CMServer
An Object-Oriented Framework for Website Development and Content Management

Diploma Thesis

Michael Grossniklaus
< michael@vis.ethz.ch >

Prof. Dr. Moira C. Norrie
Supervisor: Beat Signer

Global Information Systems Group
Institute for Information Systems
Department of Computer Science

March 3rd 2001
Abstract

Providing content for heterogeneous platforms becomes more and more important. With the arrival of new mark-up languages such as CHTML or WML, separation of content and layout is a more critical property than ever for any system for website development. In this diploma thesis we present an object-oriented framework for multi-target website development and content management, implementing these key issues based on XML and XSLT.
# Contents

1 Introduction  .................................................................................. 1  
  1.1 Problems of Web Content Management ............................................ 1  
    1.1.1 Static Web Pages ........................................................................ 1  
    1.1.2 Mingling of Content and Layout .................................................. 1  
    1.1.3 Content Reusability ................................................................... 2  
    1.1.4 Querying and Processing of Content .......................................... 2  
  1.2 Requirements of Web Content Management ........................................ 2  
    1.2.1 Separation of Content, Structure and Layout ............................... 2  
    1.2.2 Abstraction of Objects ................................................................ 2  
    1.2.3 Multiple Presentation Channels .................................................. 3  
    1.2.4 Multilingual Content .................................................................. 3  
    1.2.5 Workflow and Object Lifecycles .................................................. 3  
    1.2.6 Object Validation ....................................................................... 3  
    1.2.7 Extensibility ............................................................................... 3  
  1.3 Existing Approaches ....................................................................... 4  
    1.3.1 Document Based Approach .......................................................... 4  
    1.3.2 Data Based Approach .................................................................. 5  
  1.4 A New Approach to Content Management .......................................... 6  

2 Data Storage ..................................................................................... 7  
  2.1 Web Data Types ............................................................................... 7  
    2.1.1 Text ............................................................................................ 7  
    2.1.2 Picture ....................................................................................... 7  
    2.1.3 Universal Resource Locator (URL) ................................................. 8  
    2.1.4 Link ............................................................................................ 8  
    2.1.5 Mixed Content ........................................................................... 8  
  2.2 Database Schema ........................................................................... 8  
    2.2.1 Components ............................................................................... 8  
    2.2.2 Contents ..................................................................................... 14  
    2.2.3 Languages .................................................................................. 17  
    2.2.4 Templates ................................................................................... 17  
    2.2.5 Workflow States ......................................................................... 18  
    2.2.6 Gatekeepers ............................................................................... 18  

3 Generation of Pages ......................................................................... 19  
  3.1 Generation of XML ......................................................................... 19  
    3.1.1 Description of the Process ............................................................ 19
### CONTENTS

3.1.2 A Sample Situation ............................................ 20
3.1.3 A Note on the Used DTD ...................................... 25

3.2 Generation of XSLT .................................................. 26
3.2.1 Description of the Process .................................... 26
3.2.2 A Sample Situation .............................................. 27

3.3 Application of XSLT to XML ...................................... 30
3.3.1 Description of the Process .................................... 30
3.3.2 A Sample Situation .............................................. 30

4 Implementation .......................................................... 33
4.1 Overview ................................................................... 33
4.2 Servlet .................................................................... 34
4.2.1 Page Generation .................................................. 34
4.2.2 Page Updating ..................................................... 34
4.2.3 Servlet Parameters .............................................. 35
4.3 Database Classes ..................................................... 36
4.3.1 Package Overview ............................................... 36
4.3.2 Detailed Implementation Notes ............................. 39
4.4 XSL Stylesheet .......................................................... 43
4.5 Utility Package .......................................................... 43
4.5.1 Package Overview ............................................... 43

5 Conclusions ................................................................. 47
5.1 Lessons Learned ....................................................... 47
5.1.1 Fulfilment of Requirements ................................. 47
5.1.2 Used Technologies .............................................. 48
5.1.3 Open Issues ........................................................ 48
5.2 Future Work ............................................................. 48
5.2.1 Open Issues ........................................................ 49
5.2.2 Further Enhancements ......................................... 49

A Server Setup ............................................................... 51
A.1 CMServer ................................................................ 51
A.2 httpd.conf ................................................................ 51
A.3 tomcat.conf ............................................................. 52
A.4 server_cmserver.xml ................................................. 52
A.5 Startup Apache and Tomcat ....................................... 53

B Server Configuration ..................................................... 55
B.1 Information About the Database ............................... 55
B.2 Information About the Server Configuration ............... 55
B.3 Information About the Website ................................. 56
B.4 Information About XSL Stylesheets ......................... 57

C Updating Pages ........................................................... 59
C.1 Update Templates ..................................................... 59
C.2 Generated Document ............................................... 63
Introduction

In the last few years the Internet has become a major source of information and advertising. Providing up-to-date content is a mission-critical issue. In this chapter we discuss the challenges of providing information on the web and establish a set of basic requirements for a content management system. Then we shall have a look at existing solutions and examine how they meet those requirements. Finally we introduce our approach of an object-oriented content management system based on XML and XSLT.

1.1 Problems of Web Content Management

When managing content on a website one faces a variety of problems. In this section we describe some of these problems in detail and show how they are being handled without content management.

1.1.1 Static Web Pages

A large number of web pages on today’s Internet are static. As a consequence the content is distributed among a potentially large number of documents and obfuscated by the mark-up. To facilitate generation and management of such pages various editors (e.g. Microsoft Frontpage) have been developed that empower the user to modify the documents visually and keep track of the structure and references of a website.

1.1.2 Mingling of Content and Layout

Web documents are a combination of content and the mark-up that describes its rendering (e.g. HTML). When attempting to change the layout of such a website, a web engineer faces the major task of modifying and updating all pages to the new design by hand. Migrating to a new target platform such as cellular phones or personal digital assistants (PDAs) requires to create a whole new set of documents with the content formatted in the appropriate mark-up.
1.2. REQUIREMENTS OF WEB CONTENT MANAGEMENT

1.1.3 Content Reusability

As there is no notion of content components in static web documents, code representing the same object has to be duplicated in all documents where the object should occur. For instance a link to another web page may be present on multiple pages. When the target of such a link is no longer valid or moves to another location the web developer maintaining the website will have to search through all files and update the appropriate URLs. This process is quite tiresome and very error prone as spelling errors or abandoned resources are hard to track.

1.1.4 Querying and Processing of Content

When working with mark-up documents, a lot of the information about the structure and the meaning of the components contained in a document is lost. Querying and processing such information is extremely tedious since this meta information will have to be reconstructed prior to the processing of the data. As an example of a situation where querying a website becomes important, one can think of a program checking if all links contained in a website are still valid. It is very inefficient and not particularly elegant if such an application would need to parse all web documents to extract the included URLs before being able to check them.

1.2 Requirements of Web Content Management

From the problems occurring with traditional websites discussed above, a list of requirements for a content management system (CMS) can be constructed. In this section we present some of the most commonly described properties of such systems and motivate their importance.

1.2.1 Separation of Content, Structure and Layout

A content management system should separate content from structure and presentation. Given that property websites become more easily manageable as the individual parts can be modified or changed independently from each other. Maintaining large web systems also implies that the responsibility of keeping content up-to-date and designing the look-and-feel of a site will fall into different hands. For example, one can imagine that there is a potentially large number of information providers in charge of the content and only a few web designers in charge of the layout. In that case this separation enforces the natural distribution of skills and supports the cooperation of these various groups.

1.2.2 Abstraction of Objects

A desirable property of a content management system is the notion of an object abstraction. When developing websites one usually reuses a given resource more than once. A good example for such a resource reuse would be a logo image appearing on virtually every page. If this logo image is not encapsulated as an object, the replacement of the resource becomes tedious as the reference to the image file is replicated on every page. On the other hand such a modification is very simple when working with objects as the reference is stored only at one place in the system. Modifying this object will cause all other objects referencing it to include the new resource.
1.2.3 Multiple Presentation Channels

With the continuing emergence of mobile platforms the set of available target mark-up languages is also growing. A good content management system will therefore provide an easy mean to support multiple presentation channels without modifying the structure or content stored in the system. Supporting a new platform in such a system should only require the specification of the presentation and perhaps the addition of some content in platform specific formats.

1.2.4 Multilingual Content

Organizations with customers around the world need to provide the information on their website in as many languages as possible. To provide an improvement over a traditional document based website a content management system has to support multiple languages elegantly. Again the notion of an object abstraction encapsulating the various content instances associated with the object becomes important.

1.2.5 Workflow and Object Lifecycles

When collaborating in the field of information publishing, people often establish workflow processes that define how an object is developed, validated and released. These processes can be modelled as a directed graph with the workflow states as its nodes and the possible state changes as directed edges linking one state to another. To support such workflow processes a content management system has to offer a possibility to define workflow states and transitions. Furthermore it must be possible to associate the content objects in the system to the defined states.

1.2.6 Object Validation

Another important property of content management is the notion of object validation. By object validation we mean the process checking whether a given component should appear on the final document. This can be decided on numerous criteria. Some of the more common include the workflow state of an object or a given expiry date. When supporting different user profiles, it is also possible to use this simple, yet powerful concept to generate personalized webpages.

1.2.7 Extensibility

Extensibility is a key property of any software system built today. As a consequence web content management systems have to satisfy this requirement too. When building a website, it should be possible to extend the system with user defined types representing semantically rich objects. In doing so, it is possible to construct complex components from the objects provided by the content management system thereby simplifying content management and providing a higher level of abstraction for the content engineer.
1.3 Existing Approaches

When looking at the numerous content management solutions available on the market today, one may find that they can be partitioned into two groups. The first group we discuss here, approaches the problem out of the perspective of a document and its objects. The second group takes the approach of bringing complex data objects to the web.

1.3.1 Document Based Approach

Most first generation content management systems use the document based approach to bring content to a web platform. This solution has developed out of the viewpoint of web designers and web masters. As these groups of professionals think of a website in terms of pages, texts, images, tables and links such a system will provide types and storage facilities for all these components. This concept is schematically shown in figure 1.1.

Figure 1.1: Document Based Approach

To compose pages from the components two methods can be observed in practice.

Templates This approach uses textual specifications (templates) stored in the database to build a page from the basic components. Such a template usually includes the references to the objects, the structure and the layout mark-up of the final page. Although this approach still provides a separation of content and layout, it does not enforce separation of content and structure. It is therefore the responsibility of the author of the templates to ensure that the structure of the data does not change from one target-platform to another. Despite its drawbacks this methods is relatively easy in its use for skilled web masters as the templates are usually very similar to an ordinary static web page. Migration to such a system is not a problem as the learning curve for the existing administrators is not very steep.

Scripting Another solution for page composition is the use of scripting and active pages. The layout and structure of a page have to be encoded using the concepts of an imperative programming language. While this concept provides by far the most flexible solution, a system built in that way is not easy to maintain as layout and structure are obfuscated by the statements of the scripting language. Note that again the structure is not separated from the design in this approach. Hence moving to other target platforms is a challenging task as a whole new set of scripts has to be developed. Migration to such a platform usually requires additional training for the developers in charge of the website, as they will have to acquire programming skills.

