all the other views and dialogs. The `CDFrame` is usually stored in a field called `parent` in the different parts of the application that needs access to the frame.

The `props` object mentioned in the start of this section, is stored in this class, providing easy access to its contents for all the other classes in the application.

JFrame
<b>CDFrame</b>
+props:Properties
+CDFrame

## CDMarqueeHandler

A marquee handler takes care of the mouse interaction in the graph component; multiple selection, single selection, connection of nodes etc. This application needs a custom handler to handle the insertion of edges between the nodes.

The handler also changes the cursor if it is above a port and it highlights the port itself. Temporary drawing of edges while the user is trying to connect two nodes is also provided by the marquee handler.

The marquee selection (done by pressing a mouse button and dragging a box around the items to be selected) is also handled by this class.

If the handler receives an event that it does not handle, the event is dispatched to its super class.

BasicMarqueeHandler
<b>CDMarqueeHandler</b>
#start:Point #current:Point #port:PortView #firstPort:PortView -graph:JGraph
+CDMarqueeHandler +isForceMarqueeEvent:boolean +getSourcePortAt:PortView #getTargetPortAt:PortView +mousePressed:void -paintConnector:void -paintPort:void +mouseDragged:void +mouseReleased:void +connect:void +mouseMoved:void

## ClassModel

`ClassModel` defines the model used in the graph component, while the class looks very simple it has a major impact on how the graph behaves.

The model used for the graph component in this application is pretty simple though. The only thing that is needed in order to make the graph behave as expected, is to describe which connections that can be made using edges.

And as there are not that many restrictions on the shape of an classification. The model only have to prohibit the user from making *self-references* on nodes. The classifications must also be *cycle-free*, but since this is a bit more complex to detect, this code is implemented as a part of the validation function in the `Grapher.java`.

DefaultGraphModel
<b>ClassModel</b>
+acceptsSource:boolean +acceptsTarget:boolean

## ClassProperties

This is a view to the data stored in each graph node (Class). The `ClassProperties` panel occupies the lower-right corner of the application.

The class registers itself as a `GraphSelectionListener` and every time a selection event occurs, it checks to see if a single node was selected, if so it shows the contents of `NodeData` object associated with the node. If the selection spans anything else than exactly one graph node, all the fields in the panel is cleared in order to avoid misunderstandings.

The panel has a button (indicated with the “√” symbol) used to apply changes in the text fields to the data stored in the selected graph node.

The class also registers itself as a `DocumentListener` listening for events in the text fields it has. If the content of one of the fields change, the button used to apply the changes to the graph is enabled.

The text fields also have listeners enabling the user to hit “Enter” in a field, to make the `NodeData` object get updated with the values from all the fields in the `ClassProperties` panel.

The most interesting methods in this class are the `propagateChangesToGraph()` and `propagateChangesFromGraph()`. They perform the actual moving of node data between the fields in the panel and the selected graph node.

This class relies on a reference to the `EntryList` class when handling the assigned item references.

JPanel <i>GraphSelectionListener</i> <i>DocumentListener</i> <b>ClassProperties</b>
-parent:CDFrame #grapher:Grapher -nameJL:JLabel -descrJL:JLabel -keywJL:JLabel -nameTF:JTextField -descrTF:JTextArea -keywTF:JTextArea #entityList:JList #entityListing:EntryList -okJB:JButton -removeJB:JButton -nameText:String -descrText:String -keywText:String
+ClassProperties -setLayout:void +propagateChangesToGraph:void -resetFields:void +valueChanged:void -propagateChangesFromGraph:void +getSelectedNode:DefaultGraphCell +setEntryListing:void -enableOK:void +insertUpdate:void +removeUpdate:void +changedUpdate:void

## ClassificationDesigner

Contains the `main` method that starts up the application. The application is started by creating a frame with 2 split panes, creating and adding the three panels that goes into the three locations. Furthermore the mutual references between the `EntryList` and `ClassProperties` classes are setup.

ClassificationDesigner
+main:void

## ClassificationProperties

This is a dialog that allows the user to change the properties of the Classifications. Each of the fields in this dialog must have a non-empty value in order to form a valid classification.

The dialog can exist in two different contexts: Standalone or as part of a sequence.

The sequence mode is used when a user selects to create a new classification, this triggers a sequence starting with `SelectDataDoc` →

`ClassificationProperties` →

`PresentationFormat`, when running in this mode each dialog depends on the information entered in the former.

The standalone mode is used when the user wishes to change existing properties. The dialog is not dependent on other dialogs when running in this mode, and the field validation is not as strict.

The most interesting field in this dialog is the *Class Entry Element* field. This field allows the user to select which XML tag that denotes the items to be referenced in the data document.

The code that populates the combo-box uses an XQuery to determine the candidates for XML constructs that can be indexed.

The XQuery looks like this:

```

1 String query = "fn:distinct-values(for $a in doc(\""+
2     parent.props.getProperty("data-document", "Error, data
3     document not found")+
        "\\")//*[@id] return fn:name($a))";

```

As it can be seen the XML-tag candidates in the data document are those elements that contain an `id` attribute. Furthermore should the XML construct selected as *Class Entry Element* contain at least one simple element, that can be used to describe the item reference to the element, but this constraint is enforced in the `PresentationFormat` dialog description.

This dialog also supports a *Stylesheet* property, but this is not a part of the implementation, so the `showStylesheet` flag is false and the GUI components for the style sheet selection are not added to the dialog.

The class also contains an inner class - `XSLFileFilter`, which is a simple file filter used in the file chooser for the style sheet field (when style sheets are enabled).

JDialog
<b>ClassificationProperties</b>
-parent:CDFrame -me:JDialog -nameTF:JTextField -stylesheetTF:JTextField -keywordEdit:JTextField -keywordList:JList -keywordListModel:DefaultListModel -classEntryNameCombo:JComboBox -descriptionJTA:JTextArea -changeJB:JButton -resetJB:JButton -cancelJB:JButton -okJB:JButton -stylechooser:JButton -keywordAdd:JButton -keywordRemove:JButton -defaultMsg:String -listdata:Object[] -status:boolean -standalone:boolean -defaultStyleSheet:String -showStylesheet:boolean
+ClassificationProperties -parseDataDoc:void -setLayout:void -resetValues:void -storeValues:void -restoreValues:void -validateFields:boolean +showMe:boolean
-XSLFileFilter

## EntityListItem

This is the Java representation of the item references that are essential to the way the classification designer works. An item reference contains a string representation (name) of the item that it references, and a unique `id` enabling the applications to locate the correct element in the XML data document.

<i>Serializable</i>
<b>EntityListItem</b>
-value:String -id:String
+EntityListItem +toString:String +getId:String +equals:boolean

The constructor takes a string, where the first part is the name of the list item and the second is the id. The two parts are separated by “`@@@`”, enabling the user to define custom list item names, without changing the format of this class.

This class also defines an `equals(Object obj)` method, allowing us to detect whether two `EntityListItem`s are equal based on their contents instead on object identity (this is for instance used when importing classifications and synchronizing the loaded classification with the lists in the `EntryList` class).

## EntryList

This is the panel that goes in the upper right corner of the application. This class is handling the lists of item references. `EntryList` basically features three different lists: *Assigned Items*, *Unassigned Items* and *All Items*. When a user has selected a node in the graph, it is possible to select item references from the lists and “move” them down (into the `NodeData` of the selected node).

<i>JPanel</i>
<b>EntryList</b>
-parent:CDFrame -classProperties:ClassProperties #unassignedList:JList #assignedList:JList #allList:JList -unassignedListModel:DefaultListModel -assignedListModel:DefaultListModel -allListModel:DefaultListModel -tabbedPane:JTabbedPane
+EntryList -setLayout:void +synchronizeListsWithGraph:void -getActiveList:JList +buildList:boolean +buildAllList:boolean +setClassProps:void

The *All Items* list are constructed based on the value of the `props.getProperty("entity-list-query", "")`. If no value is defined the list is not built until the necessary data are available.

In order to keep the three lists synchronized with the data in the graph nodes, it is necessary to keep track of which `EntityListItem`'s that are assigned and so on. One of the problems is that any item can be assigned to multiple graph nodes, so when the item reference is removed from the graph node, it is not certain whether it should appear in the *Unassigned Items* list or not.

Instead of making a complex piece of code to track item references across multiple classes/models, it was decided to make a simpler solution. The *All Items* list is static and if two out of the three lists are known, it is trivial to compute the last one, so it was decided to run through the graph nodes and use the assigned item references to build the *Assigned Items* list, and then finally computing the unassigned list.

This is not the most efficient solution, however it is a very robust way of handling it. Even if something should happen to bring the lists out of synchronization, next time some change happened they would all be back in a consistent state again.

### GeneralServerDialog

The `GeneralServerDialog` handles almost all the server related tasks, like upload, download and deletion of data collections on the server.

Upon calling the constructor, the program specifies what type of document that the current action is working with (classification/data document), and what type of action (upload, download or remove) that is to be performed.

The contents of this dialog depends on the output from the `DBUpdateSession` bean, so the bean must be looked up<sup>5</sup> and authenticated before setting up the layout. If no server details are found in `parent.props` the dialog opens the `ServerProperties` dialog and waits for it to return with the values. After the bean is looked up successfully, a *silent authentication* is tried, given that the sufficient log-on data already is present in the `parent.props`. If no username/-password is saved in the `parent.props`, the `AuthenticateDialog` is shown.

If both bean lookup and authentication went well, the layout is setup and the dialog is ready to be activated with the `showMe()` method.

JDialog	
<b>GeneralServerDialog</b>	
+UPLOAD_ACTION:int +DOWNLOAD_ACTION:int +REMOVE_ACTION:int +DATADOC_TYPE:int +CLASSIFICATION_TYPE:int	-parent:CDFrame -action:int -type:int -actionText:String -typeText:String -toFrom:String -list:JList -file:JTextField -collectionname:JTextField -status:boolean -me:JDialog -bean:DBUpdateSession -setupOK:boolean
+GeneralServerDialog -setLayout:void +showMe:boolean +setupOK:boolean -compressString:byte[] -decompressByteArray:String -createNodeMap:Hashtable -getNodeByName:Node	-RemoveAction -UploadAction -DownloadAction

<sup>5</sup>Done by using the static methods in BeanTools.

## Grapher

JPanel GraphModelListener <b>Grapher</b>	
<pre> -parent:CDFrame #graph:JGraph #undo:Action #redo:Action #remove:Action #group:Action #ungroup:Action #cut:Action #copy:Action #paste:Action #nodeEdit:Action #classificationProperties:Action -undoManager:GraphUndoManager -classProps:ClassProperties -selectionListener:GraphSelectionListener -grapherKeyListener:KeyListener +stateChanged:boolean -toolbar:JToolBar -defaultError:String  +Grapher +setClassProps:void -insert:void -insert:Port -insert:void </pre>	<pre> -undo:void -redo:void #updateHistoryButtons:void #prepareExport:Vector -locateClassRoot:DefaultGraphCell -noGraphCycles:boolean -noGraphCyclesN:boolean #buildDOM:Document -numberGraphNodes:void #processDOMtoGraph:boolean -generateGraph:int -processGraphToDOMNodes:void setupToolBar:JToolBar #setupMenuBar:JMenuBar -validateClassification:boolean -checkNecessaryFields:boolean -checkNecessaryNodeData:boolean -stateChanged:void -resetState:void -setState:void -getState:Vector +graphChanged:void -createNodeMap:Hashtable -getNodeByName:Node -getNodesByName:Vector  #EventRedirector #CDDFileFilter </pre>

`Grapher` is the panel that contains the `JGraph` component, it is the “main view” of the application. Since this class contains the `JGraph` component, most of the graph related code are placed in this file. Things like the menu-bar and tool-bar are also found in this class, since their actions are related to the graph.

Two `insert` methods exists in this class. One is used for generating new clean nodes (empty `NodeData`) and other one inserts a node with a filled out `NodeData` object associated (used when importing an existing classification from XML).

The `Grapher` contains three methods concerning state: `getState()`, `setState()` and `resetState()`. The state are a `Vector` containing a snapshot of a state. The state can be used for saving/loading a work session etc.

The grapher also contains a couple of methods used to validate the classification structure. The application has three ways of doing validation of a classification:

1. Count of root nodes - A classification has exactly one (no incoming edges).
2. Cycle detection - A classification must not contain cycles.
3. An XML representation of a classification can be validated using an XML Schema.

The *Cycle detection algorithm* is pretty simple. It starts at the root node and makes a depth-first search, where the paths are saved as lists of lists of nodes in a hashtable. When a node is visited, all of the paths leading to it parents are looked up. If the node already exists in one of them, a cycle is detected and the algorithm is finished. If the node does not exist in any of the paths, it copies the paths, appends itself to the end of each path, saves the list of paths in the hashtable and all its child nodes are then visited recursively.

All data on the WH system server is in XML format, so in order to upload a classification it must be stored in an XML document. The following procedure is used for exporting a classification to XML.

1. Validate classification, using root count and cycle detection and check necessary `props` values.
2. Create a DOM using the data in `parent.props` and in the graph.
3. If all necessary data exists and the DOM is created, then transform the DOM to an XML document.

Import of an XML document to the `ClassificationDesigner` is almost the reverse procedure:

1. Parse file into a DOM using a validating parser (uses the `classification.xsd` XML Schema).
2. If DOM is created, create `NodeData` objects, populate `EntryList`, fill values into `parent.props` and generate the graph. Finally synchronize the lists.

The class has a few helper methods for generating DOM trees, node id's and so on. It seems important to mention `createNodemap()`, `getNodeByName()` and `getNodesByName()`, they are very handy because a standard Java DOM is a bit tedious to navigate through, by creating a node-map (maps node names to the actual DOM node) it is possible to extract data from a DOM without too much pain. The `getNode(s)ByName()` returns a single node or a list of nodes associated with a name.

In order to force the classifications to have the correct structure, the graph uses the `ClassModel` as internal model. The only thing that can break the structure is when the user deletes a node with associated edges. In that case the edges sticks around, but one of the ends is not connected. The appropriate fix for this *feature* was to change the `remove` method, making it more “greedy” when a node is deleted; the node itself and all edges connected to it, are removed from the graph.

As this class contains the classification itself and the menu/tool-bars most of the code that calls the other dialogs and such is present in this class.

### GrapherKeyListener

The `GrapherKeyListener` is hooked up to the graph and is called whenever the user hits a key while working in the “graph” area. This class makes it possible to select some graph elements and hit the “delete” key to delete them.

<i>KeyListener</i>
<b>GrapherKeyListener</b>
-grapher:Grapher
+GrapherKeyListener +keyTyped:void +keyPressed:void +keyReleased:void



### GrapherSelectionListener

Another “helper-class”, this one is hooked up to the graph and reacts on selection events. This listener handles the enabling (and disabling) of the cut, copy and paste icons on the ClassificationDesigner tool-bar.

<i>GraphSelectionListener</i>
<b>GrapherSelectionListener</b>
-grapher:Grapher
+GrapherSelectionListener +valueChanged:void

### GrapherUndoManager

Yet another “helper-class” this class extends the regular `GraphUndoManager` allowing more control over the GUI. This class enables/disables the undo and redo actions in the graph component. Works like a swing undo-manager.

<i>GraphUndoManager</i>
<b>GrapherUndoManager</b>
-grapher:Grapher
+GrapherUndoManager +undoableEditHappened:void

### InfoDialog

This class is used whenever the application needs to send a small message to the user. The `InfoDialog` comes in two different shapes, with and without a “Cancel” button.

This dialog basically enables the application to display error messages and have the user press “OK” before recovering. Making the dialog able to display both “OK” and “Cancel” enables the application to use the dialog for relaying questions to the user, for instance questions regarding overwriting of existing files and so on.

Long messages can be split up by inserting “\n” in the message.

<i>JDialog</i>
<b>InfoDialog</b>
-parent:Container -choice:int +OK_SEL:int +CANCEL_SEL:int
+InfoDialog +InfoDialog -init:void -initc:void -closeMe:void +showMe:int

### NodeData

A simple object containing the information that can be stored in a class (graph node). This class implements a `DefaultGraphCell.ValueChangeListener` enabling the graph to notify it of changes to the data.

It also overrides the `clone()` method, graph nodes are cloneable, so their user-objects should also be.

The `toString()` override makes the name show up in the boxes in the graph.

<i>DefaultGraphCell.ValueChangeListener</i> <i>Serializable</i>
<b>NodeData</b>
name:String description:String keywords:String id:String itemRefs:DefaultListModel
+NodeData +toString:String +clone:Object +valueChanged:Object

## ObjectCloner

This is a little convenience method, it is seldom that *deep copy* is needed in Java, but when it is, the `clone()` method should be used. However when working with more complex data structures, the `clone()` method becomes troublesome to define for all objects. A much simpler solution is to use *serialization* to clone objects. The static `deepCopy(Object oldObj)` method simply tries to write the `oldObj` into a stream and then read it out again. If this succeeds the returned object is a clone of the `oldObj`.

ObjectCloner
-ObjectCloner +deepCopy: Object

Cloning is mostly used in the “state” methods of `Grapher`, because it is not possible to store for instance the state of the graph by doing a `JGraph state = grapher.graph;`. A `JGraph state = grapher.graph.clone();` is a better attempt, but not good enough, because the `graph` component apparently contains some reference fields that are not marked as transient, so the cloning fails.

However it is possible to extract just enough information from the graph, to be able to capture the “state” of the graph and just `deepCopy` that.

## PresentationFormat

A dialog that enables the user to specify, how the name of the item references should be constructed, using the data in the data document.