The main advantage of this solution is its acceptance among web designers and content publishers. The systems falling into this category usually preserve the concepts known to document editors and publishers. Depending on the actual implementation of the content management system there can be some new skills that have to be acquired but the overall mechanisms
stay the same. A major drawback of these solutions is the integration of existing data or the creation of higher level objects. To support this notion customized development has to be done and proprietary solutions cannot be avoided.

1.3.2 Data Based Approach

In contrast to the first approach, this solution is not mapping the elements of documents into a data storage facility, but uses arbitrary data objects and publishes them on the web. Whereas the first document based approach is inspired by the concepts of publishing and document objects, this second approach which we call the data based approach, evolved around databases and data management systems. Figure 1.2 illustrates this technique graphically.

![Figure 1.2: Data Based Approach](image)

As arbitrary databases rarely provide information about the structure or layout in which the data should be presented or interpreted, this solution needs to introduce other concepts to transform the stored data into a website. Again there are two possible methods to achieve this.

One-to-One Relationship The most popular method in bringing such data to a website is to establish a one-to-one relationship between a database object type and a page of a website. This approach is commonly used for web stores or library systems where a single database object (e.g. a book) is display as a web page. Use of query mechanisms and database views extend this solution to display multiple objects of the same type on a single page. The overall structure of the data and the design of the web page cannot be defined in the database and has to be specified by another mechanism in the system.

Scripting As before scripting is an option that can be used to build pages in this case too. Again scripting provides unlimited possibilities, but has also proven hard to maintain and administrate. Good examples for data based approaches using scripting techniques are portal sites that combine various database objects into a single page depending on the logged on user. These so called web applications involve a great amount of specialized development. Maintenance and evolution of the code base is a time consuming and complicated process as web scripting languages rarely provide for software design aspects.

Integration of existing and legacy data is with no doubt the main advantage of solutions built in this way. On the other hand it is never possible to bring the data to the web without a significant amount of new development. As databases and database models tend to be as different and heterogeneous as the people who create them, it is virtually impossible to find a solution that fits them all. Another disadvantage is the shift in the required knowledge to develop or administer such a website. Common web masters knowing about mark-up documents and perhaps scripting are no longer able to work with such a site as profound skills in database technologies and real programming must be present.
1.4 A New Approach to Content Management

Both approaches presented in the previous section have important and good concepts that should be integrated into any content management system. Whereas the first solution provides ample facilities for structuring documents and websites in an easy and intuitive way, the second solution allows unlimited possibilities of extending and customizing the system with user-defined data types.

On the other hand we have seen that both systems suffer from serious drawbacks and limitations. The first technique is very narrowly directed at the experienced web designer and does not cater for structured or higher level objects in any way. The second solution comes from the world of databases. Therefore focus is on the data and the relation between the different items. Presentation or layout is not integrated into the system and has to be specified elsewhere.

In this diploma thesis we want to show how the best aspects of both approaches could be integrated into one system without losing too much of the power of the individual solutions. Our system which we will present in the following chapters in more detail can be viewed as a middle course between the two types of systems discussed here. We strongly believe that any content management system today should provide basic web data types as found in classic web documents as well as facilities for complex user defined object types and extensibility. At the same time such a system will have to fulfil the basic requirements of a content management system defined earlier in this chapter as well.
To store the structure and the data of a website, our system uses the object-oriented Object Model System (OMS) \[2\] which has been developed at the Global Information Systems Group at ETH Zurich as a database. In this chapter we want to discuss the schema of the database and show how the metadata and the web components are managed.

### 2.1 Web Data Types

When designing a database to store information about the structure and content of a web page, it is useful to think about the basic data types of such documents first. Given these thoughts, it is easier to build a database schema to hold this information. The following sections present the set of data types that we have chosen in our implementation.

#### 2.1.1 Text

The most basic data type of any web page is pure text. Text stored in this format does not contain any formatting information such as boldface, italics, underline or even line breaks. It must however provide the possibility to store and manage multilingual content. This form of text is used, when the text does not need to contain other components such as images and is free of formatting. Examples might be the curriculum vitae or job title of a person.

#### 2.1.2 Picture

Pictures are an omnipresent form of content on the Internet. Reason enough to have a basic data type in our system dedicated to the storage and management of these resources. As images can contain text and are often used as banners or headings, this data type must also be capable of managing multilingual content. Further restrictions in the field of mobile computing even require a possibility to store an image in more than one format as for instance WAP phones are not yet capable of displaying colour images.
2.1.3 Universal Resource Locator (URL)

An important characteristic of the Internet is its use of hypertext. Any document can reference other documents thus enabling context driven navigation. Target locations are usually specified by universal resource locators (URLs). To manage these references it is useful to store them separate from other objects. This separation from links for instance ensures that the URL of a resource is stored in the system only once, thereby facilitating maintenance of the website when such a resource locator changes or the referenced resource is moved to another point in the Internet.

2.1.4 Link

The basic data type of a link is closely coupled with the one of the universal resource locator described in the section before. An object of this type uses an arbitrary component as resource and puts it together with an URL object to form a link as known from the Internet. Several links may link the same URL to different web components, again ensuring that a single URL is stored only once in the system. As a link is a pure structural concept there is no need to incorporate any multilingual concepts into this data type.

2.1.5 Mixed Content

With the data types introduced so far, it is possible to build web pages. But sometimes it is also useful to mix multiple components into one object. For instance it would not be possible to inline pictures or links into a text with the concepts discussed so far. To solve this problem, we introduce the data type mixed content to allow such combination in the form of references to other objects in the database as mark-up in the text. Of course, this type has to provide multilingual concepts for itself as well as for the components incorporated within.

2.2 Database Schema

After having discussed the basic data types a web content management system should offer, in the next few sections we show how these concepts are realized. As we will use the OMS Java data management framework [3] to implement our system, we will model our database using the OM Model. The complete schema is shown in figure 2.1.

2.2.1 Components

The central part of the schema are the component objects. These objects represent all components that can be included and rendered on a web page. In the following sections these object types shall be discussed in turn, by describing the motivation for such an object, giving its specification in the database and explaining how these objects are supposed to work depending on the respective attribute values.

CMComponents

The type cmcomponent is the common super type of all objects that can be displayed on a web page. All components representing the basic web data types discussed in the previous
CHAPTER 2. DATA STORAGE

Figure 2.1: OM Database Schema
section are extensions of this type. Note that the default collection CMComponents can contain no instances having only this type, since all sub collections have been defined with a partition constraint.

As this type is the heart and soul of our content management system, we will try to describe it further by giving the exact specification as declared in the CMServer schema file.

```plaintext
type cmcomponent
   ( name : string;
   description : string;
   );
```

There are two attributes name and description for this type. The name of a component is used to provide some information about the object. It can be though of as an identifier, which should be unique, although this constraint is never enforced by the system. The name of a component can for instance be used to reference it from within a mixed content context. The second attribute provides the facility to store a description for this object. The description is a simple string containing information about the function or the content of the object. It is provided to support the user in the management of his objects and has no direct impact on the content management system. It has nevertheless been included to support future development of user-interface programs.

**CMContainers**

![Diagram](https://via.placeholder.com/150)

Figure 2.2: Building a Hierarchic Tree Structure

The next very important component of the database schema are containers represented by collection CMContainers and type cmcontainer respectively. The concept of a container is used to build hierarchic object trees and thus provide the facility to reuse certain collections of objects in more than one place. Figure 2.2 illustrates this concept.

In our database schema we employed the composite pattern as described in [1] and shown in figure 2.3. This pattern introduces a subtype container of the abstract type component to aggregate a set of further components or containers. In our implementation, type cmcomponent takes the role of the component and cmcontainer is the derived subtype representing the container object. To aggregate a set of components into one container we use the association hasComponents with domain cmcontainer and range cmcomponent.

The following excerpt from the schema definition file shows how the type cmcontainer is implemented in our system. Note that it only adds two attributes to the existing attributes of type cmcomponent.
Figure 2.3: The Composite Pattern

```
attribute
order
is a ranking of components. This concept provides a facility to give a certain
order to the objects aggregated in the container. When no order is specified, the objects will be
rendered on the website in the sequence in which the database returns them. It is also possible
to specify only partial orders. In that case the objects contained in the order attribute will be
rendered first, followed by the unordered rest of remaining objects. The second attribute
named
function
is a string that can be freely used to denote the function of this container
in some way or another. Common used values would be page, collection or mixed. This rather
inelegant convention had to be introduced into the system to be able to tell objects contained
in the sub collections of CMContainers apart.

CMTexts

Type
cmtext
is an extension of type cmcomponent and represents the data type for simple
unformatted text. In the composite pattern, it takes the role of a leaf node of the hierarchical
structure of a page. As this type adds no further attributes to its super type, an excerpt of the
schema file is omitted. It may seem strange to declare a subtype that does not change or add
anything to its supertype. This has been done to allow mapping to a different Java class in
the case of objects of type cmtext than objects of type cmcomponent. In that way the
actual Java implementation is nicer in terms of object-orientation, as the code in the class
representing type cmcomponent does not to know anything about the behaviour of objects
of type cmtext.

CMPictures

To represent pictures in the system, we have introduced type cmpicture. Similar to type
cmtext, this type is also an extension of type cmcomponent and can therefore take the role
of a leaf in the tree representing the hierarchy of the structure of the web page. In contrast
to type cmtext, type cmpicture not only subtypes its super type but also adds further
attributes as shown in the sample of the schema file below.
There are three additional attributes in type cmpicture. Attribute border can be used, to specify the width of the border, that should be rendered around the image. The second and the third attribute, width and border respectively, specify the preferred width and height of the image. Although the actual decision if this information is used on the targeted user-agent lies within the associated template, we have nevertheless decided to include them in this object to provide an easy way to influence the behaviour of the templates.

CMUrls

Type cmurl represents an universal resource locator (URL) in the database. As the types before this is also an extension of type cmcomponent and can theoretically be included in any container. In practice however objects of type cmurl rarely function as individual components but are rather used as parts of link objects discussed below.

```
type cmurl subtype of cmcomponent
(  reference : string;
  internal  : boolean;
  valid     : boolean;
);
```

Three further attributes are added by type cmurl to its super type cmcomponent. The first attribute reference is a value of type string containing the actual URL. This reference can either be stored absolute (e.g. http://www.globis.ethz.ch) or relative to the server's address (e.g. pictures/moira.jpg). Attribute internal then gives information about this mode of specification. It is true, if the URL is of the second form and false otherwise. The last attribute (valid) can be used by programs checking the validity of URLs within a website (so-called URL-Checkers). If such a program would find an invalid link it could set the attribute to the value false, thereby indicating to the server engine, that this reference has to be treated specially or needs to be corrected. How this situation is handled in detail has to be determined by the preferences of the owner of the website.

CMLinks

A type closely coupled with type cmurl is type cmlink. This type is used to represent links in our system. It combines an arbitrary component of the system together with a universal resource locator. Type cmlink too is an extension of type cmcomponent and its instances can thus be used anywhere on the website. The following sample from the database schema file shows the exact definition of type cmlink.