The dialog will not work unless a data document and a class entry element is defined. If the prerequisites for the dialog are in place, the dialog queries the data document to find “simple” sub-elements of the “class entry element”. The system needs at least one simple element to be contained, because the name of the item references for the elements are extracted from one or more simple sub-elements.

In the WH system the class entry element is “Site” and an obvious simple sub element is “Name”. However it could be that some user would like to have the item reference name contain for instance “SiteNumber-Name” for each site. In this case the user should select “SiteNumber” a separator and “Name” and press “OK”. This would cause the “entity-list-query” string to be populated with the XQuery, that will generate the right item reference names.

JDialog PresentationFormat
-me: JDialog -parent: CDFrame -addJB: JButton -remJB: JButton -sepJB: JButton -upJB: JButton -downJB: JButton -jl: JList -jr: JList -leftmodel: DefaultListModel -rightmodel: DefaultListModel -previewField: JTextField -status: boolean -setupOK: boolean -queryChanged: boolean -oldQuery: String
+PresentationFormat -validStartData: boolean +showMe: boolean +setupOK: boolean -setupLayout: void -makeListQuery: List +queryChanged: boolean
-RightListChangeListener

Please note that changing the presentation format in an existing classification, causes the `ClassificationDesigner` to rebuild the “All Items” list, update all `NodeData` objects in the graph and finally synchronize the lists in `EntryList`. The `NodeData`

references are updated by looking the “new” values up in the “All Items” list using the stored id’s.

### SelectDataDoc

This is the dialog that enables the user to assign a data document to a classification. Once the user has selected a data document, the file details are stored in `parent.props` under the name: “data-document”.

JDialog	
<b>SelectDataDoc</b>	
-parent:	CDFrame
-me:	JDialog
-status:	boolean
-tf:	JTextField
+SelectDataDoc	
-restoreValues:	void
-setupLayout:	void
+showMe:	boolean

Changing the data document for an existing classification is a drastic measure and should not be necessary in any case. However it is possible to do this, but in order to keep the classification in a state, where it can be trusted that item references still point to valid entries in the data document, all item references are stripped off the graph and assignments must be done over again using the new data.

### ServerProperties

`ServerProperties` is the dialog concerning connection properties. The user must enter an IP address and possibly a port on the WH system server, in order to setup a connection for executing the `DBUpdateSession` EJB.

JDialog	
<b>ServerProperties</b>	
-parent:	CDFrame
-me:	JDialog
-server_name:	String
-server_port:	String
-status:	boolean
-testOK:	boolean
-servername:	JTextField
+ServerProperties	
-setupLayout:	void
-lookupBean:	DBUpdateSession
-lookupBeanHome:	DBUpdateSessionHome
+showMe:	boolean
-testConn:	void

The user has the possibility to “test” the parameters entered. If a connection is tested OK, it is indicated in `parent.props` and the other methods that need to connect to the EJB will not show this dialog, they will just connect silently.

### XMLErrorHandler

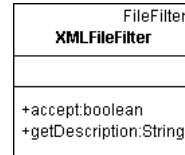
A simple error handler, enabling the application to detect errors, while using the validating XML parser in relation to “Import Classification”. This class is hooked up to the parser, and its error methods are called whenever an error occurs.

ErrorHandler	
<b>XMLErrorHandler</b>	
-parent:	CDFrame
+XMLErrorHandler	
+warning:	void
+error:	void
+fatalError:	void

This class just pops up an `InfoDialog` with a hint about what could have happened. Then it throws an exception that are caught in the places utilizing the parser, and the appropriate measures are taken; the user is alerted and exception stack-trace is generated and output on the console.

## XMLFileFilter

A simple class used for filtering out XML files in the file choosers. Used in `Grapher` in relation to import, export and data document choosing and in `GeneralServerDialog`, whenever a document need to be up- or downloaded.



## 4.4 Tests

### 4.4.1 Tests of the WH System

The WH System was not testing using some clever test strategy. The WH System is after all just a demo application, so there was no reason to test everything for every possible and impossible situation. Testing and debugging was primarily done through a web browser, and the system seems robust. Naturally there are things that could be improved (these are mentioned in the summary), but there does not seem to be any errors in the functionality.

Appendix [D](#) on page [123](#) shows some screen shots from the WH System website. There are screen shots of all facilities in the WH System and most of the screen shots are explained briefly. Users who has not yet seen the screen shots or tried the WH System website, are strongly encouraged to read this appendix.

### 4.4.2 Test of ClassificationDesigner

Since most of the functionality in the classification designer is taken care of by the `JGraph` component, testing of the designer was performed through the user interface. The application seems to be robust when using the graphical user interface. Another validation of the application is done by making sure that the files that it outputs, are valid. The XML classifications are validated using the XML Schema, and they are able to “survive” being downloaded from the server, imported, exported, and uploaded to the WH System. This indicates that the application is doing what it is supposed to.

The user manual for the designer goes through an example showing that the intended functionality of the classification designer is present and working. The manual can be found in appendix [I](#) on page [151](#).

## Chapter 5

# Discussion

### 5.1 Improvements to the WH System

An obvious improvement to the WH System would be to implement the only missing search facility: The “expand scope” query which should help the user expanding their search scope, when a search returns few or zero results.

The functionality for searching further in the “current search result” dataguide could also be improved. For the moment the user must chose a category in order to perform this search – this should probably be changed so all the categories are searched, when no category is chosen.

Another possible enhancement of the search facilities could be to combine several different queries. For example, if the user fails to choose any categories in the “advanced search” facility, the system could detect this and perform a simple search instead of showing an error page.

A page where advanced users can formulate queries using an advanced HTML form, generated from a schema extracted from the data document, would be a nice add on. The form could make it possible to search in specified site attributes. For instance, a user could specify, that he wanted to see sites inscribed in 1981 (a query based on the `<Inscribed>` property).

Some other possible improvements, that are less interesting in relation to this project, are extensions to the administration facilities. These facilities could be:

- A user interface for editing or adding site information to the database, would be a very convenient addition.
- A facility for controlling the layout of site information in the web page, that shows details about a site chosen by the user. Currently, presentation of site information is controlled by a XSLT style sheet, that must manually be edited and reinserted into the XML database.

## 5.2 Reuse of WH System Components

Some components can be reused for other purposes. The classification structure is very general and it can be used for representation of all ontologies. Obviously this can be reused together with the XQuery that transforms classifications into dataguides, and the small JavaScript for folding/unfolding categories in the dataguide. The only requirement for reuse of these components is, that there is some data suitable for being represented in a lattice structure.

The strategies for performing queries based on classifications and site data, could be reused, though it requires quite a few changes – mostly changes to the presentation layer of the application.

## 5.3 Improvements to the ClassificationDesigner

### 5.3.1 Relational Back End

As many ordinary systems based on relational database systems could benefit from the use of classifications to index data, it could be an interesting project to investigate, how much extra work there would be needed, in order to make the classification based search run on a relational back end.

The classification documents would still need to be in XML format, in order to have the benefits from the hierarchical structure. The item references in classification documents can reference anything that has a unique id. Hence it would be possible to have the classification documents classify relational data using the existing XML schema.

However the WH System would need some major adjustments to work with the relational backend.

Changing the classification designer would not be an enormous task, basically it is just a few of the dialogs that need to be changed and the method building the “All Items” list in the `EntryList` class. When the lists are generated everything should work *out of the box*.

### 5.3.2 Editing of Data Documents

It would be a handy feature if the users were able to edit data documents (documents containing site information) in the ClassificationDesigner, and this would call for the implementation of a simple XML editor.

However the documents can be edited in any editor and if the user are familiar with an XML-aware editor, that one would probably outperform the one that the classification designer would feature. But it could be made so that changes to

the data document in the classification designer would get expressed as XUpdate statements, allowing the eXist database just to alter the necessary elements, instead of having to pull the entire data collection out of the database and replacing it with an updated version.

### **5.3.3 Better List Handling**

The lists in the classification designer could be easier to use. The only way to locate sites in them are by scrolling to the site and select it. A good extension could be an option for entering a query and have the lists filter out all the sites that did not match the query. This would help users looking for some particular site locate it a lot easier.





## Chapter 6

# Conclusion

Before starting on this project we knew that sites in the World Heritage list could be classified using ontologies. Ontologies offer an ordering which is intuitive to most people. This is a big advantage because it can be used to express complex relationships between sites, on a form that is easy to understand for the ordinary users of the WH System.

One of the goals of this project was to see if semistructured data models are suitable for representing classifications of sites, as well as the sites themselves. It turns out that SSD is excellent for representing the hierarchical structure of the classifications, since it allows arbitrary nesting of structures. SSD is also suitable for storing the site information and this is due to the schema less data model that does not impose any constraints on the data.

Whereas the XML itself is a good choice for storing the site data and classifications, the tools for handling XML data is another story. Many of the technologies are very new, and most of them does not even have their final specifications defined yet. XSLT, XPath and XML Schema have been around for a while and are well-proven, while the XQuery and XML:DB specifications are still just working drafts.

Some XQuery implementations exist and in general two different approaches are taken, with respect to the changing specifications. Some implementors do not implement things that they think will change in the next draft, while others try to implement the latest working draft completely. The specification for XQuery is getting a lot closer to its final state, and the number of implementations is getting higher each month.

We have had some trouble with the different XQuery processors, it really shows that they are the “first” generation of the technology. Some of them had a lot of strange errors, that got fixed during this project. But it seems that there is quite a lot of people with interest in the processors, so many of the issues get fixed fast. Many of the XQuery processors are closely related to XSLT processors, thus giving them a head start compared to the few processors starting from scratch.

The XML databases are another story. There are not really any precedent implementations, so the different implementors have a hard time getting it all together.

Some of the players in the field of relational databases, are trying to extend their implementations, but due to the very different nature of SSD, we think that they will have a hard time making really good solutions. Oracle is probably the RDBMS vendor that is handling XQuery best, for the time being.

However some of the projects implementing native XML databases, seem to be somewhat usable at this point. One of the biggest problems for the XML databases is, that no standard for interfacing with them exists. Some of the implementations uses the XML:DB API while others invent their own API.

The current freely available XML databases are not very mature yet. The eXist database that we use for back end is working reasonably well, but it certainly still have some serious issues. For instance it truncates elements containing single characters under certain circumstances. This feature took a while to track down and “solve”. The quick fix was to append a white-space to the single characters in the data documents, thereby removing the risk of them being truncated.

Due to the mentioned issues (and a lot of other like them) with the XQuery processors and the XML databases, the implementation of the WH System took a while longer than expected. For instance the selection of database was done by selecting the one that caused us the least trouble, while the selection really should be based on the one performing best.

It would be very nice with a database that supports XQuery, this would give our system a performance boost. But it seems that we would need to postpone this project quite a while, before a free XML database implementation with XQuery support will be available.

We think that the idea of using dataguides to help users create somewhat complex queries without them even knowing about it, is good and works well. Using dataguides to represent the ontology data, makes navigation in the data intuitive for the users.

As mentioned above, many of the technologies are not really mature enough for a production system yet, but once the specifications become final, a lot of implementations will probably emerge. But as the things look at the moment, we do not recommend use of the new open source XML technologies just yet. The WH System could probably be implemented on top of a commercial XML database implementation, such as Ipedo or Tamino, but they are very expensive.

# Bibliography

- [w3c] *World Wide Web Consortium website*  
<http://www.w3.org>
- [XQ-funop] *XQuery 1.0 and XPath 2.0 Functions and Operators specification*  
<http://www.w3.org/TR/xpath-functions/>
- [XQ-lang] *XQuery 1.0: An XML Query Language*  
<http://www.w3.org/TR/xquery/>
- [dotw] *Data on the Web*  
 From Relations to Semistructured Data and XML  
 By Serge Abitebould, Peter Buneman and Dan Suciu  
 ©2000 by Morgan Kaufmann Publishers  
 ISBN: 1-55860-622-X
- [unesco] *The Unesco World Heritage site*  
<http://whc.unesco.org>
- [vr-heritage] *The VRheritage.org site*  
<http://www.vrheritage.org>
- [jgraph] *The home of JGraph*  
<http://jgraph.sourceforge.net>
- [exist] *Homepage of eXist XML database*  
<http://exist.sf.net>
- [xindice] *Homepage of Xindice XML database*  
<http://xml.apache.org/xindice>
- [tomcat] *Homepage of Jakarta Tomcat*  
<http://jakarta.apache.org/tomcat/index.html>
- [qexo] *Qexo XQuery implementation website*  
<http://www.gnu.org/software/qexo/>
- [ipsi] *IPSI-XQ XQuery implementation*  
[http://ipsi.fhg.de/oasys/projects/ipsi-xq/index\\_e.html](http://ipsi.fhg.de/oasys/projects/ipsi-xq/index_e.html)
- [java] *Sun's Java website*  
<http://www.javasoft.com>
- [PJSP] *Professional Java Server Programming*  
*J2EE 1.3 Edition*  
 By Subrahmanyam Allamaraju et al.  
 ©2001 Wrox Press

ISBN: 1-861005-37-7

[xmldb] *XML:DB website*

<http://www.xmldb.org>

[jboss] *JBoss Java application server website*

<http://www.jboss.org>

[saxon] *Saxon XQuery processor website*

<http://saxon.sourceforge.net/>

[ipedo-rev] *Review of the IPedo XML database in PC Magazine*

<http://www.pcmag.com/article2/0,4149,13139,00.asp>

[vr-heritage prototype] *Prototype of the new VRHeritage.org site*

<http://www.vrheritage.org/engine/explorer>

## Appendix A

# XML Document – CD Catalog

```
1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <CD-catalog>
3   <Album>
4     <PurchaseInfo currency="DKK"/>
5     <Artist>Red Hot Chili Peppers</Artist>
6     <Title>Blood Sugar Sex Magic</Title>
7     <RecordLabel>Warner Bros.</RecordLabel>
8     <Year>1991</Year>
9     <Genre>Rock</Genre>
10    <Track>
11      <Name>The Power Of Equality</Name>
12      <Duration>4.00</Duration>
13    </Track>
14    <Track>
15      <Name>If You Have To Ask</Name>
16      <Duration>4.11</Duration>
17    </Track>
18    <Track>
19      <Name>Breaking The Girl</Name>
20      <Duration>5.03</Duration>
21    </Track>
22    <Track>
23      <Name>Funky Munks</Name>
24      <Duration>5.22</Duration>
25    </Track>
26    <Track>
27      <Name>Suck My Kiss</Name>
28      <Duration>3.35</Duration>
29    </Track>
30    <Track>
31      <Name>I Could Have Lied</Name>
32      <Duration>4.10</Duration>
33    </Track>
34    <Track>
35      <Name>Mellowship Slinky In B Major</Name>
36      <Duration>4.00</Duration>
```

```
37     </Track>
38     <Track>
39         <Name>The Rightious And The Wicked</Name>
40         <Duration>4.05</Duration>
41     </Track>
42     <Track>
43         <Name>Give It Away</Name>
44         <Duration>4.45</Duration>
45     </Track>
46     <Track>
47         <Name>Blood Sugar Sex Magik</Name>
48         <Duration>4.31</Duration>
49     </Track>
50     <Track>
51         <Name>Under The Bridge</Name>
52         <Duration>4.34</Duration>
53     </Track>
54     <Track>
55         <Name>Naked In The Rain</Name>
56         <Duration>4.30</Duration>
57     </Track>
58     <Track>
59         <Name>Apache Rose Peacock</Name>
60         <Duration>4.43</Duration>
61     </Track>
62     <Track>
63         <Name>The Greeting Song</Name>
64         <Duration>3.14</Duration>
65     </Track>
66     <Track>
67         <Name>My Lovely Man</Name>
68         <Duration>4.45</Duration>
69     </Track>
70     <Track>
71         <Name>Sir Psycho Sexy</Name>
72         <Duration>8.24</Duration>
73     </Track>
74     <Track>
75         <Name>They're Red Hot</Name>
76         <Duration>1.44</Duration>
77     </Track>
78 </Album>
79 <Album>
80     <PurchaseInfo currency="DKK" />
81     <Artist>Pearl Jam</Artist>
82     <Title>Ten</Title>
83     <RecordLabel>Sony Music</RecordLabel>
84     <Year>1992</Year>
85     <Genre>Rock</Genre>
86     <Track>
87         <Name>Once</Name>
88         <Duration>3.51</Duration>
89     </Track>
90     <Track>
```

```
91     <Name>Even Flow</Name>
92     <Duration>4.53</Duration>
93 </Track>
94 <Track>
95     <Name>Alive</Name>
96     <Duration>5.40</Duration>
97 </Track>
98 <Track>
99     <Name>Why Go</Name>
100    <Duration>3.19</Duration>
101 </Track>
102 <Track>
103     <Name>Black</Name>
104     <Duration>5.43</Duration>
105 </Track>
106 <Track>
107     <Name>Jeremy</Name>
108     <Duration>5.18</Duration>
109 </Track>
110 <Track>
111     <Name>Oceans</Name>
112     <Duration>2.41</Duration>
113 </Track>
114 <Track>
115     <Name>Porch</Name>
116     <Duration>3.38</Duration>
117 </Track>
118 <Track>
119     <Name>Garden</Name>
120     <Duration>4.58</Duration>
121 </Track>
122 <Track>
123     <Name>Deep</Name>
124     <Duration>4.10</Duration>
125 </Track>
126 <Track>
127     <Name>Release</Name>
128     <Duration>6.30</Duration>
129 </Track>
130 <Track>
131     <Name>Alive (live)</Name>
132     <Duration>4.55</Duration>
133 </Track>
134 <Track>
135     <Name>Wash</Name>
136     <Duration>3.34</Duration>
137 </Track>
138 <Track>
139     <Name>Dirty Frank</Name>
140     <Duration>5.32</Duration>
141 </Track>
142 </Album>
143 <Album>
144     <PurchaseInfo price="8.99" currency="GBP" />
```

```
145 <Artist>Red Hot Chili Peppers</Artist>
146 <Title>By The Way</Title>
147 <RecordLabel>Warner Bros.</RecordLabel>
148 <Year>2002</Year>
149 <Genre>Rock</Genre>
150 <Track>
151   <Name>By The Way</Name>
152   <Duration>3.37</Duration>
153 </Track>
154 <Track>
155   <Name>Universally Speaking</Name>
156   <Duration>4.19</Duration>
157 </Track>
158 <Track>
159   <Name>This Is The Place</Name>
160   <Duration>4.17</Duration>
161 </Track>
162 <Track>
163   <Name>Dosed</Name>
164   <Duration>5.12</Duration>
165 </Track>
166 <Track>
167   <Name>Don't Forget Me</Name>
168   <Duration>4.37</Duration>
169 </Track>
170 <Track>
171   <Name>The Zephyr Song</Name>
172   <Duration>3.52</Duration>
173 </Track>
174 <Track>
175   <Name>Can't Stop</Name>
176   <Duration>4.29</Duration>
177 </Track>
178 <Track>
179   <Name>I Could Die For You</Name>
180   <Duration>3.13</Duration>
181 </Track>
182 <Track>
183   <Name>Midnight</Name>
184   <Duration>4.55</Duration>
185 </Track>
186 <Track>
187   <Name>Throw Away Your Television</Name>
188   <Duration>3.44</Duration>
189 </Track>
190 <Track>
191   <Name>Cabron</Name>
192   <Duration>3.38</Duration>
193 </Track>
194 <Track>
195   <Name>Tear</Name>
196   <Duration>5.17</Duration>
197 </Track>
198 <Track>
```