```
type cmlink subtype of cmcomponent
(  resource  : cmcomponent;
  target     : cmurl;
);
```

The first property added by type cmlink is the attribute resource. The value of this field stores the component the link should be attached to. Although in theory one can use any kind of object having type cmcomponent as resource of a link, very few types are actually sensible. Among them are for instance types like cmtext, cmpicture or cmdirectory. The second attribute named target stores a reference to the URL of the target site of this link object.
CHAPTER 2. DATA STORAGE

**CMDirectory**

A rather odd feature in the inheritance hierarchy rooted at type `cmcomponent` is the inclusion of type `cmdirectory`. Objects of this type are used to represent directories or folders on the web server. This is useful to be able to create a more complex structure and provide a possibility of grouping similar pages together. Despite this role as a management component, objects of type `cmdirectory` must also be able to be included on web pages. This may seem strange at first, but is completely sensible when one thinks of objects like navigations or sitemaps. These components use the directory hierarchy and the directories themselves, to display an overview of the structure of the website, thereby providing the user with an easy way of access the information on the website.

```plaintext
type cmdirectory subtype of cmcomponent
    ( root : boolean;
    );
```

As can be derived from the above specification of type `cmdirectory`, taken from the database schema file, there is only one additional attribute. The boolean value stored in the attribute `root` is true if the directory is the root directory of the web server’s directory structure and false otherwise.

**CMPages**

There is no special type to represent a page of the website contained in the database. However, there is a special collection with member type `cmcontainer` to store all available pages. As all objects of type `cmcontainer` map to the same Java class, there is no way for instance to distinguish a collection from a page. To solve this problem it is advisable to set the attribute function of page objects to the string `page`. Doing so enables selection of the appropriate template during generation of the target mark-up.

**CMCollections**

Similar to the collection `CMPages` discussed in the section above, `CMCollection` has no own member type but uses type `cmcontainer` for its elements. As the `partition` constraint over the sub collections of `CMContainers` requires that there can be no instances in `CMContainers` that are not included in one of its sub collections, this collection is just a mean to ensure this constraint without classifying this objects as pages or mixed contents. Collections are used to group objects that are used together on multiple pages. An example of such a collection can be seen in figure 2.2.

**CMMixeds**

Objects of type `cmmixed` are used to represent the basic data type mixed content. This data type allows to combine text and references to other objects that should be inlined into the text. We have therefore decided to represent this data type as a container, containing the components referenced from the text. Beside these special semantics, this type does not define any additional attributes. A separate type has nevertheless been created as the rendering of objects of this type is significantly different from that of an ordinary container.
CMNaviation

One substantial advantage of content management systems is the possibility to generate certain objects automatically. In our system there are two types of such objects. The first is cmnavigation which is used to represent the website’s navigation. A navigation is a collection of links to the major parts of a website represented by the directory structure. It does however not contain any references to the pages contained within the directories. This component is purely calculated and contains no data of its own. As the following specification of type shows, there is only one attribute that allows configuration of objects of this type.

```plaintext
type cmnavigation subtype of cmcomponent
  ( context : boolean;
  );
```

The attribute context can be used to specify whether the navigation should behave context sensitive or not. When the navigation mode is set to context sensitive, the structure of the navigation will change according to the current page. In this mode it will include the subdirectories of the current directory, the super directory and the root directory of the website. Normally only the top directories, i.e. the subdirectories of the root directory and the root directory itself are visible in the navigation.

CMSitemap

An object type very similar to the one of cmnavigation is the second calculated object type, named cmsitemap. Objects of this type are used to display a complete overview over the whole website. This overview includes the entire directory structure. Optionally all objects of this type can be configured to include the individual pages as well. The following excerpt of the database schema file shows the exact specification of this type.

```plaintext
type cmsitemap subtype of cmcomponent
  ( includepages : boolean;
  );
```

There is only one additional attribute in type cmsitemap in comparison to its super type cmcomponent. The attribute includepages can be set to true or false, if pages should be included or omitted respectively.

2.2.2 Contents

To be able to handle content in multiple content languages and formats, the actual representation of this data had to be separated from the objects discussed in the previous section. In this respect the components presented above act as proxy objects merely representing the concept of an item of a website, whereas the components described in this section represent the actual data. The relationship between an component and its content is established by various associations.

CMContents

Similar to the hierarchy of components there is a common super type for all content objects. Type cmcontent defines the attributes common to all content objects. As there is a partition
constraint on the sub collections of collection CMContents there can be no instances in this collection that are not contained in any of its sub collections. This design approach is closely related to the concept of an abstract class in Java.

```
type cmcontent
 ( mimetype   : string;
 );
```

Type cmcontent defines only the attribute mimetype. This string value is used to hold information about the mime type of the content object. This can be used to select the appropriate content type for a given target platform. For instance it is not possible to display images having mime type image/jpg or image/gif on mobile platforms such as WAP phones. It is also possible to extend the semantic of a mime type to allow distinction of various content formats such as full-length texts in contrast to a summarized version.

**CMTitleContents**

Objects of type cmtitlecontent are used to store a language or format dependent title of a component. These contents can be associated with any component in the system. Hence it is possible to use this concept very flexible for many demands. For instance, title contents are used to give a heading to a text or to provide a link text for a directory.

```
type cmtitlecontent subtype of cmcontent
 ( title       : string;
 );
```

Only one additional attribute is defined in type cmtitlecontent in comparison to its super type. This attribute title is used to store the title of the associated component as a string value.

**CMTextContents**

Type cmtextcontent represents the content objects associated with components of type cmtext. These are very simple objects, as the following excerpt from the schema file shows.

```
type cmtextcontent subtype of cmcontent
 ( content     : string;
 );
```

There is only one more attribute that can be set in objects of type cmtextcontent. The value of attribute content stores the text associated with the text component as a string.

**CMPictureContents**

Pictures too can have language and format dependent parts associated with them. To represent these content objects, type cmpicturecontent has been defined. As the following specification taken from the database schema file shows, there are three attributes that can be set in objects of this type.
2.2. DATABASE SCHEMA

```plaintext

type cmpicturecontent subtype of cmcontent

(  image : string;
  alternate : string;
  relative : boolean;
);
```

Attribute image is a string holding the path to the image file on the web server. Similar to the field reference in type cmurl, this path can be specified relative or absolute. The second attribute named alternate provides the possibility to store a string that should be used as an alternate representation of the picture, when the user-agent is incapable of displaying the stored image type. Finally, attribute relative holds a boolean value that contains information about how the image path was stored. If it is set to false, the server engine presumes that the path is specified absolute and renders it without modification into the resulting document. If however the attribute is set to true, it is assumed that the image is located in the configured picture directory of the web server and the path to that directory is prepended to the value of attribute image.

CMMixedContents

The last type of content objects, cmmixedcontent, is used to represent the format or language dependent part of objects having the basic type mixed content. Objects of this type are used, whenever it is required to embed components into text or when a text contains further mark-up. For a complete specification of this data type refer to the sample from the schema file below.

```plaintext

type cmixedcontent subtype of cmcontent

(  content : text;
  keywords : set of string;
);
```

There are two attributes in type cmixedcontent. Attribute content contains the text represented by this object. In contrast to the according attribute in type cmtextcontent this string may contain further mark-up as shown in the sample below.

```
<weboject name="initial_technologies"/> o provide ultimate flexibility CMServer is built on standard technologies. To export data, CMServer uses the eXtensible Mark-up Language (XML). When rendering data, CMServer relies on XML Stylesheet Language (<weboject name="XSL_Link">XSLT</weboject>). XML as well as XSL are W3C standards widely accepted in the commercial and academic field!<linebreak/>
```

The above sample demonstrates a typical use of such an mixed content object. As can be seen from this example, two different types of mark-up elements can be used.

**Formatting** It is possible to include tags, that describe the formatting of the text, such as boldface, italics, underline or line breaks. These tags are not processed by the server engine, but have to be handled in the template responsible of rendering the component. A good place to include transformation that converts these tags, is the generic fall-back stylesheet. In the example shown below, the tag `<linebreak/>` is an occurrence of such a use.
CHAPTER 2. DATA STORAGE

References As this content type represents mixed content, it has to be possible to embed other objects. To restrict this option and to ensure that the principles of content management are not violated, it is not possible to reference any object, but only objects included in the associated container. This assures, that the objects in a mixed content component exist themselves in the databases and prevents uncontrolled introduction of new content. Normally such a reference is resolved by inlining the referenced component. If however there is more mixed content included in the element tag, this data gets appended to the referenced object as special content and can be used by the rendering template to display additional or alternate information. The two webobject tags in the sample given below are examples of such uses, with and without included content.

2.2.3 Languages

Content objects are always associated with a language to provide a possibility to select the accurate content based on the wishes of the requesting user-agent. To represent these languages in the database, type cmlanguage has been created. An exact specification of this type has been included in the sample from the schema file below.

```
type cmlanguage
{  name : string;
  id   : string;
};
```

There are two attributes in this type, name and id. The first one offers a possibility to store a human readable name for the language, that can be used to manage the object in a more user friendly way. The second attribute contains the id of the language. This string is a combination of two-letter shortcut denoting the language and optionally a two-letter shortcut denoting the country. The language Swiss German for instance would be denoted by the id de_CH. This mode of identifying languages is commonly used by all major web user-agents.

2.2.4 Templates

To render the components specified in the database for a given target platform, the system uses templates. These templates are also managed by the database. There are two possibilities to attach a template to a component. First there is the notion of a default template, represented by the association hasDefaultStyle and second, templates can be managed dependent on the context where a given component occurs. The context of a component is defined as the enclosing container object. Consequently the context dependent template is associated by hasStyle to the association specifying the inclusion of a template within a container.

```
type cmtemplate
{  name   : string;
  getTemplate : ( ) -> ( template: string );
};
```

There is only one attribute defined in type cmtemplate. The attribute name is used to provide an intuitive and descriptive name for the template, so users will have an idea what a given templates does. This is important to simplify the selection process of a template for a given component. The actual template is not stored in this type, but in its subtypes.
htmltemplate and wmltemplate. There is however a method to retrieve the template code from the object, which allows unified handling of all subtypes of cmtemplate even when it is not known how these types are going to represent the template code.

2.2.5 Workflow States

As motivated before it is important to be able to classify the components stored in a content management system’s database in terms of an object life cycle. The individual stages of such a life cycle are represented in our system by objects of type cmworkflowstate shown in the definition below.

```haskell
type cmworkflowstate
  ( name : string;
    description : string;
  );
```

Two attributes can be set in objects of this type. The first one named name gives the possibility to store a short name or identifier for this object. The attribute description should be used to specify further information about the given workflow state. It is advisable to include details about the meaning of this state and the implications that can be derived from it for a given component.

2.2.6 Gatekeepers

The concept of object validation is implemented in our system by gatekeeper objects having type cmgatekeeper. Gatekeeper objects can be associated with any object in the system to provide a method of validating a given object.

```haskell
type cmgatekeeper
  ( name : string;
    description : string;
    passesGate : ( comp: cmcomponent ) -> ( res: boolean );
  );
```

Attribute name of type string contains the a short name or identifier for this object. The second attribute named description again provides a facility to specify more information about the given gatekeeper object. Third and most important, there is the method passesGate which takes a component as its argument and returns true or false if the given component is valid or invalid respectively. This method is called for all gatekeepers attached to a component and the results are conjugated to retrieve a final result. In other words a component is only included on a web document if all associated gatekeepers return the value true.
After having laid out how the elements of the website are managed by the database component, in this chapter we want to focus on the process how pages are assembled from these objects. There are three steps involved in the generation of a page. First, an XML document is generated. At the same time an XSLT stylesheet is assembled. The third step of page rendering is the application of the stylesheet to the XML document to produce a representation of the page in the desired mark-up. These three steps are described in this chapter and illustrated with a small example.

3.1 Generation of XML

The task of this first step of the generation of a page is to retrieve the elements of a page from the database, assemble and convert them into an XML document.

3.1.1 Description of the Process

When a user-agent sends a request for a page to the server engine, the following steps are undertaken to produce the desired page.

Parameter Evaluation The user-agent sends the desired page as a parameter of the servlet. This path included in this value is taken and parsed, to retrieve the affected directories and the requested page. Note, that it is also possible to omit the page at the end of the path. In this case the server assumes that there is a page named index.html in the last specified directory.

Page Retrieval With the chain of directories, the servlet tries to map the path to the structure defined in the database. When this mapping is successful, i.e. when the path is valid, the last directory is retrieved from the collection CMDirectories. Now, a collection of all pages contained in this directory is determined by restricting the domain of
association containsDocuments to the retrieved directory. This yield a binary collection whose range is the desired collection of contained pages. If the specified page can be found in that collection, the process to generate the page continues, if not, an empty page is delivered back to the user-agent.

**XML Generation** After the retrieval of the page object, the generation of the XML document can begin. To do so, every encountered CMComponent is asked to produce an XML representation of itself. As this process iterates over the child components of container components, it suffices to convert the page component (which itself is a container component) into XML. All objects contained in the page get converted recursively along the way. Throughout the process of XML generation the Java Document Object Model (JDOM) [8] is used.

If this process completes successfully, the server will be able to proceed to the second and third step of the generation of the page, which process the created XML document further. Before looking at these steps however, we want to introduce a simple sample situation that illustrates the processes at work.

### 3.1.2 A Sample Situation

To show how the generation of page works at a more intuitive level, we shall introduce a small example, which will guide us through this chapter. This sample situation consists of one page with a title and a link in its body. The link component uses a picture component as resource and has a URL as target.

![Figure 3.1: Structure of the Sample Web Page](image)

This constellation is shown schematically in figure 3.1. The page is actually represented by an object of type cmcontainer in the database, but for reasons of comprehensibility it is labelled cmpage in the figure. Note that references that are stored as associations in the database are shown as an arrow with a dot at one end, where as simple attribute references are displayed as single-lined arrow.

**Input Data**

This situation can now be entered into an OMS Database. To do so, one can use the graphical front end of OMS Pro or create a file containing statements of the data manipulation language (DML). As this text-based version of the database content is easier to understand in the context of this report, we have chosen to discuss input data using that representation.
The first step when generating such a DML file, is the creation of all objects. Then every object can be dressed with the respective values. As there are only few objects in our system, we show here how this would be done for our sample situation.

```xml
create o927;
dress o927 as cmcomponent values (  
    name = 'index.html';
    description = 'Sample Website';
);
dress o927 as cmcontainer values (  
    order = [];  
    function = page;
);

create o928;
dress o928 as cmcomponent values (  
    name = 'sampleurl';
    description = 'Sample URL';
);
dress o928 as cmurl values (  
    reference = 'http://www.cmserver.org';  
    internal = false;
    valid = true;
);

create o929;
dress o929 as cmcomponent values (  
    name = 'samplepicture';
    description = 'Sample Picture';
);
dress o929 as cmpicture values (  
    border = 0;
    width = null;
    height = null;
);

create o930;
dress o930 as cmcomponent values (  
    name = 'samplelink';
    description = 'Sample Link';
);
dress o930 as cmlink values (  
    resource = o929;
    target = o928;
);

create o931;
dress o931 as cmcontent values (  
    mimetype = 'text/txt';
);
dress o931 as cmtitlecontent values (  
    title = 'Sample Website';
);```
create o932;
dress o932 as cmcontent values (
    mimetype = 'image/jpg';
);
dress o932 as cmpicturecontent values (
    image = 'CMServerLogo.jpg';
    alternate = 'CMServer';
    relative = false;
);

After creating all objects and dressing them with values, the new objects have to be inserted into the appropriate collection. Usually there is a default collection for each type, but objects can also occur in multiple collections.

insert into collection 'CMComponents': [o927,o928,o929,o930];
insert into collection 'CMContainers': [o927];
insert into collection 'CMPages': [o927];
insert into collection 'CMUrls': [o928];
insert into collection 'CMPictures': [o929];
insert into collection 'CMLinks': [o930];
insert into collection 'CMContents': [o931,o932];
insert into collection 'CMTitleContents': [o931];
insert into collection 'CMPictureContents': [o932];

Finally and perhaps most important, the associations between the objects have to be established. An example of such an association would be the link between the title content and the page in our example (e.g. the tuple (o927,o931) in the data definition file).

insert into association hasContent: [(o927,o931),(o929,o932)];
insert into association hasTitleContent: [(o927,o931)];
insert into association hasPictureContent: [(o929,o932)];
insert into association hasComponents: [(o927,o930)];

This data can now be incorporated into the content management system by modifying the server configuration file cmserver.properties. To get the system working, however, the content objects have to be associated with a default language. Some additional data manipulation statements are necessary to do so.

create o933;
dress o933 as cmlanguage values (
    name = 'English (Default)';
    id = en;
);
insert into collection 'CMLanguages': [o933];
insert into association inLanguage: [(o931,o933),(o932,o933)];

Furthermore, a server root directory has to be created and the page has to be inserted into that directory. Without this step, the server will not be able to find the page. The required DML statements are shown below.
When all these steps have been undertaken, the servlet will be able to produce a document, that will be rendered with default templates. The resulting web page is shown in figure 3.2.

**Resulting XML**

The process described above will create the following XML document when invoked with the URL [http://localhost/oms/servlet/webengine?page=index.html](http://localhost/oms/servlet/webengine?page=index.html) on our sample data.
<webobject oid="o927" type="cmcontainer" name="index.html"
    function="page">
    <property name="name">
        <string>index.html</string>
    </property>
    <property name="description">
        <string>Sample Website</string>
    </property>
    <property name="order">
        <collection id="919" oid="920" membertype="cmcomponent"/>
    </property>
    <property name="function">
        <string>page</string>
    </property>
    <property name="title">
        <string>Sample Website</string>
    </property>
    <components>
        <webobject oid="o930" type="cmlink" name="samplelink">
            <property name="name">
                <string>samplelink</string>
            </property>
            <property name="description">
                <string>Sample Link</string>
            </property>
            <property name="resource">
                <webobject oid="o929" type="cmpicture"
                    name="samplepicture">
                    <property name="name">
                        <string>samplepicture</string>
                    </property>
                    <property name="description">
                        <string>Sample Picture</string>
                    </property>
                    <property name="border">
                        <integer>0</integer>
                    </property>
                    <property name="image">
                        <string>http://localhost/pictures/CMServerLogo.jpg</string>
                    </property>
                    <property name="alternate">
                        <string>CMServer</string>
                    </property>
                </webobject>
            </property>
            <property name="target">
                <webobject oid="o928" type="cmurl" name="sampleurl">
                    <property name="name">
                        <string>sampleurl</string>
                    </property>
                </webobject>
            </property>
        </webobject>
    </components>
</webobject>
3.1.3 A Note on the Used DTD

The document type definition (DTD) underlying the created XML documents is based on the OMS/XML DTD presented in [9]. Furthermore it incorporates some recently added features included in the high-level document type definition by Beat Signer. On the other hand there are some requirements occurring in this system that are not compatible with the database inspired approach used in these DTDs. These proprietary extensions and modifications to the OMS/XML DTDs are summarized in this section.

Renaming To provide the user who is in charge of writing the XSLT templates with a more web-inspired XML document, three elements have been renamed in comparison to the original DTD.

Element instance has been renamed to webobject as these objects are no longer in a one-to-one relationship with a database instance. These new webobject elements combine the information stored in the component object and the associated content objects and therefore they represent rather page elements than database elements.

The second renaming was the one from attribute to property. This change is motivated by the slightly different semantic of these elements. Whereas in the original DTD these tags were denoting attributes of a single instance, they now represent all properties of a component object and its associated content objects. To avoid confusion, the misleading name was changed to a less database and object inspired term.

The last change in terms of new names is a subtle one. The identifier for the database type text has been modified from text to string. As the difference between the two types becomes only apparent in the graphical user interface of OMS Pro, there is no justifiable argument not to map those types to the same element.

Structural Changes There are two structural extensions in the DTD of the documents produced by the servlet. Both modifications are motivated by the concept of a container object, that has to be able to incorporate its child components into its XML representation.
To allow such structures, the element components was introduced as a sub element of a webobject having type cmcontainer. The subelements of the components element are all objects linked to the container by the association hasComponents.

The second of these structural updates affects all webobject elements of data type cmmixed. Note, that these components are a special case of containers too. An object of this type has to be able to incorporate the associated mixed content. Although this as well could be realized by the components element of its super type, we have chosen to introduce a new element mixed for this case. Doing so, it is possible to distinguish between the different types of the sub elements of the respective element more elegantly.

3.2 Generation of XSLT

The second step of the page generation process is the assembling of an XSLT stylesheet containing templates to render all components occurring in the XML document. Actually this process is done in parallel with the above mechanism, but for reasons of simplicity it is presented here as two separate steps.

3.2.1 Description of the Process

As stated, the stylesheet is assembled at the same time as the database components are converted to XML. This has the major advantage, that each component can decide for itself what template should be used to render the object. The stylesheet generation process can be split into the following steps.

Stylesheet Creation  The first thing that is done by the servlet upon receiving an incoming request for a page, is the allocation of a new stylesheet object for the new request. Every request has its own stylesheet, since the stylesheets may be very different depending on what components are included on a page. This stylesheet object is given to each component when it is asked to produce an XML representation of itself.

When the stylesheet is created, a set of variables are created and inserted. These values represent parameters of the server configuration that have to be accessible to the individual templates. Examples of such variables are system paths, date and time values.