---

```
199     <Name>On Mercury</Name>
200     <Duration>3.28</Duration>
201 </Track>
202 <Track>
203     <Name>Minor Thing</Name>
204     <Duration>3.37</Duration>
205 </Track>
206 <Track>
207     <Name>Warm Tape</Name>
208     <Duration>4.16</Duration>
209 </Track>
210 <Track>
211     <Name>Venice Queen</Name>
212     <Duration>6.07</Duration>
213 </Track>
214 </Album>
215 <Album>
216     <PurchaseInfo price="115" currency="DKK" />
217     <Artist>D.A.D</Artist>
218     <Title>Riskin' It All</Title>
219     <RecordLabel>Medley Records</RecordLabel>
220     <Year>1991</Year>
221     <Genre>Rock</Genre>
222 <Track>
223     <Name>Bad Craziiness</Name>
224     <Duration>3.16</Duration>
225 </Track>
226 <Track>
227     <Name>D-Law</Name>
228     <Duration>3.48</Duration>
229 </Track>
230 <Track>
231     <Name>Day Of Wrong Moves</Name>
232     <Duration>3.58</Duration>
233 </Track>
234 <Track>
235     <Name>Rock'n'Rock Radar</Name>
236     <Duration>2.36</Duration>
237 </Track>
238 <Track>
239     <Name>Down That Dusty 3'rd World Road</Name>
240     <Duration>4.23</Duration>
241 </Track>
242 <Track>
243     <Name>Makin' Fun Of Money</Name>
244     <Duration>4.08</Duration>
245 </Track>
246 <Track>
247     <Name>Grow Or Pay</Name>
248     <Duration>4.59</Duration>
249 </Track>
250 <Track>
251     <Name>Smartboy Can't Tell Ya'</Name>
252     <Duration>3.15</Duration>
```

```
253     </Track>
254     <Track>
255         <Name>Riskin' It All</Name>
256         <Duration>2.37</Duration>
257     </Track>
258     <Track>
259         <Name>Laugh 'n' A½ </Name>
260         <Duration>3.24</Duration>
261     </Track>
262 </Album>
263 <Compilation>
264     <PurchaseInfo price="12.99" currency="GBP"/>
265     <Title>The very best of MTV unplugged 2</Title>
266     <RecordLabel>Warner Music and Universal International Music</
        RecordLabel>
267     <Year>2003</Year>
268     <Genre>Pop/Rock</Genre>
269     <Track>
270         <Name>Every Breath You Take</Name>
271         <Artist>Sting</Artist>
272         <Duration>5.07</Duration>
273     </Track>
274     <Track>
275         <Name>Wicked Game</Name>
276         <Artist>Chris Isaak</Artist>
277         <Duration>4.54</Duration>
278     </Track>
279     <Track>
280         <Name>Zombie</Name>
281         <Artist>The Cranberries</Artist>
282         <Duration>4.17</Duration>
283     </Track>
284     <Track>
285         <Name>Imitation Of Life</Name>
286         <Artist>R.E.M.</Artist>
287         <Duration>4.07</Duration>
288     </Track>
289     <Track>
290         <Name>Layla</Name>
291         <Artist>Eric Clapton</Artist>
292         <Duration>4.40</Duration>
293     </Track>
294     <Track>
295         <Name>Four Seasons In One Day</Name>
296         <Artist>Crowded House</Artist>
297         <Duration>5.32</Duration>
298     </Track>
299     <Track>
300         <Name>Cornflake Girl</Name>
301         <Artist>Tori Amos</Artist>
302         <Duration>5.32</Duration>
303     </Track>
304     <Track>
305         <Name>Have I Told You Lately</Name>
```

---

```
306     <Artist>Rod Stewart</Artist>
307     <Duration>3.59</Duration>
308 </Track>
309 <Track>
310     <Name>Like A Rolling Stone</Name>
311     <Artist>Bob Dylan</Artist>
312     <Duration>8.29</Duration>
313 </Track>
314 <Track>
315     <Name>Human Behaviour</Name>
316     <Artist>ØBjrk</Artist>
317     <Duration>3.24</Duration>
318 </Track>
319 <Track>
320     <Name>Crazy</Name>
321     <Artist>Seal</Artist>
322     <Duration>4.49</Duration>
323 </Track>
324 <Track>
325     <Name>Beds Are Burning</Name>
326     <Artist>Midnight Oil</Artist>
327     <Duration>4.48</Duration>
328 </Track>
329 <Track>
330     <Name>Run, Baby, Run</Name>
331     <Artist>Sheryl Crow</Artist>
332     <Duration>5.04</Duration>
333 </Track>
334 <Track>
335     <Name>I'm Ready</Name>
336     <Artist>Bryan Adams</Artist>
337     <Duration>4.25</Duration>
338 </Track>
339 <Track>
340     <Name>In The Air Tonight</Name>
341     <Artist>Phil Collins</Artist>
342     <Duration>4.57</Duration>
343 </Track>
344 <Track>
345     <Name>Don't Let The Sun Go Down On Me</Name>
346     <Artist>Elton John</Artist>
347     <Duration>5.55</Duration>
348 </Track>
349 </Compilation>
350 </CD-catalog>
```



## Appendix B

# Schema for the CD Catalog

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <schema xmlns='http://www.w3.org/2001/XMLSchema'>
3  <element name='CD-catalog'>
4    <complexType>
5      <sequence maxOccurs='unbounded'>
6        <choice>
7          <element name='Album'>
8            <complexType>
9              <sequence>
10               <element name='PurchaseInfo'>
11                 <complexType>
12                   <attribute name='price' type='decimal' minOccurs
13                     = '0' />
14                   <attribute name='currency' type='string' />
15                 </complexType>
16               </element>
17               <element name='Artist' type='string' />
18               <element name='Title' type='string' />
19               <element name='RecordLabel' type='string' />
20               <element name='Year' type='string' />
21               <element name='Genre' type='string' />
22               <element name='Track' maxOccurs='unbounded'>
23                 <complexType>
24                   <sequence>
25                     <element name='Name' type='string' />
26                     <element name='Duration' type='decimal' />
27                   </sequence>
28                 </complexType>
29               </element>
30             </sequence>
31           </complexType>
32         </choice>
33       <element name='Compilation'>
34         <complexType>
35           <sequence>

```

```
36         <complexType>
37             <attribute name='price' type='decimal' minOccurs
38                 = '0' />
39             <attribute name='currency' type='string' />
40         </complexType>
41     </element>
42     <element name='Title' type='string' />
43     <element name='RecordLabel' type='string' />
44     <element name='Year' type='string' />
45     <element name='Genre' type='string' />
46     <element name='Track' maxOccurs='unbounded'>
47         <complexType>
48             <sequence>
49                 <element name='Name' type='string' />
50                 <element name='Artist' type='string' />
51                 <element name='Duration' type='decimal' />
52             </sequence>
53         </complexType>
54     </element>
55 </sequence>
56 </complexType>
57 </element>
58 </choice>
59 </sequence>
60 </complexType>
61 </element>
62 </schema>
```

## Appendix C

# Deployment

### C.1 Deployment of the WH System

This section describes the installation procedure of the WH System *in Unix-like systems*. The root folder of the WH System application has the following structure:

**/db/** Contains all XML files needed in the XML database. The folder structure is exactly the same as it must be in the XML database.

**/descriptors/** Deployment descriptors and other J2EE configuration files.

**/jsppages/** Contains the JSP pages used in the web archive.

**/lib/** Java libraries necessary to deploy the application.

**/src/** WH System source files

**build.properties.sample** A sample build.properties file.

**build.xml** Ant configuration file.

The root folder will be referred to as `$WHSYS_HOME`

The following software must be installed before beginning the installation of the WH System (the version numbers indicate the versions we used – other versions may work as well, but we have not tested it):

- Sun's J2SDK v. 1.4.2
- Apache Ant v. 1.5.2
- JBoss J2EE application server v. 3.2
- eXist XML database v. 0.9.2

Remember to set the system variables:

- `JAVA_HOME` Must point at Java installation directory.
- `ANT_HOME` Must point at Apache Ant installation directory.
- `JBOSS_HOME` Must point at JBoss installation directory.
- `EXIST_HOME` Must point at eXist installation directory.

Go through *all* of the following steps:

- Start eXist with the command:  

```
$EXIST_HOME/bin/server.sh -p 4001 -x 4041
```

 You can start eXist on different ports, but then you must edit `$WHSYS_HOME/descriptors/ejb-jar.xml` to reflect the changes.
- Edit `$EXIST_HOME/client.properties`. Change the “uri” option to:  

```
uri=xmlldb:exist://localhost:4041/exist/xmlrpc
```
- Copy `$WHSYS_HOME/build.properties.sample` to `$WHSYS_HOME/build.properties` and begin editing the file. The `libdir` attribute should be the full path to:  

```
$WHSYS_HOME/lib
```

**MAKE SURE THIS DIRECTORY IS READABLE BY THE USER JBOSS RUNS AS.**  
 The `jbossServerDir` attribute should be the full path to:  

```
$JBOSS_HOME/server/default/deploy
```

**MAKE SURE THIS DIRECTORY IS WRITABLE BY THE USER YOU USE FOR THE INSTALLATION.**
- Now the XML files in `$WHSYS_HOME/db/` must be inserted in the database. This is simply done by running the command: `sh inidb.sh` from the directory `$WHSYS_HOME/db/`.
- Create an eXist user called “whuser” using the eXist client tool (run `EXIST_HOME/bin/client.sh`. The “whuser” must be owner all the `wh` collection and all subcollections and XML files in the subcollections. We used the password “fraggel” for the user. If you use a different password you must edit the file:  

```
$WHSYS_HOME/descriptors/ejb-jar.xml
```

 to reflect the changes.
- Start JBoss with the command: `$JBOSS_HOME/bin/run.sh`
- Now it is time to install the application. Go to the root directory `$WHSYS_HOME` and run the command: `ant jbossinstall`. This command should compile the application and automatically install it into JBoss.
- Pray it works!!

## C.2 Deployment of the ClassificationDesigner

Deployment of the ClassificationDesigner is described in the first section of the classification designer manual in appendix I on page 151.



## Appendix D

# Screen Shots of the WH Web Application

### D.1 Welcome Page


Visitors to the website initially arrives at this page.



The screenshot shows the 'WORLD HERITAGE EXPLORER' website. The header includes a globe icon and the text 'WORLD HERITAGE EXPLORER'. Below the header is a navigation menu with links: 'Home | Simple Search | Advanced Search | Search for Categories | About'. The main content area features a welcome message: 'Welcome to the World Heritage demo application'. This is followed by a paragraph stating: 'This website provides several facilities for searching the WH list for sites. Notice that this is *not* an official website for World Heritage. This website is made as part of a Master Thesis project at the Technical University of Denmark.' Below this is another paragraph: 'Please use the links in the top of the page for navigation.' and the word 'Enjoy!'. To the left of the text are five small square images: a book cover, a mountain landscape, a lighthouse, a mountain range, and a person on a beach. The footer contains the text: 'A master thesis project by Chris Poulsen and Martin R. N. Christensen Website design by the World Heritage group'.

## D.2 About Page

The about page has information about the project.


WORLD HERITAGE EXPLORER

[Home](#) | [Simple Search](#) | [Advanced Search](#) | [Search for Categories](#) | [About](#)

### About this website

This website is a demo application created as part of a Master Thesis project by:

Chris Poulsen and  
Martin Christensen

### Abstract

*World Heritage (WH) is an organization which aims at preserving particularly interesting areas, monuments etc. Each site is described on a website.*

*In order to help users navigate the existing World Heritage website, some categorizations have been created, for instance it is possible to browse categories based on location or site type.*

*It is difficult to make good categorizations and take advantage of the possibilities that they offer. But good categorizations expresses a lot of information about the sites that they cover. Categorizations can be used to make some complex queries, for example it is possible to suggest sites that are related to each other based on some category property.*


*The goal of this project is to suggest a way of making categorizations of semis structured data, explore the possibilities that categorizations of semistructured data offer, generate a framework that supports easy generation of categorizations, explore how queries can take advantage of categorizations and how query results can be presented to the user in a usable manner.*

*Because of the wide variety in the available information about the different World Heritage sites, a semi structured data model is created and a software system that illustrates some of the different principles is modeled and implemented using Open Source Software and existing XML recommendations, from the World Wide Web Consortium, such as XQuery and XPath.*

A master thesis project by Chris Poulsen and Martin R. N. Christensen  
 Website design by the World Heritage group






## D.3 Search for Categories

The “search for categories” page with the help menu shown. The keyword “asia” is entered.

**WORLD HERITAGE EXPLORER**

[Home](#) | [Simple Search](#) | [Advanced Search](#) | [Search for Categories](#) | [About](#)

### Use the form below to search for categories in the dataguides



**[ - ] HELP**

This search facility searches for categories by keyword and *not* the sites in the WH list. You could for example search for "asia", which will return all categories containing the word "asia" in the name

You must enter at least one keyword in order to search for categories. Keywords must be seperated by space.

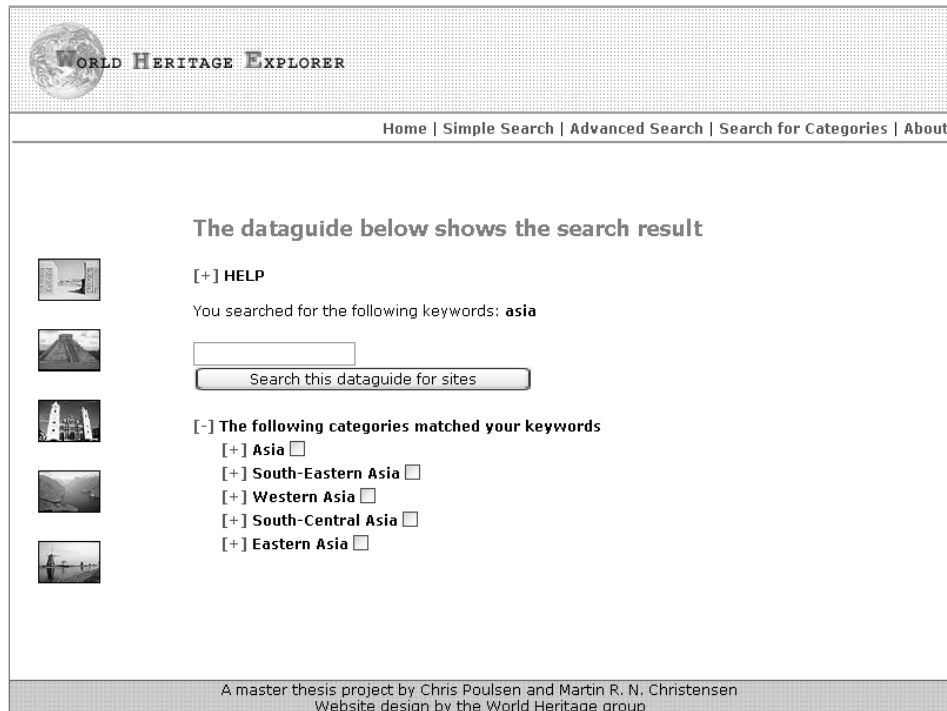
You can enter as many keywords as you want - the system will search for all the keywords. The search is case insensitive.

You can use regular expressions in the keywords. The regular expression: `.*d[ea]nmark` would for instance return all categories containing a word that has "denmark" or "danmark" in it.

A master thesis project by Chris Poulsen and Martin R. N. Christensen  
Website design by the World Heritage group

## D.4 Result of Search for Categories


Result of the search for categories containing the keyword “asia”.



The screenshot shows the 'World Heritage Explorer' website interface. At the top left is the logo, and at the top right is a navigation menu with links for 'Home', 'Simple Search', 'Advanced Search', 'Search for Categories', and 'About'. The main content area features a vertical sidebar of five small images on the left. The main text area displays the heading 'The dataguide below shows the search result' followed by a '[+] HELP' link. Below this, it states 'You searched for the following keywords: asia' and includes a search input field with a 'Search this dataguide for sites' button. A section titled '[-] The following categories matched your keywords' lists four categories, each with a '[+] ' and a checkbox: 'Asia', 'South-Eastern Asia', 'Western Asia', 'South-Central Asia', and 'Eastern Asia'. The footer contains the text: 'A master thesis project by Chris Poulsen and Martin R. N. Christensen' and 'Website design by the World Heritage group'.






## D.5 Advanced Search Form

Advanced search form. The dataguide *classification by geographical location* is unfolded, and the categories *France, Germany, Netherlands and Northern Europe* has been selected. The keywords “castle”, “fort” and “brick” has been entered in the keyword text field.


WORLD HERITAGE EXPLORER

[Home](#) | [Simple Search](#) | [Advanced Search](#) | [Search for Categories](#) | [About](#)

**Advanced search facility:**

**[+] HELP**

Use the form below, if you want to make an advanced query in the WH site list

castle fort brick


Search in selected subjects

- **[-] Classification by geographical location**  
 This dataguide categorizes the WH sites by their geographical location around the world
  - [-] All sites in geographical location classification**
  - [+] Oceania
  - [+] Asia
  - [+] Americas
  - [-] Europe 
    - [-] Western Europe 
      - [+] Switzerland
      - [+] France
      - [+] Germany
      - [+] Netherlands
      - [-] Liechtenstein
      - [+] Ireland
      - [-] Monaco
      - [+] Belgium
      - [+] Luxembourg
      - [+] Austria
    - [+] Eastern Europe
    - [+] Southern Europe
    - [+] Northern Europe
  - [+] Africa
- **[+] Classification by miscellaneous categories**






A master thesis project by Chris Poulsen and Martin R. N. Christensen  
 Website design by the World Heritage group

## D.6 Advanced Search Result as Dataguide

Result of the search from the previous screen shot. The result is shown as a dataguide – the user has unfolded some of the categories in the dataguide.


WORLD HERITAGE EXPLORER

[Home](#) | [Simple Search](#) | [Advanced Search](#) | [Search for Categories](#) | [About](#)

The dataguide below shows the search result

**[+] HELP**

You searched for the following keywords: **castle,fort,brick**

You received 25 hits from the query.

Search this dataguide for sites

**[-] Search results**


- [-] All sites in geographical location classification**
- [-] Europé** 
  - [-] Western Europé** 
    - [+] France**
    - [+] Germany**
    - [+] Netherlands**
  - [-] Northern Europé** 
    - [+] Sweden**
    - [-] Denmark** 
      - Roskilde Cathedral
      - Kronborg Castle
    - [+] United Kingdom**
    - [+] Estonia**
    - [+] Finland**

• [View result as a list](#)

A master thesis project by Chris Poulsen and Martin R. N. Christensen  
 Website design by the World Heritage group

## D.7 Advanced Search Result as List

Result of the search from the previous screen shot. The result is shown as a list. Notice the number in parentheses behind the site names are the *hitpoints* assigned to the sites. The sites with most hitpoints are placed in the top of the list.


WORLD HERITAGE EXPLORER

[Home](#) | [Simple Search](#) | [Advanced Search](#) | [Search for Categories](#) | [About](#)

The dataguide below shows the search result

[+] **HELP**

You searched for the following keywords: **castle,fort,brick**

You received 25 hits from the query.

Search this dataguide for sites

- Castles and Town Walls of King Edward in Gwynedd (3)
- Castles of Augustusburg (2)
- Durham Castle and Cathedral (2)
- Kronborg Castle (2)
- The Defence Line of Amsterdam (2)
- The Historic Fortified City of Carcassonne (2)
- The Historic Town of St George and Related Fortifications, Bermuda (2)
- Wartburg Castle (2)
- Collegiate Church, Castle and (1)
- Fortress of Suomenlinna (1)
- Hanseatic City of Lübeck (1)
- Hanseatic Town of Visby (1)
- Historic Centre of (1)
- Historic Centres of Stralsund and Wismar (1)
- Loire Valley between Sully-sur-Loire and Chalonnes (1)
- Maulbronn Monastery (1)
- Old and New Towns of Edinburgh (1)
- Provins, Town of Medieval Fairs (1)
- Roskilde Cathedral (1)
- Royal Domain of Drottningholm (1)
- Studley Royal Park including the Ruins of Fountains Abbey (1)
- The Historic Centre (Old Town) of Tallinn (1)
- The Luther Memorials in Eisleben and Wittenberg (1)
- Tower of London (1)
- Upper Middle Rhine Valley (1)

• [View result as a dataguide](#)

A master thesis project by Chris Poulsen and Martin R. N. Christensen  
 Website design by the World Heritage group

## D.8 Site Information about Kronborg Castle

Information about the site “Kronborg Castle”.


WORLD HERITAGE EXPLORER

[Home](#) | [Simple Search](#) | [Advanced Search](#) | [Search for Categories](#) | [About](#)

### Kronborg Castle

**Brief Description:**

Located on a strategically important site commanding the Sund, the stretch of water between Denmark and Sweden, the Royal castle of Kronborg at Helsingør (Elsinore) is of immense symbolic value to the Danish people and played a key role in the history of northern Europe in the 16th-18th centuries. Work began on the construction of this outstanding Renaissance castle in 1574, and its defences were reinforced according to the canons of the period's military architecture in the late 17th century. It has remained intact to the present day. It is world-renowned as Elsinore, the setting of Shakespeare's Hamlet.

**Justification for Inscription**

Criterion: C (iv)  
Kronborg Castle is an outstanding example of the Renaissance castle, and one which played a highly significant role in the history of this region of northern Europe.

**Other resources:**

**Reports:**  
Report of the 24th Session of the Committee

**Other links:**  
Kronborg Castle (Official Web site)  
Kronborg Castle (Slots- og Ejendomsstyrelsen / Castles and Properties Agency)


- [Find similar sites](#)

A master thesis project by Chris Poulsen and Martin R. N. Christensen  
 Website design by the World Heritage group







## D.9 Site Information about Roskilde Cathedral

Information about the site “Roskilde Cathedral”. Notice that there is some additional information, which was not present in the previous screen shot (Kronborg). This site has information about its location and its architectural style (in the bottom of the screen shot).


WORLD HERITAGE EXPLORER

[Home](#) | [Simple Search](#) | [Advanced Search](#) | [Search for Categories](#) | [About](#)



### Roskilde Cathedral

---

Denmark
Island of Sjaelland  
12 ° 4 ' 47.2E, 55 ° 38 ' 32.5N

---

**Brief Description:**

Built in the 12th and 13th centuries, this was Scandinavia's first Gothic cathedral to be built of brick and it encouraged the spread of this style throughout northern Europe. It has been the mausoleum of the Danish royal family since the 15th century. Porches and side chapels were added up to the end of the 19th century. Thus it provides a clear overview of the development of European religious architecture.

**Justification for Inscription**

Criterion: C (ii)  
exhibit an important interchange of human values, over a span of time or within a cultural area of the world, on developments in architecture or technology, monumental arts, town-planning or landscape design.

Criterion: C (iv)  
be an outstanding example of a type of building or architectural or technological ensemble or landscape which illustrates (a) significant stage(s) in human history.

**Inscribed:** 1995

Other resources:

**Reports:**  
Report of the 19th Session of the Committee

**Other links:**  
National Museum of Denmark

Other properties of this site:

**ArchitecturalStyle:** Gothic

- [Find similar sites](#)

A master thesis project by Chris Poulsen and Martin R. N. Christensen  
 Website design by the World Heritage group

## D.10 “Similar Sites” to Roskilde Cathedral

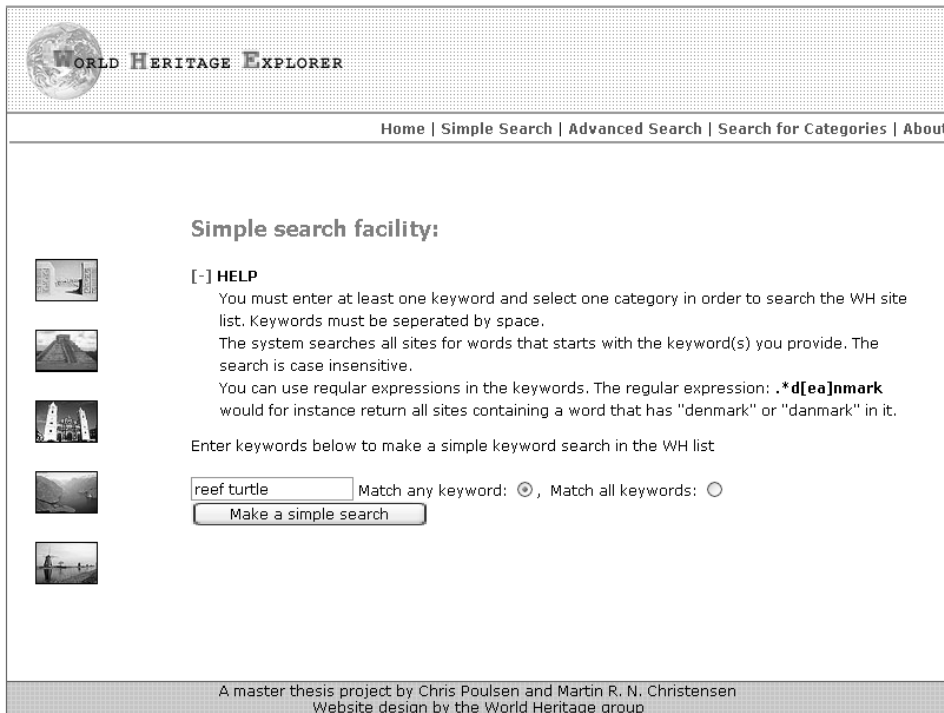
Shows sites similar to Roskilde Cathedral. The site is only categorized as a danish site and as a cultural site, so these two categories are part of the result dataguide.



The screenshot displays the 'World Heritage Explorer' interface. At the top left is the logo, and at the top right is a navigation menu with links for Home, Simple Search, Advanced Search, Search for Categories, and About. The main content area features a heading 'The dataguide below shows related sites' followed by a list of filters: '[+] HELP', '[-] Sites in same categories as chosen site', '[+] Denmark ', and '[+] All Cultural Properties '. Below these are two bullet points: '• Show one more ancestor level' and '• Search this dataguide'. On the left side of the dataguide, there are five small thumbnail images representing different heritage sites. At the bottom of the page, a footer contains the text: 'A master thesis project by Chris Poulsen and Martin R. N. Christensen' and 'Website design by the World Heritage group'.

## D.11 Simple Search Form, Showing Help Info

The simple search facility with the help menu shown. The keywords “reef” and “turtle” has been entered. The *match any keyword* option is selected.



The screenshot shows the 'World Heritage Explorer' website's search interface. At the top, there is a navigation bar with links for 'Home', 'Simple Search', 'Advanced Search', 'Search for Categories', and 'About'. The main content area is titled 'Simple search facility:' and includes a '[-] HELP' section. The help text explains that users must enter at least one keyword and select a category, and that the search is case-insensitive. It also provides a regular expression example: `.*d[ea]nmark`. Below the help text, there is a text input field containing 'reef turtle' and two radio buttons: 'Match any keyword:' (which is selected) and 'Match all keywords:'. A 'Make a simple search' button is located below the input field. On the left side of the search area, there are five small thumbnail images representing different World Heritage sites. At the bottom of the page, there is a footer with the text: 'A master thesis project by Chris Poulsen and Martin R. N. Christensen' and 'Website design by the World Heritage group'.









## D.12 Result of a Simple Search

Result of the simple search for the keywords “reef” and “turtle”. The search result is simply a list with all matching sites. The brief descriptions are shown in the list, and the site names are links to the detailed site information.


WORLD HERITAGE EXPLORER

[Home](#) | [Simple Search](#) | [Advanced Search](#) | [Search for Categories](#) | [About](#)

The keyword search returned the following results

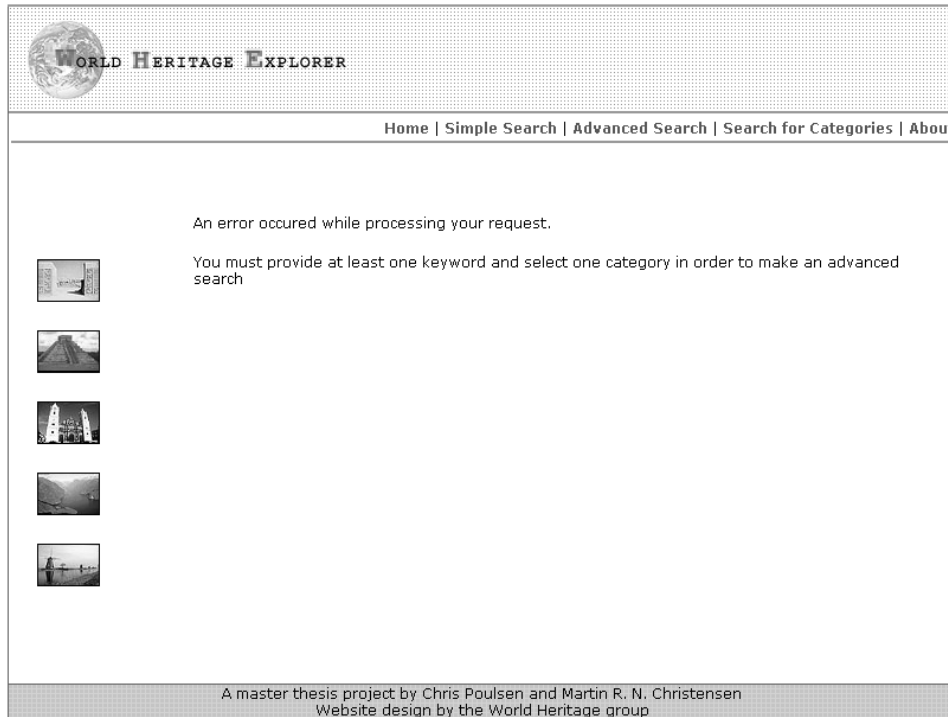









- **Great Barrier Reef**  
The Great Barrier Reef is a site of remarkable variety and beauty on the north-east coast of Australia. It contains the world's largest collection of coral reefs, with 400 types of coral, 1,500 species of fish and 4,000 types of mollusc. It also holds great scientific interest as the habitat of species such as the dugong ('sea cow') and the large green turtle, which are threatened with extinction.
- **Belize Barrier- Reef Reserve System**  
The coastal area of Belize is an outstanding natural system consisting of the largest barrier reef in the northern hemisphere, offshore atolls, several hundred sand cays, mangrove forests, coastal lagoons and estuaries. The system's seven sites illustrate the evolutionary history of reef development and are a significant habitat for threatened species, including marine turtles, manatees and the American marine crocodile.
- **Brazilian Atlantic Islands: Fernando de Noronha and Atol das Rocas Reserves**  
Peaks of the Southern Atlantic submarine ridge form the Fernando de Noronha Archipelago and Rocas Atoll off the coast of Brazil. They represent a large proportion of the island surface of the South Atlantic and their rich waters are extremely important for the breeding and feeding of tuna, shark, turtle and marine mammals. The islands are home to the largest concentration of tropical seabirds in the Western Atlantic. Baia de Golfinhos has an exceptional population of resident dolphin and at low tide the Rocas Atoll provides a spectacular seascape of lagoons and tidal pools teeming with fish.
- **Banc d'Arguin National Park**  
Fringing the Atlantic coast, the park is made up of sand dunes, coastal swamps, small islands and shallow coastal waters. The austerity of the desert and the biodiversity of the marine zone result in a land and seascape of exceptional contrasting natural value. A wide variety of migrating birds spend the winter there. Several species of sea turtle and dolphin, which fishermen use to attract shoals of fish, can also be found.
- **Sian Kaan**  
In the language of the Mayan peoples who once inhabited this region, Sian Ka'an means "origin of the sky". Located on the east coast of the Yucatan peninsula, this biosphere reserve contains tropical forests, mangroves and marshes, as well as a large marine section intersected by a barrier reef, and provides a habitat for an abundance of fauna and flora.
- **Whale Sanctuary of El Vizcaino**  
Located in the central part of the peninsula of Baja California, the sanctuary contains exceptionally interesting ecosystems. The coastal lagoons of Ojo de Liebre and San Ignacio are very important reproduction and wintering sites for the grey whale, harbour seal, California sea-lion, northern elephant-seal and blue whale. The lagoons also offer shelter to four species of the endangered marine turtle.
- **Tubbatha Reef Marine Park**  
The park covers 33,200 hectares, including the North and South Reefs, and is a unique example of an atoll reef with a very high density of marine species. The North Islet serves as a nesting site for birds and marine turtles. The site is an excellent example of a pristine coral reef with a spectacular 100-metre perpendicular wall, extensive lagoons and two coral islands.
- **Aldabra Atoll**  
The atoll is comprised of four large coral islands which enclose a shallow lagoon; the group of islands is itself surrounded by a coral reef. Due to difficulties of access and the atoll's isolation, Aldabra has been protected from human influence and has as such become a refuge for some 152,000 giant tortoises, the world's largest population of this reptile.

A master thesis project by Chris Poulsen and Martin R. N. Christensen  
 Website design by the World Heritage group

## D.13 Error Page

An error page - the user forgot to enter a keyword or select a category when trying to use the advanced search facility.





## **Appendix E**

# **EJB Classes from the WH System**

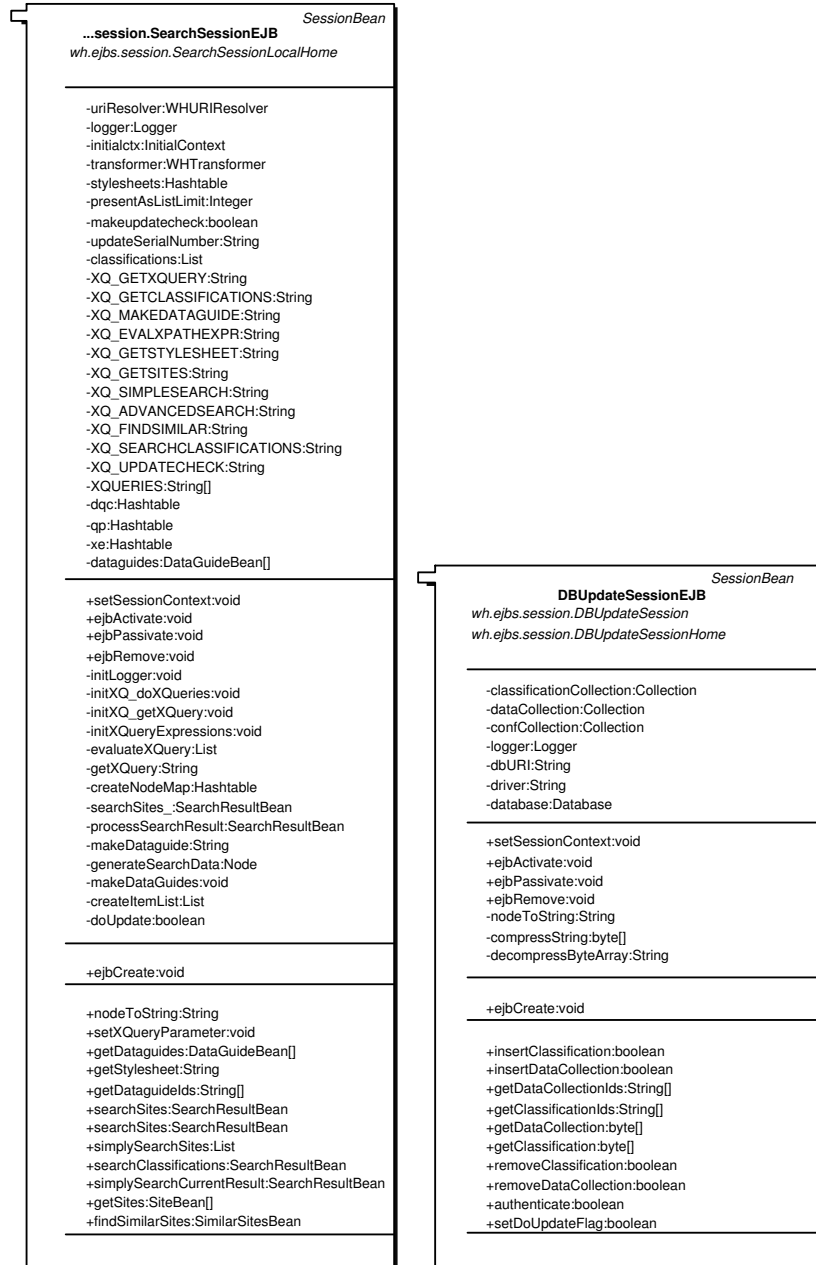


Figure E.1: EJB classes from the WH system



## Appendix F

# XPath Performance Tests

### F.1 Test Programs for eXist

The tests of eXist uses the full data document – this has <Sites> as root element. Additionally there is a document containing only one site – this document has SmallSites as root element.

#### 0 Sites, 1 Keyword