Another important feature is the inclusion of the root transformation. A template matching to the root of the XML document is included by default in the XSLT stylesheet. This transformation is of utmost importance to start the recursive transformation process on the stylesheet processor.

Last but not least there always is the import of the generic stylesheet used to render components that do not have an individual template. Another use of the generic stylesheet is a way to specify templates for tags used in mixed content components. This method is preferred than including these templates in the template that is in charge of rendering the given component.

Processing of Components  As the component has a reference of the current stylesheet object when it is rendering itself to XML, it can decide which template should be added
to the stylesheet. There are multiple templates that can be selected by the component for its rendering. Depending on the desired mark-up, different templates have to be selected.

But there is also more than one way to associate templates for the same mark-up with one component. First there can be a default template accessible by the association hasDefaultStyle. The second possibility is the one of a context dependent template specified by the association hasStyle. Normally a component will try to include the default template into the stylesheet. On the other hand if there is a context dependent template, it is considered to be more specific and will be preferred it over a default template.

At all times, templates are only included in the stylesheet if they are not already contained in the stylesheet.

Finishing Processing  
At the end of the XML generation process, the completed stylesheet is taken back and passed along to the stylesheet processor for the next stage of page rendering.

3.2.2 A Sample Situation

To illustrate the process of stylesheet generation, we return to our sample situation. There are four components in the content management system at the moment. We have to include a template for every one of them. The following sections describe the templates that would be required to render this page in HTML.

**cmpage**

This template’s task is to render the top container, i.e. the page itself. To do so it begins by outputting the enclosing \(<html>\) tag. Then in the head of the HTML document it uses the value of the components title property to give the document a heading. After the head is processed the template renders the body of the output document. In this sample template, we simply apply further templates to the children of element components.

```xml
<xsl:template match="webobject[@type='cmcontainer' and @function='page']">
  <html>
  <head>
    <title>
      <xsl:value-of select="property[@name='title']" />
    </title>
  </head>
  <body>
    <xsl:apply-templates select="components/*" />
  </body>
  </html>
</xsl:template>
```
cmppicture

Rendering a picture in HTML is more complex as a lot of attributes can be added to the `<img>` tag. The template first sets the attribute `src` in this tag. Then it tests if there are still more possible attributes to include. Thereby it checks for a border, width, height and an alternate textual representation respectively.

```xml
<xsl:template match="webobject[@type='cmppicture']">  
  <img>
    <xsl:attribute name="src">
      <xsl:value-of select="property[@name='image']" />
    </xsl:attribute>
    <xsl:if test="property[@name='border']">
      <xsl:attribute name="border">
        <xsl:value-of select="property[@name='border']" />
      </xsl:attribute>
    </xsl:if>
    <xsl:if test="property[@name='width']">
      <xsl:attribute name="width">
        <xsl:value-of select="property[@name='width']" />
      </xsl:attribute>
    </xsl:if>
    <xsl:if test="property[@name='height']">
      <xsl:attribute name="height">
        <xsl:value-of select="property[@name='height']" />
      </xsl:attribute>
    </xsl:if>
    <xsl:if test="property[@name='alternate']">
      <xsl:attribute name="alt">
        <xsl:value-of select="property[@name='alternate']" />
      </xsl:attribute>
    </xsl:if>
  </img>
</xsl:template>
```

cmurl

In contrast to the previous template, the template to output a URL component is fairly simple. It only takes the text value of the reference property of the component and transfers it into the output document.

```xml
<xsl:template match="webobject[@type='cmurl']">  
  <xsl:value-of select="property[@name='reference']" />
</xsl:template>
```

cmlink

The template for a link creates an HTML anchor element. First it sets the `href` attribute to its target property by applying another template. Then it checks for a suitable resource to be included inside the anchor element. The first choice would be a special content provided by the user in a mixed content component. If no such content is found the resource property of
the link is used and the according template is applied to it. If even this fails, the resource is
set to the same value as the target, to display the URL on the browser.

    <xsl:template match="webobject[@type='cmlink']">
        <a name="{property[@name='name']}">
            <xsl:attribute name="href">
                <xsl:apply-templates select="property[@name='target']" />
            </xsl:attribute>
            <xsl:choose>
                <xsl:when test="property[@name='special']">
                    <xsl:apply-templates select="property[@name='special']/mixed" />
                </xsl:when>
                <xsl:otherwise>
                    <xsl:choose>
                        <xsl:when test="property[@name='resource']">
                            <xsl:apply-templates select="property[@name='resource']" />
                        </xsl:when>
                        <xsl:otherwise>
                            <xsl:apply-templates select="property[@name='target']" />
                        </xsl:otherwise>
                    </xsl:choose>
                </xsl:otherwise>
            </xsl:choose>
        </a>
    </xsl:template>

Stylesheet Preamble

As discussed, the servlet includes a set of useful variables and values into the stylesheet. To
give an impression of what information is available, the header of a generated stylesheet is
included here. Note that some values may vary due to different configurations and environ-
mental conditions.

    <?xml version="1.0" encoding="UTF-8"?>
    <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        version="1.0">
        <xsl:output omit-xml-declaration="yes" indent="yes" encoding="UTF-8" doctype-public="-//W3C//DTD XHTML 1.0 Transitional/EN" doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" />
        <xsl:variable name="timestamp">
            Thu Feb 22 19:51:11 GMT+01:00 2001
        </xsl:variable>
        <xsl:variable name="picturepath">
            http://localhost/pictures/
        </xsl:variable>
3.3 Application of XSLT to XML

The final step of the page generation process is the application of the XSLT stylesheet to the XML document. The following section describes that process and demonstrates it on our sample situation.

3.3.1 Description of the Process

After the XML document and the according XSLT stylesheet have been created by the previous two stages, they can be combined to produce a document in the mark-up specified by the stylesheet. To do so, a stylesheet processor is employed. For our implementation, we have decided to use the Xalan library, which is part of the Apache XML Project [7].

As Xalan expects the XML data in form described by the document object model (DOM), we have to convert our JDOM representation of both the document and the stylesheet to that representation. After this conversion, the stylesheet processor may perform its task and transforms the data into the mark-up of the user-agent.

3.3.2 A Sample Situation

At this moment, we have one last look at our sample situation. As we specified templates to render the page in HTML, the resulting web document will be of that type.

Figure 3.3: Sample Web Page Rendered By Internet Explorer
Resulting Web Document

As displayed below we get a very simple HTML document corresponding exactly to the situation specified at the beginning of this chapter. How this page is rendered by an HTML capable user-agent can be seen in figure 3.3