```
echo "/NonExisting/Site[match-any(.,'shark')]/Name/text()" | exist -c /db/wh/DataCollection -x
```

#### 1 Site, 1 Keyword

```
echo "/SmallSites/Site[match-any(.,'shark')]/Name/text()" | exist -c /db/wh/DataCollection -x
```

#### 730 Sites, 1 Keyword

```
echo "/Sites/Site[match-any(.,'shark')]/Name/text()" | exist -c /db/wh/DataCollection -x
```

#### 1 Site, 10 Keywords

```
echo "/SmallSites/Site[match-any(.,'shark', 'animal','forest','castle','kronborg','reef','ocean','plant','whale','ruin')]/Name/text()" | exist -c /db/wh/DataCollection -x
```

#### 730 Sites, 10 Keywords

```
echo "/Sites/Site[match-any(.,'shark', 'animal','forest','castle','kronborg','reef','ocean','plant','whale','ruin')]/Name/text()" | exist -c /db/wh/DataCollection -x
```

## F.2 Test Programs for Saxon

The tests uses the files “datadoc.xml”, which is contains all sites in the WH list – the root element is <Sites>, and “small.xml” that contains only one site and has SmallSites as root element.

### 0 Sites, 1 Keyword

```
doc("small.xml")/NonExisting/Site[contains(string(.), 'Darwin')]/
  Name/text()
```

### 1 Site, 1 Keyword

```
for $site in doc("small.xml")/SmallSites/Site
let $text := string($site)
where contains($text, 'Darwin') return $site/Name/text()
```

### 730 Sites, 1 Keyword

```
for $site in doc("datadoc.xml")/SmallSites/Site
let $text := string($site)
where contains($text, 'Darwin') return $site/Name/text()
```

### 1 Site, 10 Keywords

```
for $site in doc("small.xml")/SmallSites/Site
let $text := string($site)
where
  contains($text, 'Darwin')
  or contains($text, 'animal')
  or contains($text, 'forest')
  or contains($text, 'castle')
  or contains($text, 'kronborg')
  or contains($text, 'reef')
  or contains($text, 'ocean')
  or contains($text, 'plant')
  or contains($text, 'whale')
  or contains($text, 'ruin')
return $site/Name/text()
```

### 730 Sites, 10 Keywords

```
for $site in doc("datadoc.xml")/SmallSites/Site
let $text := string($site)
where
  contains($text, 'Darwin')
  or contains($text, 'animal')
  or contains($text, 'forest')
  or contains($text, 'castle')
```

---

```
    or contains($text,'kronborg')
    or contains($text,'reef')
    or contains($text,'ocean')
    or contains($text,'plant')
    or contains($text,'whale')
    or contains($text,'ruin')
return $site/Name/text()
```



## Appendix G

# Classification Schema

```

1 <?xml version='1.0' encoding="UTF-8"?>
2 <xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
3 <xsd:element name='Classification'>
4 <xsd:complexType>
5 <xsd:sequence>
6 <xsd:element name='DisplayName' type='xsd:string' />
7 <xsd:element name='Description' type='xsd:string' />
8 <xsd:element name='Keywords' type='xsd:string' />
9 <xsd:element name='Stylesheet' type='xsd:string' minOccurs="
  0" />
10 <xsd:element name='DataDocument' type='xsd:string' />
11 <xsd:element name='ClassEntryName' type='xsd:string' />
12 <xsd:element name='EntityListQuery' type='xsd:string' />
13 <xsd:element ref='Class' />
14 </xsd:sequence>
15 <xsd:attribute name='id' type='xsd:string' />
16 </xsd:complexType>
17 </xsd:element>
18 <!-- definitions -->
19 <xsd:element name='ItemRef'>
20 <xsd:complexType>
21 <xsd:sequence>
22 <xsd:element name='DisplayName' type='xsd:string' />
23 <xsd:element name='ItemIdref' type='xsd:string' />
24 </xsd:sequence>
25 </xsd:complexType>
26 </xsd:element>
27
28 <xsd:element name="Class">
29 <xsd:complexType>
30 <xsd:sequence minOccurs="0">
31 <xsd:element name="DisplayName" type="xsd:string" />
32 <xsd:element name="Description" type="xsd:string" minOccurs="
  0" />
33 <xsd:element name="Keywords" type="xsd:string" minOccurs="0"
  />

```

```
34     <xsd:element ref="ItemRef" minOccurs="0" maxOccurs="
      unbounded"/>
35     <xsd:element ref="Class" minOccurs="0" maxOccurs="unbounded"
      />
36 </xsd:sequence>
37 <xsd:attribute name="id" type="xsd:string" use="optional"/>
38 <xsd:attribute name="idref" type="xsd:string" use="optional"/>
39 </xsd:complexType>
40 </xsd:element>
41 </xsd:schema>
```

## **Appendix H**

# **Database Schema for the Current WH Relational Database**

The schemas on the following pages are parts of a big schema. The first three pages are connected the following way: With the first page at the top, second page beneath it and the third page at the bottom. The fourth page is not connected to anything in the first three pages.

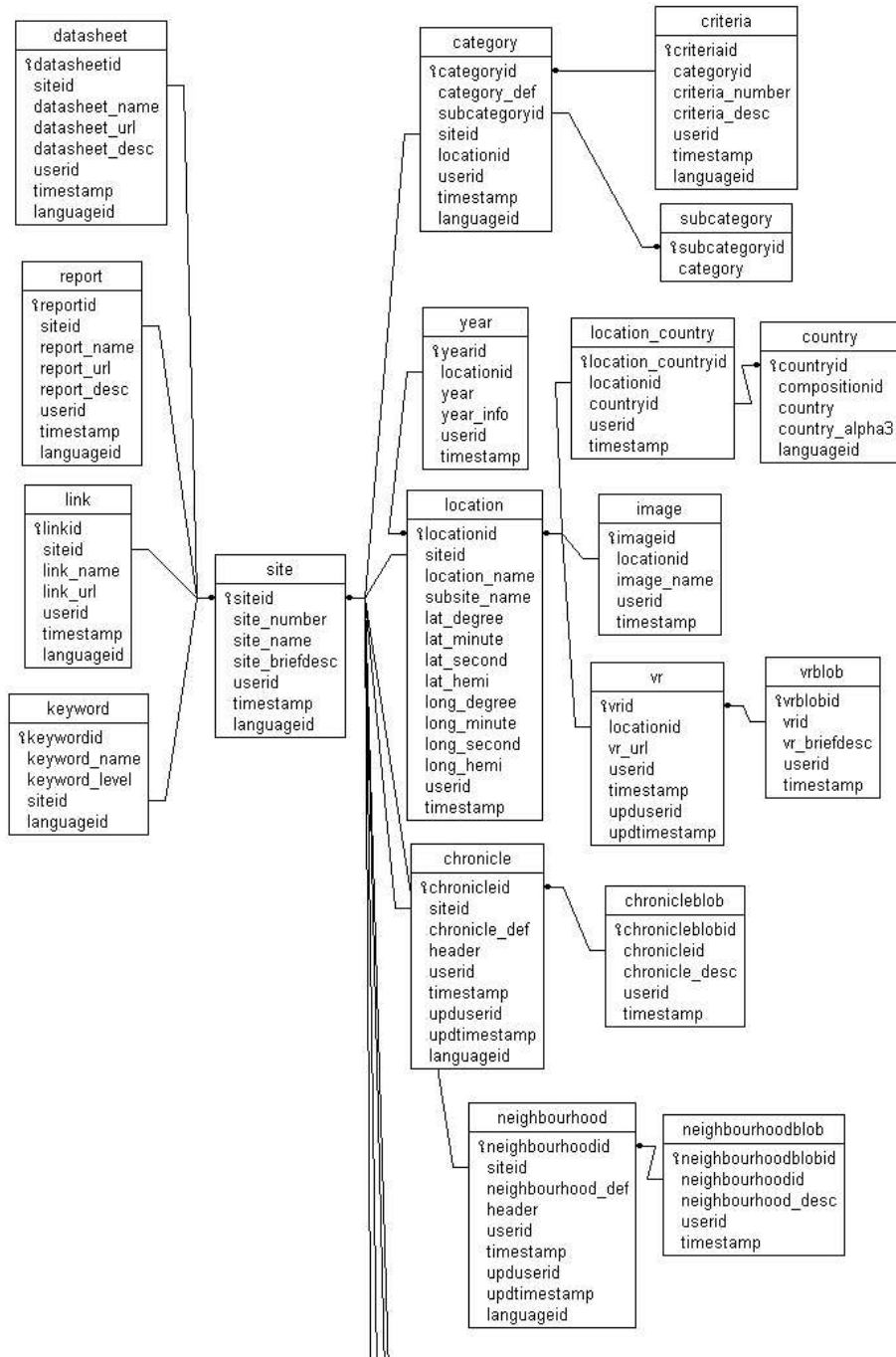


Figure H.1: Schema for the relational database



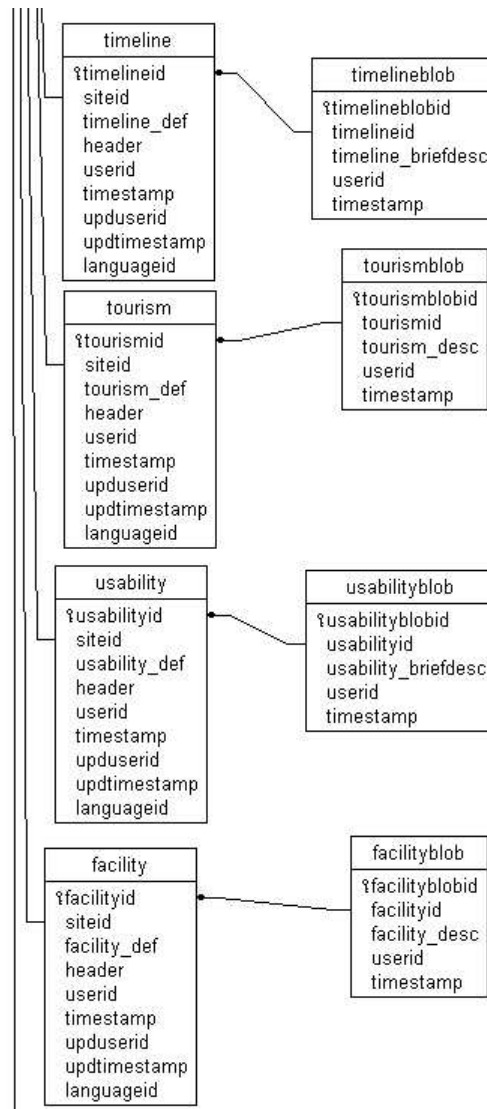


Figure H.2: Schema for the relational database

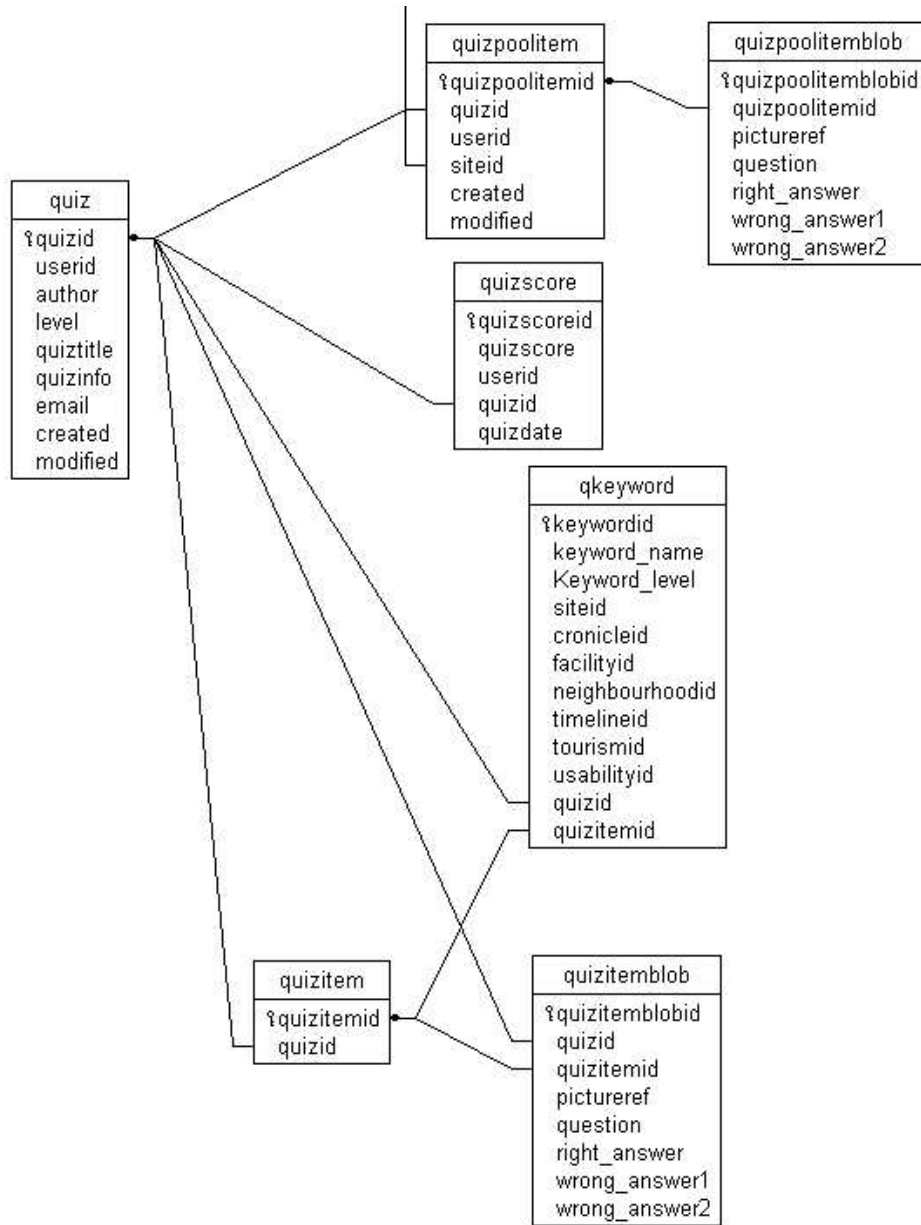


Figure H.3: Schema for the relational database

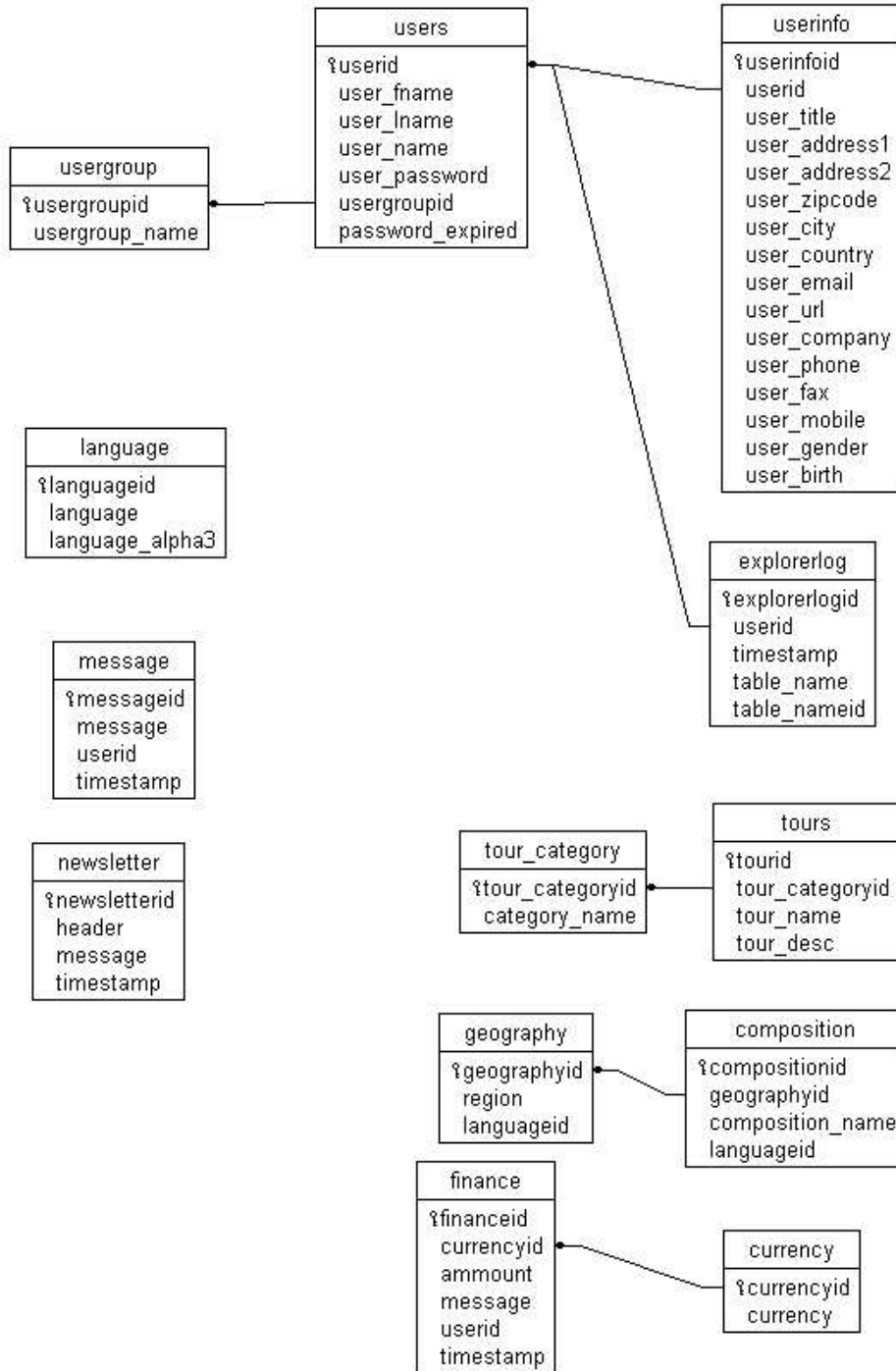


Figure H.4: Schema for the relational database



## Appendix I

# User Manual for ClassificationDesigner

### I.1 Installation

The ClassificationDesigner (CD) comes as a single file, called “CDesigner.jar”. The CD is a Java application and runs in Java version 1.4 or later. The application can be started by typing:

```
java -jar CDesigner.jar
```

In the folder where the jar file is located. In order to be able to validate the classifications, they specify an XML Schema internally. This schema is called: `classification.xsd` and should be found together with the `CDesigner.jar` file.

*Note to Windows users, for some reason the Java environment was not working correctly on our test machine, that caused the XML files downloaded from the server to be in a wrong encoding (not UTF-8). The solution is to force Java to use the right encoding by starting the application with this line:*

```
java -Dfile.encoding=UTF-8 -jar CDesigner.jar
```

### I.2 Making a Sample Classification

Once the application is started, as mentioned in the last section, the user is met with a screen that looks like the one in figure [I.1](#) on the following page.

First thing to do when creating a new application is to get a copy of the data document that the server (the World Heritage System) uses. This is done by selecting

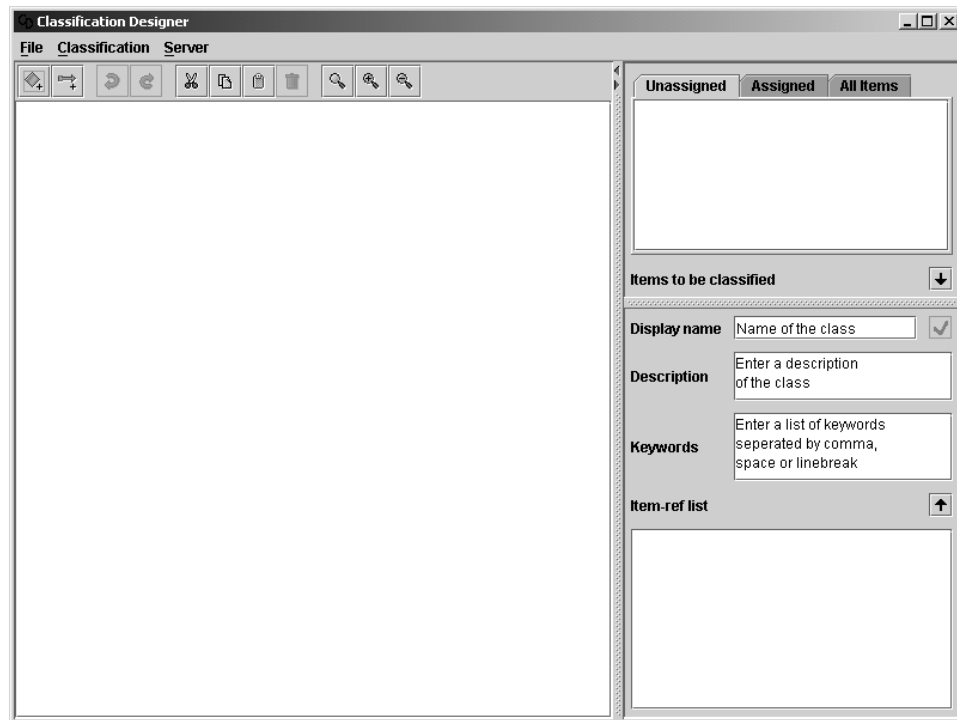


Figure I.1: Designer Startup Screen.

the **Server**→**Download Data Document**. If this is the first time the connection to the server is performed, a dialog - “Server properties” is shown, asking for the name/IP of the WH System server. The connection test should be performed and an “Info” dialog will popup informing the user whether the test succeeded or not. A successful test is shown in figure [I.2](#) on the next page.

If the connection test is successful, the user is asked to authenticate himself. This is done by putting in a username and a password for the WH System. After a user has been authenticated successfully, the system will remember the credentials for the remainder of the session. The authentication dialog is shown in figure [I.3](#) on page [154](#).

After completing the connection test and supplying proper credentials, the system shows a list of data documents stored in the WH System. The WH System uses a single document for data. A download location is specified, document on the server in the list is selected and the “Download” button is pressed.

Please note that the data document usually is large, and that it may take a while to download it. The download time depends on your network connection. While the application is downloading the XML document, the application may appear to be “frozen”, because it is waiting for the download to finish.

When the download finishes a dialog pops up informing the user the status of the

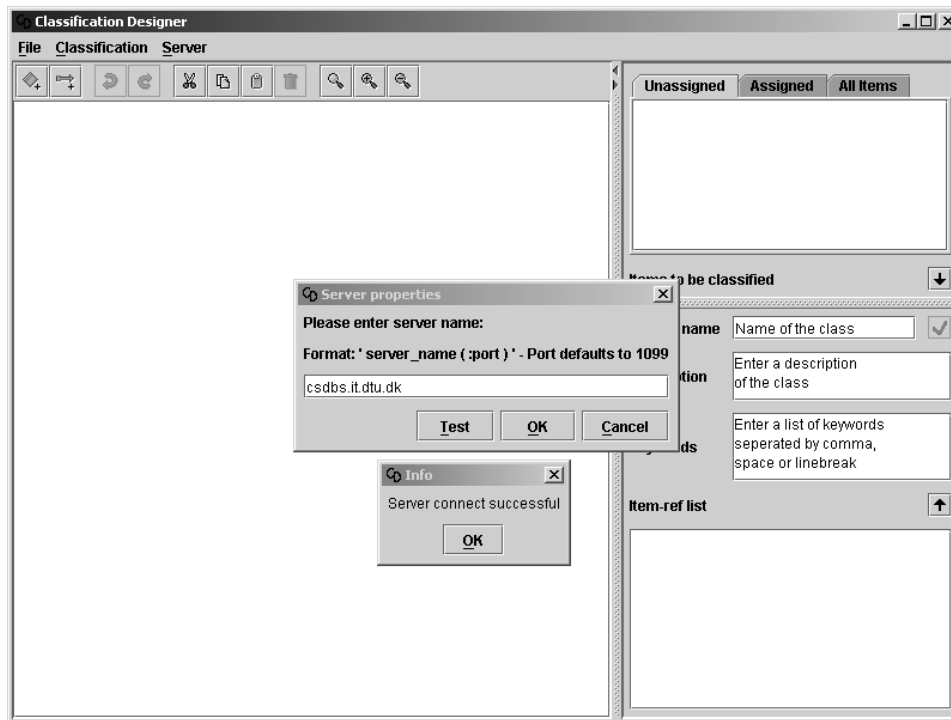


Figure I.2: Server Test Screen.

download. This is shown in figure I.4 on page 155.

Having a data document on the local hard drive, we are ready to create classifications. The easiest way to get started is to hit **File**→**New**, this starts a sequence of dialogs guiding the user through the process of entering the necessary information for the classification.

First a data document for the classification should be specified. The data document is essential for the classification. We point to the data document that was just downloaded. After the path is entered the screen looks like the one in figure I.5 on page 156. After hitting “OK” the application tried to analyze the data document to see what possibilities it offers. If the analyze goes well, A dialog for entering classification properties is shown. This dialog looks like the one in figure I.6 on page 157. This is the place where the name, description etc. for the classification is entered. The “Class Entry Element” combo box is essential. This box allows the user to specify, which structure in the data document that should be classified. In this example we are interested in classifying the “Site” elements in the data document, hence “Site” is selected. In order to be a element that can be classified, the element should contain at least one simple element (for the presentation), and a unique id attribute called *id*.

After entering valid information about the classification, it is time to specify how

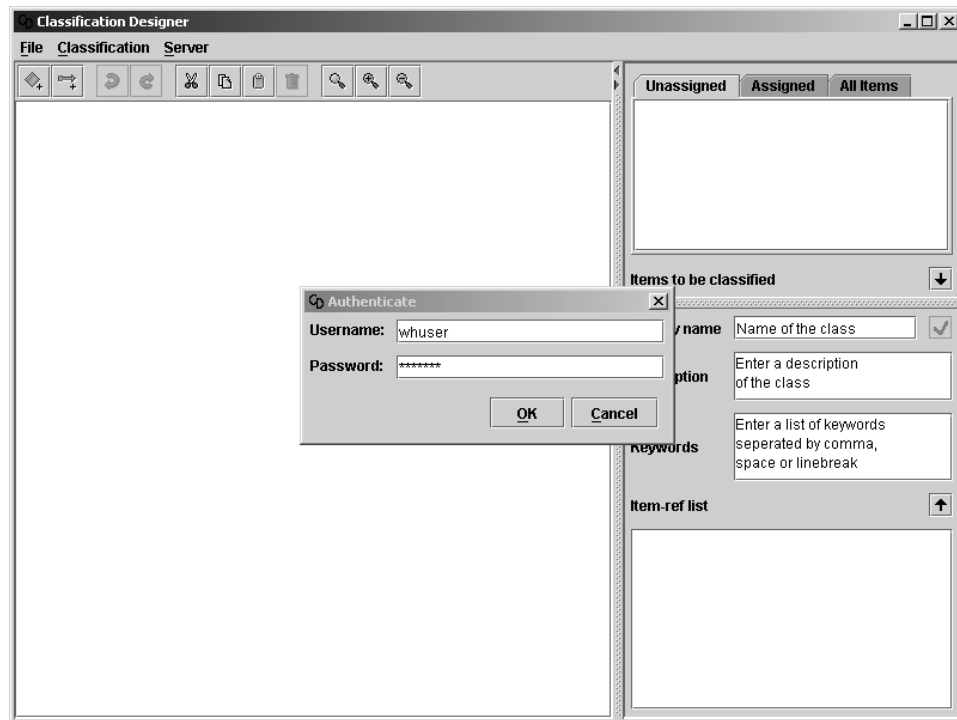


Figure I.3: Server Authentication Screen.

the elements from the data document should be presented in the classification. A dialog for selecting the format is illustrated in figure I.7 on page 158. Based on the value selected in “Class Entry Element” in the former dialog, this dialog analyzes the “Site” element to see which elements that can be candidates for a presentation format.

In the figure illustrating the example, it is specified that the value of the “Name” element in “Site” should be used for the presentation format. For site number 1, this presentation format would yield: “Galapagos Islands”.

Now the necessary information for creating a classification is entered into the system and the process of creating the structure of the classification can be initiated.

This example makes a small classification to illustrate the process. The example classification is illustrated in figure I.8 on page 159.

The classification is created by creating the nodes (representing the classes) first. A node is created by clicking on the left-most icon on the tool bar (just below “File”). The node is then selected, dragged to its place in the canvas, and data describing the node is entered in the lower right corner. The sites that should belong to a given class are selected in the lists in the upper right corner and “moved” down into the selected node by hitting the arrow pointing down.

The classes can be connected with arrows by clicking in the middle of a node and



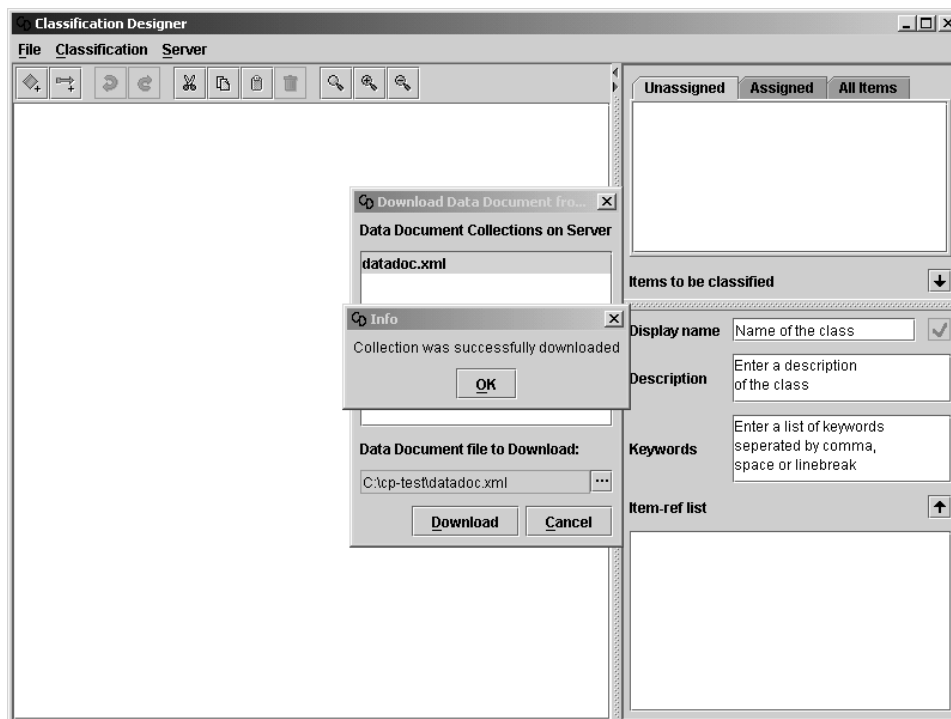


Figure I.4: Data Document Download Complete.

then dragging the arrow into another node. If a node, edge or maybe a selection of them need to be removed, just select them and hit the “delete” key on the keyboard.