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>Sample Website</title>
</head>
<body>
<a name="samplelink" href="http://www.cmserver.org">
<img src="http://localhost/pictures/CMServerLogo.jpg" border="0" alt="CMServer">
</a>
</body>
</html>
```
Implementation

After having presented, how the server component works on a process oriented level in the last chapter, we are going to examine the implementation further in terms of software architecture in this chapter.

4.1 Overview

Before getting into the individual classes and software objects, we present an overview of the server component in figure 4.1.

![Figure 4.1: Overview of the Server Architecture](image)

As can be seen from the figure, the server consists of four major parts. The first component is the servlet itself, which is in charge of accepting the incoming requests from the web server. The servlet works as a controller object, that coordinates the process of generating
pages. The database classes are the second architectural component. These objects represent the data stored on the server and can be accessed and queried by the OMS Java object management framework. As discussed, these database objects are capable of representing themselves in XML. For the generation of the XSL Stylesheet however there has to be a separate component. This stylesheet object provides functionality for template addition and duplicate avoidance. The fourth and last part of the architecture is a mere collection of utility classes catering for various needs. A detailed description of the services offered by all these components is given in the remaining sections of this chapter.

4.2 Servlet

As mentioned before, the servlet is the controller component of the server. It is in charge of evaluating the parameters given to the server contained inside the URL. Depending on the specified parameters values the servlet decides what operations have to be performed. There are a number of possible results the servlet can produce. Most simple, it can generate the requested page as described before. But the server is also capable of updating the database in order to modify the content (not the structure) of a web page. At the end of this section we give an overview over the exact use of parameters that will cause the server to return debugging information, allow to control it remotely or are involved in the process of updating the page content.

4.2.1 Page Generation

The process of page generation is the main task of the servlet component. The normal case of a dynamically built page has been described in full detail in the previous chapter and shall not be repeated here. However the reader should be aware of the fact, that the given description merely covers the normal case of page rendering. There are a number of parameters explained further below that change the behaviour of the server to produce different pages.

4.2.2 Page Updating

When the parameters given to the servlet indicate that the request would like to update the database (i.e. parameter \texttt{update=true}), the servlet starts to perform this task. The first step that has to be done is to extract the attribute values from the parameters. To be able to do so, a parameter format specifying on update has been created. Such a parameter must be of the following form.

\begin{verbatim}
id delimiter attribute "=" value .
\end{verbatim}

The delimiter is one character (usually @) that separates the object id from the attribute name. This character is freely configurable and can be defined in the configuration file in the same directory as the servlet (\texttt{cmserver.properties}) This file is read whenever the server starts. The object id has to map to an object id in the OMS Java database on the server, as this information is used to find and update the appropriate object. An example of such an update parameter would be the following.

\begin{verbatim}
o927@function=container
\end{verbatim}
When updating an object on a website, it is usually hard to figure out, which values have been modified. The simplest way therefore is to transmit all values and have the server handle this situation accurately. In this case the only way to detect if a value has changed, is to retrieve the value stored in the database, compare it with the value given by the client update request and then update the value if necessary. However this procedure is not very satisfying as it causes more database operation than necessary. Note, that in the case of a necessary update, the object in question needs to be retrieved twice, once for reading and once for writing. To avoid this, we have decided to write the objects in any case, whether necessary or not. To further suppress object retrieval we first collect all update parameters sent to the server and sort them by object id. Then the servlet constructs a vector of all (attribute, value) tuples and sends it to the according objects. In doing so, every object has to be loaded only once per update request.

4.2.3 Servlet Parameters

There are a number of further parameters the servlet accepts. These parameters serve for displaying debugging information and remote server administration.

**page=**path Specifying this parameter only triggers the servlet to behave as described in the previous chapter. The string given as path denotes the page the user-agent would like the servlet to build.

**xml=true** If this parameter is specified, the server does not apply the stylesheet to the XML document. The document is serialized and sent back to the client untransformed. This option is useful to track down possible errors in web pages. This can be quite difficult as the data passes several stages before it reaches its final presentation on the user-agent. To simplify such error seeking this option has been included.

**xsl=true** This switch is similar to the one discussed before. The main difference is that this switch triggers the server to send the XSL stylesheet instead of the generated page. To do so, again the stylesheet is not applied to the XML document, but serialized and sent back to the user-agent. This parameter too has been introduced for debugging and controlling reasons. Stylesheet generation is prone to several possible errors. To check if the right templates have been included in the stylesheet, it is quite useful to provide an option for displaying the stylesheet. Doing so, the user can easily check if everything is correct in the browser.

**reload=true** Whereas the first to parameters have been designed with debugging reasons in mind, this third one provides remote control over the server. Specifying this switch as a parameter of the servlet URL will cause the server to reload the database from the DML input file. Thus it is possible to modify the database and have it reloaded without shutting the server down.

**modify=true** As discussed earlier the server is capable of handling update requests. However it would be quite painful to send the necessary parameters manually by including them in the URL by hand in the format presented before. To facilitate the situation a special stylesheet for update requests is supported. Since this stylesheet is not very likely to change often and context dependent rendering is not needed, it is not generated
dynamically. The update stylesheet is an ordinary file located on the server. The exact location has to be specified in the configuration file. How to use this mechanism in full detail is explained in the appendix.

**update=true** Similar to the parameter described before, this one is also involved in the process of updating the page content. When this parameter is sent to the servlet, it presumes that there are update requests contained in the URL and proceeds as described in the previous subsection.

### 4.3 Database Classes

To store the data of a website, we use the OMS Java data management framework. To use this framework it is necessary to implement a Java class for every type declared in the database schema explained before. In addition to these classes there is a number of special objects that help with the database management. In this section we first give an overview of the database package and then point out a few implementation details where it is necessary. A complete reference of all methods and short explanations how they work is given in the appendix.

#### 4.3.1 Package Overview

To give an impression of the structure and content of the database package, we provide a short description of all classes in this section without going too much in the details of the actual implementation.

**Components**

**CMComponent** Class `CMComponent` is the implementation of the database type with the same name. This class plays an important role in the implementation of the database objects, as it includes methods to render every feature of an OMS Java instance in XML. However, some of these methods have been modified to meet the requirements of the content management system rather than just transforming database objects. Finally, there are some utility methods that perform customized queries on the database and help with the administration of the associated content objects.

**CMContainer** Class `CMContainer` implements the concept of the container objects in the database. To do so, it overrides the methods responsible for generating XML defined in its superclass and provides functionality to handle context-dependent templates.

**CMDirectory** The database component representing a directory on the server is implemented by class `CMDirectory`. Beside the database-specific methods, there is a set of utility methods and a customized method to generate an XML representation of the directory.

**CMLink** Links in the database are represented by class `CMLink` in the OMS Java object management system. This class only provides the methods required by the OMS Java framework and defines no further features of its own.
CHAPTER 4. IMPLEMENTATION

CMMixed Class CMMixed represents the database type for mixed content components. Since these object treat their contained components differently than a normal container component, a special implementation of the XML generating method is provided. Furthermore there are methods to handle the management of the content objects associated with this component.

CMNavigation Navigation objects are computed objects, that build their structure based on the data in the database and the specified parameters. Therefore the main part of class CMNavigation focuses on the generation of an XML element that represents the navigation component.

CMPicture Picture components are implemented by class CMPicture. The only responsibility taken by this class is the management of the associated picture content objects.

CMSitemap Similar to navigation components, sitemaps too are purely calculated objects. Their behaviour can be influenced by a set of parameters, but the better part of class CMSitemap deals with the computation of the XML representation of such a sitemap.

CMText Equal to the class representing picture components, CMText deals mainly with the administration of the associated text content objects.

CMUrl Class CMUrl is another very simple class that just provides the methods necessary to function as a part of the OMS Java framework.

Contents

CMContent Class CMContent is a very simple class. There is only one method not part of the mandatory implementation of an OMS Java instance. This method is involved in the generation of XML and is called by the associated component when it is generating an XML representation of itself.

CMMixedContent The mixed content database type is represented by instances of class CMMixedContent. Beside the methods needed by the object management framework, this class adapts the method called for XML generation according to its requirements.

CMPictureContent Representing the format dependent part of picture components is the task of class CMPictureContent. Again there is nothing special in this class with the exception of the adapted method for XML generation.

CMTContentType Instances of class CMT ContentType are the representation of the database type of the same name. Everything in the class can be expected from the definition of the type in the database. However there is again a customized method for XML generation.

CMTContentType Any component can have a language dependent title. Instances of class CMT ContentType are used to represent these objects in OMS Java. The only special method of the class is the modified process of XML generation.
Gatekeepers

**CMGatekeeper** The gatekeeper objects in the database are represented by instances of class `CMGatekeeper`. There is no method in this class apart from the required ones.

**CMWorkflowGatekeeper** Class `CMWorkflowGatekeeper` implements a special gatekeeper object that decides upon the workflow state of a given object. To do so the respective method has be specialized.

Languages

**CMLanguage** Representing the supported languages of the content management system is the task of class `CMLanguage`. With one exception there are only methods required by the OMS Java framework. The only additional method deals with the handling of language regions, i.e. combination of languages and countries.

Templates

**CMCTemplate** Class `CMCTemplate` represents the super type of all template objects in the database. The implementation merely covers the features required by the framework.

**HTMLTemplate** Instances of class `HTMLTemplate` are the representation of templates that produce HTML documents when applied to an XML document. The class is simply an implementation of the database type and provides no additional methods.

**WMLTemplate** WML documents are produced by applying the appropriate templates to an XML document. These templates are represented in the OMS Java framework by instances of class `WMLTemplate`. This class too is simply an implementation of the specified database type.

Workflow States

**CMWorkflowState** To represent a workflow state in the content management database, instances of class `CMWorkflowState` are used. The class itself is merely an implementation of the according database type and offers no additional functionality.

Support and Utility

**CMDBNames** This is a pure utility class, providing a common place to store all names used in the schema. These names need to be accessible to all database classes in order to be able to perform queries and retrieve collections.

**CMDBProxy** The class `CMDBProxy` is a singleton that is used to work with the OMS Java database. This concept was introduced to ensure synchronized access to the objects. The singleton provides also methods to reload and update the database.

**CMDBUpdate** Instances of class `CMDBUpdate` are used in the process of updating the database. One such object encapsulates an (attribute, value) pair. When the servlet is updating the database, it constructs a vector of such tuples for each object affected by the update.
**CMDBUtils** Class CMDBUtils is a collection of useful methods when working with the content manager database. There are utility functions for setting attributes, retrieving special objects or finding the best possible language match for a content object.

**CMPageContext** Objects of type CMPageContext are used when building sitemaps and navigations. They provide a possibility to store a page, its current directory and its parent directory. This becomes important, when the user navigates down a path in the structure of the website and navigation objects have to be able to reconstruct the way back to the top.

### 4.3.2 Detailed Implementation Notes

In this section, we have another, more detailed look at some classes presented before, that contain some algorithms worth commenting. Hopefully this results in a better understanding of the mechanisms at work.

**CMComponent**

Since class CMComponent is a central class of our software architecture, we will discuss it further. The most important task of the class is the generation of XML. It provides methods to render all constructs of OMS Java as XML elements. There are two methods, that will produce an XML representation.

**toXMLElement()** This method converts the given component object into an XML element named `webobject`. Then it adds all attributes of the component as `property` elements to the representation. The associated content object are included next and finally the method checks for the presence of a special content attribute that will be appended if present. This method is used whenever the full representation of an object is required.

**toXMLSummary()** As suggested by the name of this method, it only produces a summarized representation of the component. This representation includes the content objects, but leaves out the attributes and possible special content objects. This is used to save space when the omitted data is not required.

Again there is a small difference to the document type definition used normally in the OMS Java framework. The attribute name of a component is not only represented as a `property` element as required by this DTD, but is also included as an attribute to the `webobject` element. This shortcut has been introduced to make references to such components from within templates easier.

Another important feature is the handling of the associated content objects. There are two methods involved in this task. There is a number of methods to retrieve content objects from the database and append them to the XML representation. Most important however is the method `setAttributeValue` which deals with setting attributes values for this object. This method had to be modified in order to allow correct updates of the content objects. The problem here is, that the client who sends the updates does not know about the separation of component and content. Consequently the component has to delegate the update affecting its content to the appropriate content object.
Finally there is one important method, that has to be called to include the associated template of a component into the stylesheet. The method `insertXSLTemplate` first tries to retrieve the template from the database. If the attempt is successful (i.e. if there is a default template), the template will be included in the stylesheet if not already present. This method has to be called whenever a component is rendered to XML, or else the server will not be able to transform it for the target platform.

**CMContainer**

Class `CMContainer` is discussed here to show how the process of generating XML has been modified to incorporate the concept of a container. The main novelty is the inclusion of the components associated as children of the container. This process begins after the rendering of the common properties of a component, which is done by a call to the superclass’ version of `toXMLElement()`.

After these data has been included, method `addChildren()` starts with creating the special `component` element to which it appends all the contained components. To respect the order given to the container by the according attribute, it first tries to include the components specified in this ranking. Only then are the remaining components included.

Similar to its ancestor `CMComponent`, this class defines a method to handle the inclusion of templates. As there can be a special, context dependent template associated with component contained in a container, method `insertContainerXSLTemplate()` has to be called whenever a component is added to the XML representation of a container. To cater for the case that there is no such template, the method also tries to find a default template as fall-back value.

**CMMixed**

This class representing the data type mixed content is a special case of a container. The semantics of the represented data type however made it necessary to change the process of XML generation further. As described, mixed content in our system is a string with references to other components, that should be embedded into the text. This requirement means, that the components in the container (i.e. the referenced objects) do not need to be appended to the container, but have to be inlined into the text.

To implement this behaviour, method `addContents()` in charge of adding the associated contents to an arbitrary component has been overwritten to add the mixed content object and at the same time inline the referenced components. Method `inlineObjects()` is responsible for parsing the text, finding the referenced objects and inlining them into the text. Note that to prevent the objects from being added to the container as well, method `addChildren()` is overridden and left blank.

**CMContent**

Content objects are represented by instances of class `CMContent` and its subclasses. To influence how the attributes of the content object are included into the representation of the XML element of the component, there is the method `appendXMLElements()`. When creating the XML element the component will call this method with the element as parameter to allow the component to insert its properties. In this way, the separation of the component
and its content can be made transparent to the client while still allowing the content object to decide by itself how its going to be rendered.

**CMWorkflowGatekeeper**

Although the concept of gatekeepers is a simple one, the special case of the workflow gatekeeper deserves some explanation. To show how the gatekeeper works, we introduce a small example. First of all there has to be a sample workflow. For the sake of the example we choose the very simple case shown in figure 4.2.

![Figure 4.2: Sample Workflow](image)

An object in a system having this workflow can have the following life cycle. When the object is created and worked on by its author, it is in state Test. As soon as the object is ready for publishing, the author migrates it to state Approval. Now it can be approved by the author’s supervisor and moved to state Approved or Rejected. Objects in state Approved are the only objects visible to the client. Components having state Rejected are sent back into the control of their author to be modified according to the specification of the supervisor. This requirement is enforced by the workflow gatekeeper.

To classify which states allow a component to pass the gatekeeper, class CMWorkflowGatekeeper provides three collections as attributes. These collection serve to specify which workflow state should be visible to what group of people. The three collections are owner, supervisor and browser. Components having a workflow state in the first collection are visible to the author only. The second collection contains the workflow states of the objects visible to the supervisor and the owner. Collection browser contains the workflow states of the components visible to everyone. In our example we would insert state Approved into collection browser, to get the describe behaviour.

When checking a component, the workflow gatekeeper retrieves all workflow states associated with the object from the database and ensures that at least one of these states is contained in the collection browser. If this condition holds the object passes the gatekeeper and is included in the XML document. Although this concept supports personalized views of the website for different users, this concept has not been implemented since a mechanism for user management and object ownership is not yet provided by the OMS Java framework.

**CMDBUtils**

Class CMDBUtil is a collection of useful methods that implement often used functionality which cannot be attributed to one single object type. These methods are described here in
4.3. DATABASE CLASSES

detail.

**ensureType()**  This method is used to ensure that the value of a given attribute matches the type expected by the OMS Java framework. Since the method is used when updating the database, the value is given as a string found in the parameter list of the update request. If the value expected by the object management system is not of type string, the method will convert the string to the required type using the reflection application programming interface.

**getAttribute()**  Given an OMS Java instance and an attribute name, this method will try to find the object representing the according attribute. This should not be confused with the lookup of the value of the attribute.

**getXSLTemplates()**  There are two versions of this method at the moment. The first with one parameter retrieves the default template associated with an object. The second is given a container as a second argument and thus will retrieve the context dependent template associated with the component.

**getBestLanguage()**  The task of this method is to calculate the best possible language among a collection of languages. It is given a default language, that is configured for the website and the desired language stated in the request by the user-agent. As languages consist of a language together with a country, there is the possibility that the request is over- or underspecified. The method first tries to find an exact match. If this fails, it tries to find a language available, that matches the language part only. Finally the last resort is to check if the default language is present and return that language.

**lookupPage()**  Looks up a page based on a given path and a separator char. The method returns a so-called page context containing the page, if found, its directory and the parent directory of the page's directory.

**lookupDirectory()**  This method looks up a directory given by its name. There are two versions of the method one searching through all directories and one that takes a collection of directories it should search in. The second version is for instance used to look up a directory among the subdirectories of the current directory.

**vectorizeCollection()**  Takes a collection and constructs a vector with the elements contained in the collection. This is used, when rendering the components of a container according to the *order* attribute. As described this is a two phase process. First the components for which the order is known will be rendered. Then the remaining components are included in the XML document. To be able to tell which components have been rendered using the *order* attribute, we construct a vector using this method and remove all elements, after being included in the document. At the end, the vector contains only the remaining components for which no order has been specified. By traversing the vector they will be appended to the document as well.

**getTypeString()**  This utility method returns the name of the type of a template when given the according mime type. It is used to map the mime types requested by the user-agent to types in the database. This method will have to be changed, when extending the system to further template types in the future.
4.4 XSL Stylesheet

The package for XSL stylesheets contains only one class. This class extends the JDOM Document class and represents a stylesheet in our systems. Stylesheets are a special case of XML documents containing only elements from the xsl namespace.

The constructor of class XSLStylesheet creates the stylesheet for a given mark-up. It inserts the mandatory root element and some additional arguments for the configuration of the stylesheet. Additionally, it includes some values of the server configuration as XSL variables. Finally, the root transformation is included to start the transformation process.

There is one important method in the class representing a stylesheet. This method is used to insert a template into the stylesheet. To prevent duplicates, however, method addTemplate first checks if the template is not already contained in the stylesheet. This check is done using an open (sic!) hash table of object identifiers of the already contained templates.

4.5 Utility Package

The last part of the server architecture is a set of utility classes. In this chapter, we give an overview of all classes contained in the package. Where necessary, we will also include the implementation of specific methods of these classes.

4.5.1 Package Overview

BuildParameters

An object of type BuildParameters encapsulates all parameters that have to be accessible by the individual components when creating an XML representation of themselves. These parameters include, for instance, the desired language and the desired mark-up specified by the request. But there are also request-independent parameters like the configured default language and the context of the current page. The field mode provides information about the current rendering mode of the server (i.e., whether pages are rendered for modification or not). Probably, the most important parameter is the stylesheet which is provided to the components so they can include the appropriate templates.

DMLJunkFilter

This class has to be created to pre-parse the DML files that are read in by the server. These files may contain lines that had to be introduced to be compatible with OMS Pro (e.g., lines starting with omssbug), but are not needed when working with OMS Java. Class DMLJunkFilter removes these lines from the file. This is done by reading the file line by line and eliminating the lines starting with a given sequence of characters. After the whole file has been read, the file is closed and reopened for writing. To make the changes persistent, the updated content is written back to the DML file.

HelperFunctions

Class HelperFunctions contains various methods having mere utility character. The most important and useful ones are described here in further detail.
splitString() Given a string and a separator char, this method will split the string and return a vector of the resulting tokens.

createElement() This utility method creates an XML element with the given name, attribute and attribute value.

getPathPrefix() Returns the path prefix that has to be prepended to each internal page, when rendered as a link. This path contains the used protocol, the server name and the path to the servlet.

getPath() Constructs the complete path of a directory or a file on the server. This is done by prepending the server protocol and the server name to the given path or file name.

getPath() Given a mime type, this method returns a string denoting the corresponding mark-up type. This string is used, when the stylesheet is configured. For instance if given mime type text/html, the method returns html.

LanguageRegion

An object of type LanguageRegion represents the combination of a language and a country where this language is spoken. This is used to model the format of a language request as issued by the user-agent. The class provides various methods to compare two such regions. To be able to find the best possible match, when searching for a content object in a given language, the class supports comparison of only one of the two criteria.

OMSPolymorphism

A rather unpleasant feature of the OMS Java model is the different behaviour of polymorphic accesses on object. Whereas object oriented languages such as Java return the most specific type of the object in question, OMS Java returns the type on which the association is defined. To get around this problem, class OMSPolymorphism provides a method getLeafInstance(), that will retrieve the appropriate instance. There is however a problem, when an object can be of more than one subtype. In this case the user has to specify a preferred type, when calling the method or it will return null.

ServerProperties

Class ServerProperties deals with the basic configuration of the content management server. There is a configuration file (described in the appendix) that is read in and parsed by this class. The properties file is loaded on the first request for a property value. If the properties should change during the execution of the server, there is a method to reload the values from the file.

XMLElementer

Parsing and construction of XML elements is the main purpose of class XMLElementer. There are two methods that, when given a string of XML mark-up, will construct the according tree and two that can parse a string of mixed content. Each type of method is provided for elements containing a namespace and such with no namespace.
XMLExportable

XMLExportable is an interface that defines the two methods used for XML generation. There is one method that should produce the complete representation of a component (toXMLElement()), and one (toXMLSummary()) that renders a summarized version only. This interface is implemented by all classes representing components in our content management database.

XMLTagnames

Class XMLTagnames provides constants for all tag names used in the process of XML generation. This class was introduced to store these strings in a common location and not have them scattered around various classes.
Conclusions

After having presented the whole implementation of our system in detail in the previous chapters, we will present some conclusions and lessons learned in the next sections. At the end of this chapter we include some ideas how the system could be developed further in the future.

5.1 Lessons Learned

In this section we first compare, how the system fulfils the requirements stated at the beginning of this diploma thesis. Then we have a look at the technologies used and give some comments about their suitability for such a system. Finally we present some aspects that have not been implemented to full satisfaction.

5.1.1 Fulfilment of Requirements

The requirements for a content management system introduced in chapter 1 could all be fulfilled. The following overview presents all requirements again and shows how they are implemented in the system.

Separation of Content, Structure and Layout The content is stored in the database as separate objects and is therefore separated from the layout, defined by the templates and the structure of the website which is stored as associations.

Abstraction of Objects All components of a website are stored as objects in the database. The encapsulation of data as objects is therefore granted.

Multiple Presentation Channels The use of templates to render components allows to support an unlimited number of presentations.

Multilingual Content By separating the component from its content, it is possible to support different versions of the content.
5.2. FUTURE WORK

Workflow and Object Lifecycles  The introduction of arbitrary workflows that can be defined by the user supports these concepts.

Object Validation  Each object has to pass a gatekeeper before it is included in a web page. This concept caters for object validation.

Extensibility  When a user designs a web page, it is possible to define own types that are extensions of system components. By using the defined web data types, these objects get additional features for free.

5.1.2 Used Technologies

In the next two sections, we analyse the technologies used and make some comments to their suitability for our content management system.

XML and XSLT

XML combined with XSLT provides a very powerful concept to separate data from representation. Using XSLT opens the possibility to generate nearly every target document that one can imagine. Yet out of the perspective of the user the use of a new concept may cause rejection. When comparing advantages and disadvantages however it is clear that the use of these technologies is highly suitable for such a system.

OMS Java

OMS Java is very suitable for such a project, as it allows a completely object-oriented approach when designing the system. The concepts can be modelled very clean and elegantly. One drawback however is that the performance of the system is not optimal at the moment, when large collections or complex queries are involved. Further improvements in this area would be very helpful.

5.1.3 Open Issues

Although the requirements have been fulfilled as described above, there is one major drawback in the system at the time. When bringing content to the web, the information is often already available and it would be desirable to integrate this legacy data. Unfortunately this is not easily possible with our system, as it cannot use the data of other databases. When thinking about this problem one may find, that this is no trivial task and further research has certainly to be done.

5.2 Future Work

In the last section, we presented some features that have not been implemented in our system. In the remaining section we give some hints, how the system could be developed further to incorporate these concepts. Furthermore there are some other possible projects suggestions that might help to improve the application and use of our content management system.
5.2.1 Open Issues

As mentioned before, it would be useful to have a means to integrate existing databases. This is however not an easy task when the requirements of content management should be applicable. For instance, one has to think about the problem how to handle multilingual concepts with an input database providing only means for one language. It is clear, that a simple, yet powerful solution is hard to find. Nevertheless we strongly believe that the basic concepts in our system will also have to be present in a such an advanced system. Constructs like containers, pages and directories are of utmost importance when trying to structure the raw data of a database.

5.2.2 Further Enhancements

At the moment, the system is very hard to use, as there is no interface providing an easy way to work with the system. A very important addition to the system would therefore be the implementation of a tool, that allows to construct web pages and manage a website on a more intuitive level. This tool should hide the details of the database implementation and present a simplified view of the system to the user. It could also implement some default behaviour, such as mapping objects to default languages, gatekeepers and workflow states.
5.2. FUTURE WORK
Server Setup

This section should help to install the CMServer application on your system and help troubleshoot possible problems. The CMServer application runs as a servlet inside the Tomcat servlet container on an Apache HTTP web server. Before starting make sure, that you have installed the newest versions of these products. To download and get information on how to setup them visit http://www.apache.org. If the setup of these components completed successfully, working through the next steps will help you to launch the CMServer application.

A.1 CMServer

The first step is to extract the CMServer application and place it in a suitable directory. For this example we use the directory develop, located in the root directory of the file system. Unzipping the archive will create a directory structure below the specified directory containing the OMS Java system including the CMServer application. The main servlet is located at the following position.

<installation directory>\org\omsjava\cmserver\CMServlet.class

A.2 httpd.conf

The file httpd.conf contains the configuration of the Apache web server. Normally it is located in the conf subdirectory of the Apache installation directory. It is necessary to edit this file to include the Tomcat configuration as well. This can be achieved by adding the following line at the end of the file.

include "c:/progra-l/apache-l/jakarta-tomcat/conf/tomcat.conf"
Note that the use of slashes instead of backslashes and vice versa is subject to change. If you installed Tomcat in a directory containing blanks in its name, it is advisable to use the eight character representation of that directory (e.g. progra~1 instead of Program Files).

### A.3 tomcat.conf

This file is used to configure the Apache web server to work together with Tomcat. Be sure that this file is the same specified in the above section. First we have to set up a virtual directory on the web server that points to our installation of the CMServer application. In this example we shall call this directory oms. To do so, we add the following lines to the Tomcat configuration file.