The example classification puts a few of the sites into the different classes, just to have a little demo that we can upload to the WH System.

After the classification is completed, the classification should be validated. This is done in the **Classification**→**Validate** menu. If the classification is valid it should be exported to XML, with the **File**→**Export Classification to XML** menu. After the classification has been exported to an XML file, the XML file can be uploaded to the server.

The upload process is shown in figure I.9 on page 160.

All server interaction is done in the **Server** menu and in order to upload the classification, the **Upload Classification** menu is selected. The XML classification is located on the hard drive and a “Collection Name” is generated automatically. Upon hitting upload the classification is sent to the WH System. When all the changes to the classifications in the World Heritage System are performed, the WH System needs to be told that it should show the result of the new changes. This is done by selecting **Server**→**Effectuate Changes**.

Finally the result of our efforts can be seen on the server. The “Example Classification” shows up among the other classifications in the “Advanced Search” page

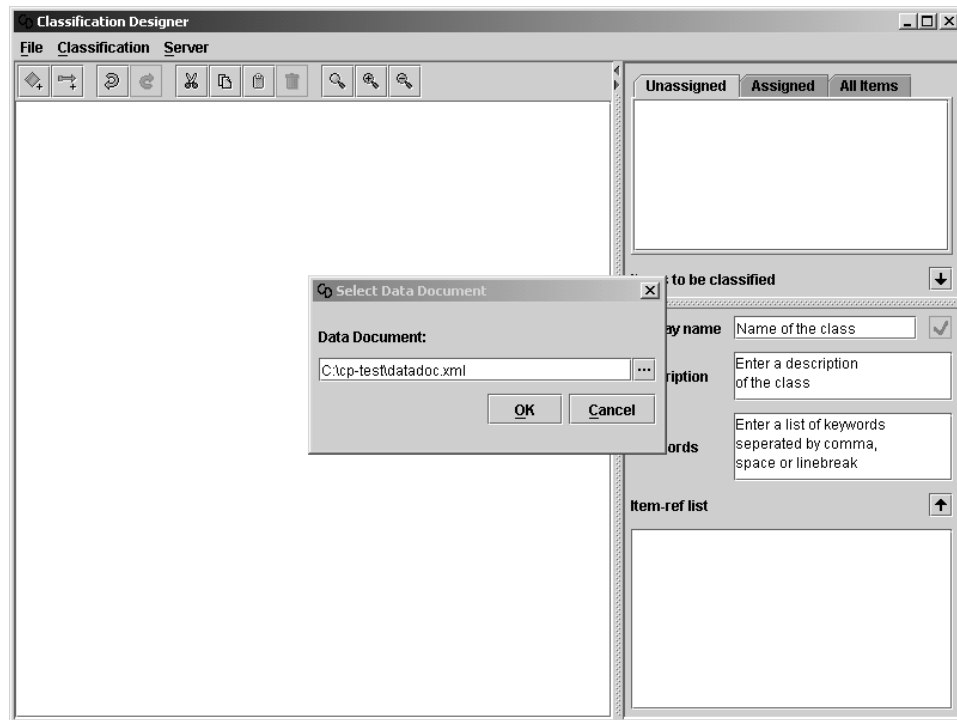


Figure I.5: Specify a Data Document.

in the WH System. A screen shot is shown in figure [I.10](#) on page [161](#).

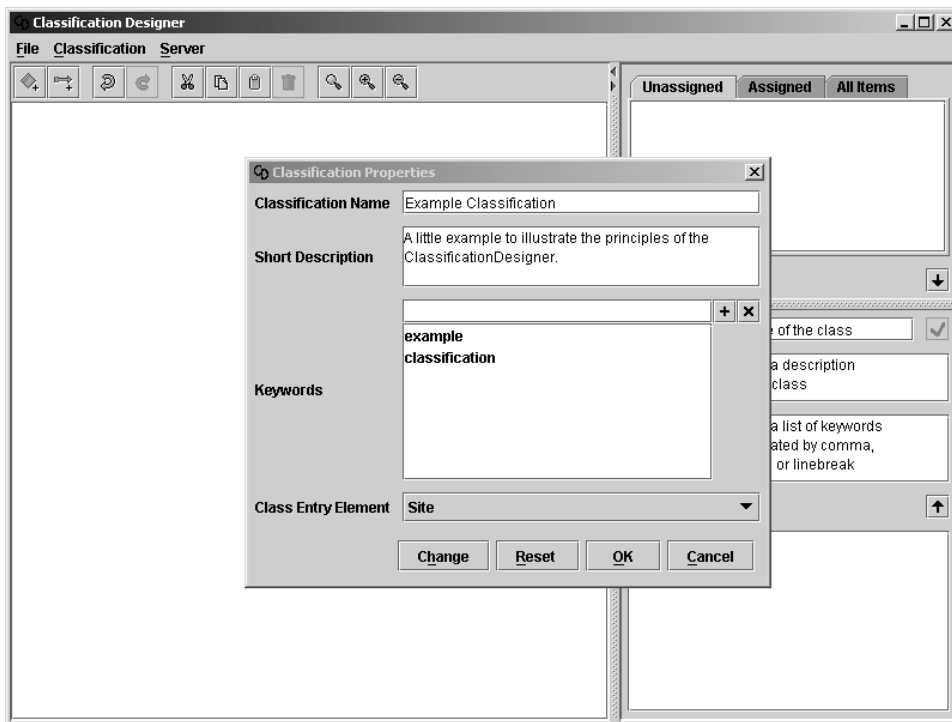


Figure I.6: Specify informations about the Classification.

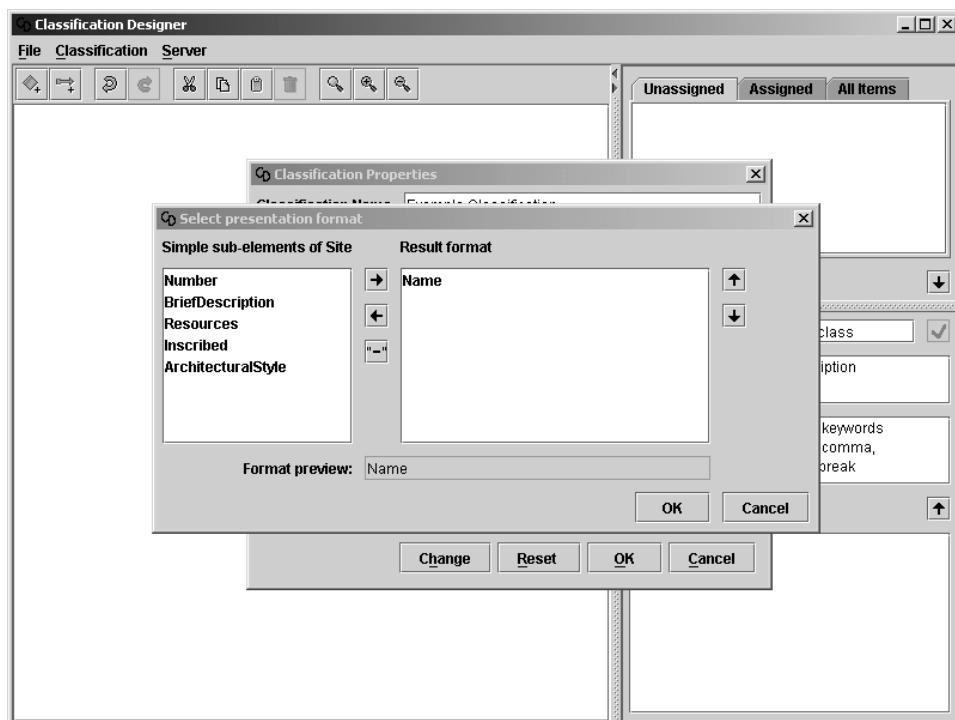


Figure I.7: Specify presentation format for the item references.

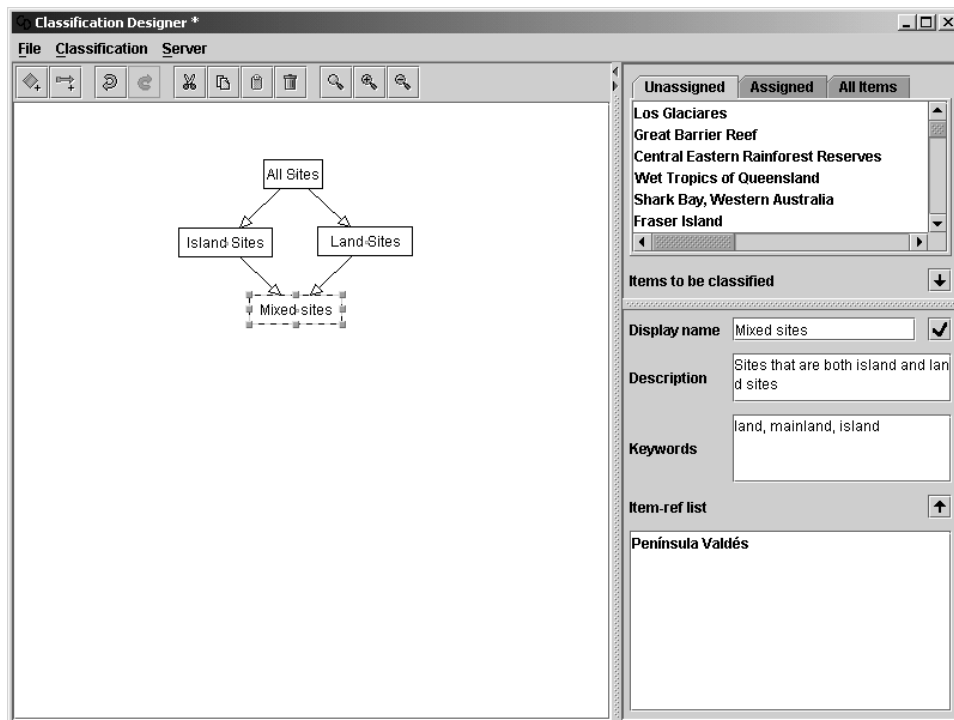


Figure I.8: The example Classification.

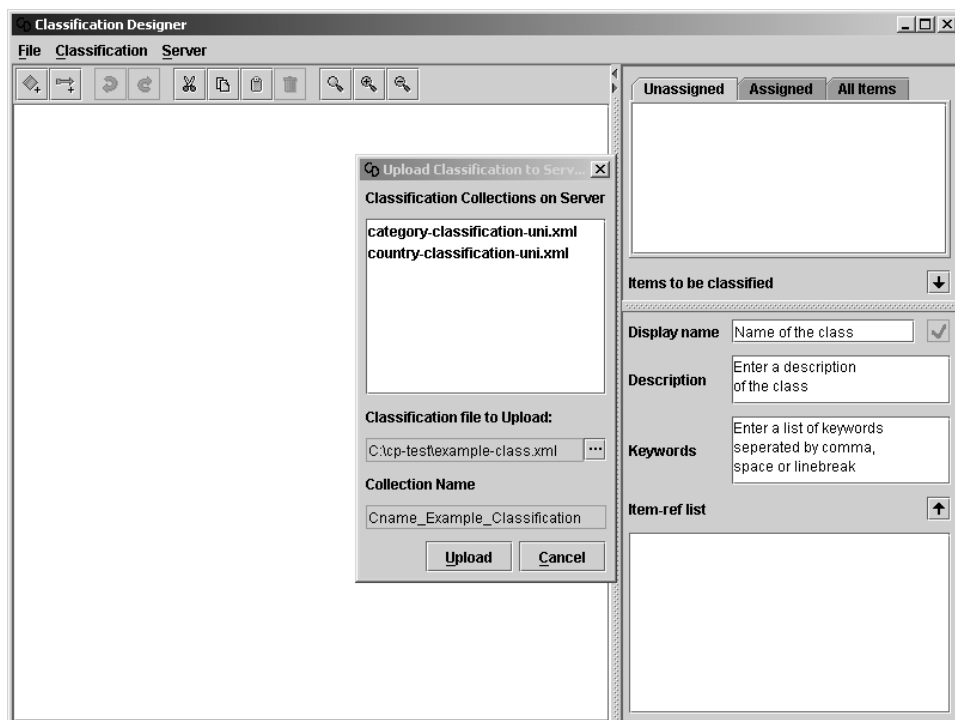


Figure I.9: Uploading the example Classification.

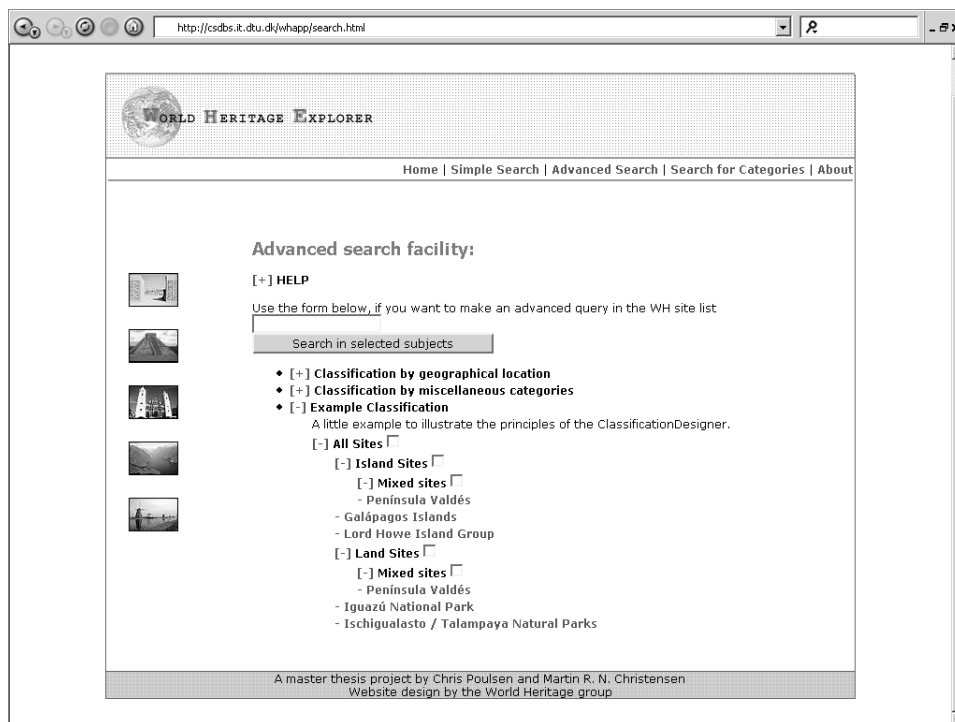


Figure I.10: WH System screched showing the example Classification.





## Appendix J

# XML XQuery Document

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <XQueries>
3   <XQuery>
4     <Name>getclassifications</Name>
5     <Data><![CDATA[
6       doc("Classifications:Classification")/Result/Classification
7     ]]></Data>
8   </XQuery>
9   <XQuery>
10    <Name>getstylesheet</Name>
11    <Data><![CDATA[
12      define variable $stylesheet external
13      doc($stylesheet)/Result/Stylesheet/Data
14    ]]></Data>
15  </XQuery>
16  <XQuery>
17    <Name>getsites</Name>
18    <Data><![CDATA[
19      define variable $sites external
20      doc($sites)/Result/Site
21    ]]></Data>
22  </XQuery>
23  <XQuery>
24    <Name>simplesearch</Name>
25    <Data><![CDATA[
26      define variable $sites external
27      doc($sites)/Result
28    ]]></Data>
29  </XQuery>
30  <XQuery>
31    <Name>simplesearch</Name>
32    <Data><![CDATA[
33      define variable $sites external
34      doc($sites)/Result
35    ]]></Data>
36  </XQuery>
```

```

37 <XQuery>
38   <Name>evalxpathexpr</Name>
39   <Data><![CDATA[
40     define variable $xpath external
41     doc($xpath)/Result
42
43   ]]></Data>
44 </XQuery>
45
46 <XQuery>
47   <Name>makedataguide</Name>
48   <Data><![CDATA[
49 declare namespace wh="WorldHeritage"
50
51 define variable $classification as node() external
52
53 (:Converts a classification into a dataguide (HTML representation):
54   )
54 define function wh:writeDataguide() {
55   <DIV>
56   <A
57     href="{concat("javascript:Toggle('",$classification/@id,")"}"
58     ID="{concat("x",$classification/@id)}"
59     style="text-decoration: none"
60   >
61   [+
62   </A>
63   <B>
64   {$classification/DisplayName/text()}
65   </B>
66   <BR/>
67   <DIV
68   ID="{concat($classification/@id)}"
69   style="margin-left: 2em;display: none"
70   >
71   {
72     let $description:=$classification/Description/text()
73     return
74     if(string-length($description)>0) then (
75       $description,
76       <BR/>
77     )
78     else()
79   }
80   {
81     for $class in $classification/Class
82     return wh:writeClass($class,$classification/@id,$
83       classification/@id)
84   }
84   </DIV>
85   </DIV>
86 }
87

```

```

88 (:Helper function for writeDataguide() :)
89 define function wh:writeClass($class,$dgid as xs:string,$idprefix
    as xs:string) {
90   if (exists($class/Class) or exists($class/ItemRef))
91     then (
92       <A
93         href="{concat("javascript:Toggle('",$idprefix,$class/@id,"'
94           ")}"
95         ID="{concat("x",$idprefix,$class/@id)}"
96         style="text-decoration: none"
97       >
98       [+
99       </A>,
100      <B>{$class/DisplayName/text()}</B>,
101      <INPUT
102        name="{concat($dgid,"checker")}"
103        type="checkbox"
104        value="{ $class/@id }"
105      />
106    )
107    else (
108      "[-] "
109      ,
110      <B>{$class/DisplayName/text()}</B>
111    ),
112    <BR/>,
113    if(exists($class/Class) or exists($class/ItemRef)) then(
114      <DIV
115        ID="{concat($idprefix,$class/@id)}"
116        style="margin-left: 0em;display: none"
117      >
118        {
119          for $subclass in $class/Class
120          return
121          if(exists($subclass/@id)) then (
122            <DIV style="margin-left: 2em">
123              {wh:writeClass($subclass,$dgid,$idprefix)}
124            </DIV>
125          )
126          else (
127            let $idref := $subclass/@idref
128            let $referredclass := $classification//Class[@id=$idref]
129            return
130            <DIV style="margin-left: 2em">
131              {wh:writeClass($referredclass,$dgid,concat("REF",$idprefix
132                ))}
133            </DIV>
134          )
135        }
136      for $itemref in $class/ItemRef
137      return wh:writeItemRef($itemref)
138    )
139  }
140 }

```

```

139     ) else ()
140
141   }
142
143   (: Helper function for writeClass() :)
144   define function wh:writeItemRef($itemref) {
145     <DIV style="margin-left: 0em">
146       <A
147         href="{concat("siteinfo.html?siteid=",string($itemref/
148           ItemIdref))}"
149         >-
150         {string($itemref/DisplayName)}
151         {if(exists($itemref/HitPoints)) then(concat(" (", $itemref/
152           HitPoints/text()," hit")) else()}
153       </A>
154     </DIV>
155   }
156
157   wh:writeDataguide()
158 ]></Data>
159 </XQuery>
160
161 <XQuery>
162   <Name>findsimilar</Name>
163   <Data><![CDATA[
164     declare namespace wh="WorldHeritage"
165     define variable $ancestordepth as xs:integer external
166     define variable $site as xs:string external
167     define variable $classifications as node()+ {doc("Classifications:
168       /Classification")/Result/Classification}
169     (:
170     Finds categories which $site is part of an builds a dataguide with
171     these categories
172     :)
173     define function wh:findSimilar() {
174       (:****
175       For each classification, first find the parentclasses of the
176       site in question
177       ****:)
178       let $result :=
179         for $classification in $classifications
180         (: Try letting the database get the parentclasses at some
181         point.
182         let $cid := $classification/@id
183         let $parentclasses := doc(
184           concat("Classifications:/Classification[@id &= '", $cid, "'"]//
185             Class[ItemRef/ItemIdref &= "'", $site, "'"])
186         )/Result/Class
187       :)
188       let $parentclasses :=
189         for $subclass in $classification//Class

```

```

187     where $subclass/ItemRef/ItemIdref/text() = $site return $
        subclass
188     (:****
189     If the ancestordepth>1
190     then return wh:findParentClasses($ancestordepth,parentclasses)
191     else return the parentclasses
192     ****:)
193     return
194     if($ancestordepth>1) then(
195         wh:addPrefixToClassIds($classification/@id,
196             wh:findParentClasses($ancestordepth - 1,$parentclasses))
197     )
198     else (
199         wh:addPrefixToClassIds($classification/@id,$parentclasses)
200     )
201     (:**** Present the result as a classification *****)
202     return
203     <Classification id="advancedsearch">
204         <DisplayName>Sites in same categories as chosen site</
205             DisplayName>
206         {$result}
207     </Classification>
208 }
209 (:
210 Finds the ancestors of the classes $classes.
211 If $depth=1 then it finds the parents
212 If $depth=2 then it finds the grandparents
213 etc.
214 :)
215 define function wh:findParentClasses($depth as xs:integer,$classes
    as node()*) {
216     (:****
217     for each class in classes find the parentnode of the class
218     if the parentnode is of type "<Class>"
219     then add it to a list of parentclasses
220     else the parent node must be of type "<Classification>" so
221     add the original class to the list of parentclasses
222     ****:)
223     let $parentclasses :=
224         for $class in $classes
225         let $parentnode := $class/parent::*
226         return
227             if(name($parentnode)="Class")
228             then($parentnode)
229             else($class)
230     (:****
231     Make a list of unique parent classes: uniqueparentclasses
232     ****:)
233     let $uniqueparentids := distinct-values($parentclasses/@id)
234     let $uniqueparentclasses :=
235         for $uniqueparentid in $uniqueparentids return ($parentclasses
            [@id=$uniqueparentid])[1]

```

```

236 (:****
237 If $depth>1 then return wh:findParentClasses($depth-1,
      uniqueparentclasses)
238 else we shouldn't go any further up - return uniqueparentclasses
239 ****:)
240 return
241   if($depth>1)
242   then(wh:findParentClasses($depth - 1,$uniqueparentclasses))
243   else($uniqueparentclasses)
244 }
245
246 (:
247 Add prefixes to Class/@id in the list of classes: $classes
248 This is done to ensure unique ids of all classes.
249 :)
250 define function wh:addPrefixToClassIds($prefix as xs:string, $
      classes) {
251   for $class in $classes
252   return
253     if(exists($class/@id)) then (
254       <Class id="{concat($prefix,$class/@id)}">
255         {$class/*[name(.)!="Class"]}
256         {wh:addPrefixToClassIds($prefix,$class/Class)}
257       </Class>
258     )
259     else (
260       <Class idref="{concat($prefix,$class/@idref)}"/>
261     )
262 }
263
264 wh:findSimilar()
265
266 ]]></Data>
267 </XQuery>
268
269 <XQuery>
270   <Name>advancedsearch</Name>
271   <Data><![CDATA[
272
273 declare namespace wh="WorldHeritage"
274 declare namespace exist="http://exist.sourceforge.net/NS/exist"
275
276 (:
277 The searchdata has the following format:
278 <SearchData>
279   <Entry>
280     <Classification>classificationid</Classification>
281     <Category>cat1</Category>
282     <Category>cat2</Category>
283     ...
284   </Entry>
285   <Entry>
286     ...
287   </Entry>

```

```

288     ...
289     <Keyword>kw1</Keyword>
290     <Keyword>kw2</Keyword>
291     ...
292 </SearchData>
293 :)
294 define variable $searchdata as node() external
295
296 define variable $classifications as node()+ external
297
298 (:
299 Creates a condition expression based on the $idlist argument. It
    should initially be called with
300 1 as $counter argument. The result is on the form: [@id='1' or @id
    ='2" or ... or @id='87']
301 :)
302 define function wh:prepareIds($idlist,$counter as xs:integer) {
303     if($counter=1) then(
304         concat("@id &= '",string($idlist[1]),"' ",wh:prepareIds($
            idlist,$counter+1))
305     )
306     else (
307         if($counter=count($idlist))
308         then (concat("or @id &= '",string($idlist[$counter]),"' ")
309         else (concat("or @id &= '",string($idlist[$counter]),"' ",
            wh:prepareIds($idlist,$counter+1)))
310     )
311 }
312
313 (: Returns true when $elem is part of the collection $elemlist.
    False otherwise :)
314 define function wh:isIn($elem,$elemlist) {
315     exists(
316         for $subelem in $elemlist
317         where $elem=$subelem return 1
318     )
319 }
320
321 (:
322 Prepare a list of keywords for use in the eXist function match-any
323 :)
324 define function wh:prepareKwList($list,$index as xs:integer) {
325     if(count($list)=$index) then(
326         concat("'",string($list[$index]),".'")
327     ) else (
328         concat("'",string($list[$index]),".'",wh:prepareKwList($list
            ,$index+1))
329     )
330 }
331
332 (:
333 Perform an advanced search using the search data in the global
    external
334 parameter $searchdata.

```

```

335 Only searches the classifications in global external parameter $
      classification
336 :)
337 define function wh:advancedSearch() {
338   (:****
339   -- Get all relevant siteids based on the searchdata
340   -- For each keyword, search the database for relevant sites that
      matches the keyword.
341   Construct a temporary result on the form:
342   <KeywordSearch>
343     <Keyword>...</Keyword>
344     <Site>...</Site>
345     <Site>...</Site>
346   </KeywordSearch>
347   -- Create a collection of unique siteids
348   -- Create a filtered dataguide result
349   ****:)
350   let $classcollection :=
351     if(not(exists($searchdata/Entry))) then (
352       for $classification in $classifications
353       return
354         <ClassificationEntry>
355           <Classification>{$classification/@id}</Classification>
356           {$classification/Class}
357         </ClassificationEntry>
358     )
359   else (
360     for $entry in $searchdata/Entry
361     let $cid := $entry/Classification/text()
362     let $chosencategories := distinct-values($entry/Category)
363     (:let $classcondition := wh:prepareIds($chosencategories
      ,1):)
364     (:return doc(concat("Classifications:Classification[@id='
      ",$cid,"']//Class[",$classcondition,""])/Result/
      Class:))
365     return
366       <ClassificationEntry>
367         {$entry/Classification}
368         {$classifications[@id=$cid]//Class[wh:isIn(string(@id)
      , $chosencategories)]}
369       </ClassificationEntry>
370   )
371   let $siteidcollection := distinct-values($classcollection//
      ItemRef/ItemIdref)
372   let $sitecondition := wh:prepareIds($siteidcollection,1)
373   let $kwlist := wh:prepareKwList($searchdata/Keyword,1)
374   let $matchingsites :=
375     doc(concat("DataCollection:/Sites/Site[match-any(.,", $kwlist, "
      )]"))/Result/Site
376   let $filteredsites := $matchingsites[wh:isIn(@id,$
      siteidcollection)]
377   let $uniquesiteids := $filteredsites/@id
378   let $resultClassification :=
379     <Classification id="advancedsearch">

```



```

380     <DisplayName>Search results</DisplayName>
381     {
382     for $classification in $classifications
383     return
384         for $class in $classification/Class
385         let $topclasses := $classcollection[Classification=$
            classification/@id]/Class
386         let $allclasses := distinct-values(($topclasses,$
            topclasses//Class)/@id)
387         return
388         wh:filterClass(
389             $class,
390             $classification,
391             $uniquesiteids,
392             $classification/@id,
393             $allclasses
394         )
395     }
396 </Classification>
397
398 (:****
399 Now create the result as a simple list with hitpoint count in
    the form:
400 <ResultList>
401     <HitCount></HitCount>
402     <ItemRef>
403         <DisplayName>...</DisplayName>
404         <ItemIdref>...</ItemIdref>
405         <MatchingCats>...</MatchingCats>
406         <MatchingKW>...</MatchingKW>
407         <HitPoints>...</HitPoints>
408     </ItemRef>
409     <ItemRef>
410         ...
411     </ItemRef>
412     ...
413 </ResultList>
414 ****:)
415 let $nestedClasscollection := <Classcollection>{$classcollection
    }</Classcollection>
416 let $simpleResultlist :=
417 <ResultList>
418     <HitCount>{count($uniquesiteids)}</HitCount>
419     {
420     for $siteid in $uniquesiteids
421         let $site := ($filteredbsites[@id=$siteid])[1]
422         let $existmatches := $site//exist:match
423         let $kwds := distinct-values(for $s in $existmatches return
            lower-case($s/text()))
424         let $matchingCats := count($nestedClasscollection//Class[
            ItemRef/ItemIdref=$siteid])
425         let $matchingKW := count($kwds)
426         return
427         <ItemRef>

```

```

428     <DisplayName>{string($site/Name)}</DisplayName>
429     <ItemIdref>{string($siteid)}</ItemIdref>
430     <MatchingCats>{$matchingCats}</MatchingCats>
431     <MatchingKW>{$matchingKW}</MatchingKW>
432     <HitPoints>{$matchingCats + $matchingKW - 1}</HitPoints>
433     {for $kwd in $kwds return <Keyword>{$kwd}</Keyword>}
434   </ItemRef>
435 }
436 </ResultList>
437 return ($resultClassification,$simpleResultlist)
438 }
439
440 (:
441 Helper function for advancedSearch()
442 Remove all subclasses of $class,
443 which does not have an id $classidcollection and
444 which does not contain or has a subclass that contains
445 a site with an id in $siteidcollection
446 :)
447 define function wh:filterClass($class as node(),$classification as
    node(),$siteidcollection,$prefix,$classidcollection) {
448   (:Get the type of Class - id or idref :)
449   let $type := if(exists($class/@id)) then("id") else("idref")
450   (:In case of a <Class idref=".." /> element, get the "real" <
    Class> element:)
451   let $thisclass :=
452     if($type="id") then(
453       $class
454     )
455     else (
456       let $idref := $class/@idref
457       return $classification//Class[@id=$idref]
458     )
459   let $subclasses := $thisclass/Class
460   let $filteredSubclasses :=
461     for $subclass in $subclasses return wh:filterClass($subclass,$
    classification,$siteidcollection,$prefix,$
    classidcollection)
462   let $itemrefs := $thisclass/ItemRef[wh:isIn(ItemIdref,$
    siteidcollection)]
463   return (
464     if(
465       (empty($itemrefs) and empty($filteredSubclasses))
466       or
467       (not(wh:isIn($thisclass/@id,$classidcollection)) and empty($
    filteredSubclasses))
468     )
469     then()
470     else(
471       if($type="idref") then(<Class idref="{concat($prefix,$
    thisclass/@id)}"/>)
472       else (
473         <Class id="{concat($prefix,$thisclass/@id)}">
474           {$thisclass/DisplayName}

```

```

475         {$thisclass/Description}
476         {$thisclass/KeyWords}
477         {$itemrefs}
478         {$filteredSubclasses}
479     </Class>
480 )
481 )
482 )
483 }
484
485 wh:advancedSearch()
486
487 ]]></Data>
488 </XQuery>
489 <XQuery>
490     <Name>searchclassifications</Name>
491     <Data><![CDATA[
492
493 declare namespace wh="WorldHeritage"
494
495 define variable $classifications as node()+ external
496 define variable $keywords as node()+ external
497
498 (:
499 Prepare a list of keywords for use in the eXist function match-any
500 :)
501 define function wh:prepareKwList($list,$index as xs:integer) {
502     if(count($list)=$index) then(
503         concat("'",string($list[$index]),"'")
504     ) else (
505         concat("'",string($list[$index]),"',",wh:prepareKwList($list,$
506             index+1))
507     )
508 }
509 (:
510 Adds $prefix to the id in $class and all subclasses to $class.
511 Referred classes (<Class idref=../>) are replaced by the
512     referred class
513 :)
514 define function wh:fixClass($class as node(),$prefix as xs:string)
515     {
516     let $thisclass :=
517         if(exists($class/@id)) then (
518             $class
519         )
520         else (
521             ($classifications//Class[@id=$class/@idref])[1]
522         )
523     return
524     <Class id="{concat($prefix,$thisclass/@id)}">
525         {$thisclass/*[name()!="Class"]}
526         {for $subclass in $thisclass/Class return wh:fixClass($
527             subclass,concat("D",$prefix))}

```

```

525     </Class>
526 }
527
528 (:
529 Search $classifications for categories containing words in $
      keywords.
530 :)
531 define function wh:searchClassifications() {
532 <Classification id="advancedsearch">
533   <DisplayName>The following categories matched your keywords</
      DisplayName>
534   {
535     let $kwlist := wh:prepareKwList($keywords/text(),1)
536     for $class in doc(concat(
537       "Classifications:/Classification//Class[match-any(
          DisplayName," ,
538       $kwlist,
539       ") or match-any(Description," ,
540       $kwlist,
541       ") or match-any(Keywords," ,
542       $kwlist,
543       ")]"
544     ))/Result/Class
545     return wh:fixClass(
546       <Class id="{concat("X",$class/@id)}">
547         <DisplayName>{string($class/DisplayName)}</DisplayName>
548         <Description>{string($class/Description)}</Description>
549         <Keywords>{string($class/Keywords)}</Keywords>
550         {$class/Class}
551         {$class/ItemRef}
552       </Class>,"")
553     }
554   </Classification>
555 }
556
557 wh:searchClassifications()
558
559   ]></Data>
560 </XQuery>
561 <XQuery>
562   <Name>updatecheck</Name>
563   <Data><![CDATA[
564
565 doc("Conf:/UpdateSerialNumber")/Result/UpdateSerialNumber
566
567   ]></Data>
568 </XQuery>
569
570 </XQueries>

```

# Glossary

**Class** A category in a classification.

**Classification** A categorization of some domain expressed in XML.

**CLI** Command Line Interface - An interface in a textual environment like a shell or a DOS prompt.

**Dataguide** Graphical representation of a classification.

**EJB** Enterprise JavaBean - A java component containing business logic e.g. for searching database backends for keywords.

**Entity Bean** A type of EJB that represents a relation in a relational database.

**ER Diagram** Entity Relation Diagram - A way of describing schemas in relational databases.

**GUI** Graphical User Interface - The fancy components with textfields, labels and so on, this is the opposite of a CLI.

**JavaBean** A Java class containing data that should be presented to a web client. JavaBeans are typically the only source of data available in JSP pages, when using the MVC design pattern. Do not confuse with *Enterprise JavaBean*.

**JNDI** Java Naming and Directory Interface - A service for looking up Java objects based on unique names for these objects – possibly across a network

**JSP** JavaServer Page - A JSP page is (mostly) responsible for presenting a page to a user in HTML. The HTML is often generated dynamically. JSP pages are quite similar to ASP or PHP pages.

**MVC** Model View Controller - A design pattern often used for Java and J2EE.

**Presentation Layer** The part of the application that creates a “window” to the underlying data. In this system the presentation layer is responsible for generating things like the data-guides and lists of search-results.

**RDBMS** Relational DataBase Management System - A traditional database, that can be queried using SQL.

**RMI** Remote Method Invocation - The Java answer to RPC.

**RPC** Remote Procedure Call - A way of calling procedures/methods on a remote machine without having to take the network into account.

**Session Bean** A type of EJB that contains business logic.

**SGML** Standard Generalized Markup Language is a standard for how to specify a document markup language or tag set.

**SQL** Structured Query Language - A standard interactive and programming lan-

guage for getting information from and updating relational databases.

**URI** Uniform Resource Identifier - A standard way (defined by W3C) of identifying resources on the Internet.

**XML** The eXtensible Markup Language