```
Alias /oms C:/develop/org/omsjava/cmserver
<Directory "C:/develop/org/omsjava/cmserver">
    Options Indexes FollowSymLinks
</Directory>
```

The next step is to set up a port used by the Apache web server to connect to the Tomcat servlet container. This port number will be used elsewhere, so do not forget it. Insert the following line into the configuration file to do so.

```
ApJServMount /oms/servlet ajpv12://localhost:8019/oms
```

Finally, it is advisable to set up a directory for the images and the stylesheets. This can be done by extending the configuration with the lines shown below.

```
Alias /pictures C:/develop/org/omsjava/cmserver/pictures
<Directory "c:/develop/org/omsjava/cmserver/pictures">
    Options Indexes FollowSymLinks
</Directory>

Alias /stylesheet C:/develop/org/omsjava/cmserver/stylesheet
<Directory "c:/develop/org/omsjava/cmserver/stylesheet">
    Options Indexes FollowSymLinks
</Directory>

<Location /oms/WEB-INF>
    AllowOverride AuthConfig
deny from all
</Location>
```

### A.4 server_cmserver.xml

The last file one needs to modify is the file server_cmserver.xml. This is the configuration file for one servlet application and is used by Tomcat, when starting the servlet container. It is possible to take the sample file provided with the Tomcat distribution, rename it and extend it to include the lines below.
A.5 Startup Apache and Tomcat

Now everything is ready to start Apache. This can be done either by using the menu item in the start menu or the script in the bin directory. To test if Apache started without problems, the URL http://localhost can be loaded in the browser. If there is a page congratulating on the installation of Apache, the next step is to start Tomcat executing the following command from the installation directory of Tomcat.
```
startup -f conf \server_cmserver.xml
```

The servlet should now be reachable under the following URL.

```
http://localhost/oms/servlet/webengine
```

The basic servlet configuration is now complete and the configuration of the server may begin.
To configure the CMServer application, there is the `cmserver.properties` file which has to be located in the same directory as the servlet class. This section discusses the various configuration options and gives helpful default values.

### B.1 Information About the Database

**cmserver.database.workspace** This option sets the string that was used in the DDL schema file to name the workspace. The suggested default value is `CMServer`.

**cmserver.database.schema** Use this option to set the path to the DDL schema file, for instance `c:/develop/sample.ddl`.

**cmserver.database.data** Use this option to set the path to the DML data file, for instance `c:/develop/sample.dml`.

**cmserver.database.checkpoint** To clean up memory, the server performs the checkpoint operation on the workspace after a given number of requests. This option is used to set this number.

### B.2 Information About the Server Configuration

**cmserver.server.serverprotocol** Sets the protocol of the server. As we are using an Apache http web server, this is normally set to `http`.

**cmserver.serverservername** This option has to be set to the same string as found in the file `httpd.conf` of the Apache configuration. If the application is run locally, this is just `localhost`.

**cmserver.server.servletname** This has to be set to the same name as configured for the Tomcat servlet container. The proposed default value is `webengine`.
B.3 INFORMATION ABOUT THE WEBSITE

**cmserver.server.servletpath** Use this option to set the path to the servlet directory of the Apache web server. This has to be done in accordance with the configuration of the webserver or internal links will not work. If the example from appendix A is used, this will be /oms/servlet/.

**cmserver.server.picturepath** This option is used to set the path to the picture directory of the web server. Again this should be the same directory as configured for the Apache server, normally /pictures/.

**cmserver.server.stylsheetpath** Similar to the option before, this parameters points to the stylesheet directory. Normally it is best to just use /stylesheet/.

**cmserver.server.parameter.reload** The name of the parameter used to trigger a reload of the database. A default would be reload.

**cmserver.server.parameter.modify** The parameter used to indicate to the server that it should produce the pages for modification. Normally set to modify.

**cmserver.server.parameter.update** Used to set the name of the parameter telling the server to switch to the update mode. Setting it to update would be a reasonable default.

**cmserver.server.parameter.page** This option is used to set the name of the parameter indicating to the server which page it has to render. Use page if there is no reason against it.

**cmserver.server.parameter.xml** To render the page in XML without application of templates the parameters with the here specified name is used. Setting it to xml causes no harm.

**cmserver.server.parameter.xsl** The parameter with the here configured name will trigger the server to return the stylesheet instead of the page. The proposed default value is xsl.

---

### B.3 Information About the Website

**cmserver.website.documentroot** Sets the name of the root directory to the given name. This is the replacement of the awkward root attribute in type cmdirectory. Every value is possible, but root is a sensible default.

**cmserver.website.defaultdocument** Configures the name of the default document the server tries to find, when given a directory with no page to render. To simulate standard behaviour, index.html is a good default value.

**cmserver.website.defaultlanguage** Sets the default language region of the website. English sites would use something like en_US.
APPENDIX B. SERVER CONFIGURATION

B.4 Information About XSL Stylesheets

cmserver.stylesheet.html.update Sets the name of the update stylesheet, used to render pages for update on an HTML capable client. The name of the stylesheet should include the mark-up it produces, so cms-update-html.xsl would be a good name.

cmserver.stylesheet.html.generic Sets the name of the default stylesheet to be used for HTML clients. Using cms-generic-html.xsl is a good default.

cmserver.stylesheet.html.encoding Configures the encoding of the HTML documents generated by the server, for instance UTF-8.

cmserver.stylesheet.html.doctype-public Sets the document type of the generated HTML documents. This set of possible values can be found at the W3 consortium, setting it to -//W3C//DTD XHTML 1.0 Transitional//EN will work.

cmserver.stylesheet.html.doctype-system This option sets the document type system of the generated HTML pages. The value has to be an URL of a DTD, for instance http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd.

The options here are applicable to other mark-up types as well. For instance it is possible to set all these parameters for WML by replacing html by wml.
As described before, the servlet is able to update the database. This is achieved through update requests sent to the server as parameters contained in the URL. The mechanisms involved have been described in full detail in previous chapters. In this appendix we show how to use the update stylesheet to provide a user-friendly update process.

## C.1 Update Templates

In this section we examine the templates that have to be included in the update stylesheet to be able to modify our webpage from the sample situation introduced in a previous chapter. The basic idea is to render all components of a webpage as form elements instead of pictures, links, tables, etc. We also include a button to submit the form back to our servlet. When submitting the form, the user-agent will construct a URL containing parameters of the following form.

```plaintext
formelement "=" value .
```

To make the update work, we have to be sure that the string `formelement` denotes the id of the appropriate object and the attribute in question separated by the configured delimiter character. Another important task of this stylesheet is to include the `update` parameter in the URL. This is most easily achieved by declaring the following hidden field in the form.

```html
<input type="hidden" name="update" value="true" />
```

The templates described below implement all these ideas. They render the necessary attributes as form elements, name them appropriately and generate a form, that sends the required parameters to the servlet when submitted.
The template for the page object is the most complex update template as it is in charge of the form and the action when the form is submitted. Note, that we give no value for the action attribute to prevent the browser from loading another page. This is done to achieve the semantic of apply rather than submit.

```xml
<xsl:template match="webobject[@type='cmcontainer' and @function='page']">
  <html>
    <head>
      <title>
        <xsl:value-of select="property[@name='title']" />
      </title>
      <meta http-equiv="pragma" content="no-cache" />
    </head>
    <body bgcolor="#FFFFFF">
      <form action="" method="post">
        <input type="hidden" name="update" value="true" />
        <table bgcolor="#CCCCCC" width="100%" border="0" cellspacing="3" cellpadding="3" align="center">
          <tr>
            <td>
              <h2>
                <xsl:value-of select="property[@name='name']" />
                Page Title<br />
                <input type="text" size="90">
              </h2>
            </td>
          </tr>
          <xsl:for-each select="components/webobject">
            <tr>
              <td>
                <xsl:apply-templates select="." />  
              </td>
            </tr>
          </xsl:for-each>
        </table>
        <br />
        <div align="center">
          <input type="submit" value="Apply" />
          <input type="reset" value="Reset" />
        </div>
      </form>
    </body>
  </html>
</xsl:template>
```
cmpicture

The template for a picture is also quite big. However, this time the only reason for complexity is the number of attributes that can be modified in a picture.

```xml
<xsl:template match="webobject[@type='cmpicture']">  
<table bgcolor="#AAAAAA" width="100%" border="0" cellpadding="3" cellspacing="3" align="center">
<tr>
<td>
<h2>
<xsl:value-of select="property[@name='name']" />
</h2>
Image Source<br />
<input type="text" size="90">
<xsl:attribute name="name">
<xsl:value-of select="@oid" />
<xsl:text>@image</xsl:text>
</xsl:attribute>
<xsl:attribute name="value">
<xsl:value-of select="property[@name='image']" />
</xsl:attribute>
<br />
Image Border<br />
<input type="text" size="90">
<xsl:attribute name="name">
<xsl:value-of select="@oid" />
<xsl:text>@border</xsl:text>
</xsl:attribute>
<xsl:attribute name="value">
<xsl:value-of select="property[@name='border']" />
</xsl:attribute>
<br />
Image Width<br />
<input type="text" size="90">
<xsl:attribute name="name">
<xsl:value-of select="@oid" />
<xsl:text>@width</xsl:text>
</xsl:attribute>
<xsl:attribute name="value">
<xsl:value-of select="property[@name='width']" />
</xsl:attribute>
<br />
</td>
</tr>
</table>
</xsl:template>
```
As before the template to render a URL as a form element is quite simple. The only attribute that shall be modifiable is the reference itself. Note however, that the changing of one URL component may alter the behaviour of many objects on the website as a single URL is typically used in many links.
cmlink

Rendering a link is conceptually simple as a link has no data of its own. The template thus only prints the name of the link component and applies the appropriate templates to the resource and the target component referenced by this link.

```xml
<xsl:template match="webobject[@type='cmlink']">
  <table bgcolor="#BBBBBB" width="100%" border="0" cellpadding="3" cellspacing="3" align="center">
    <tr>
      <td>
        <h2>
          <xsl:value-of select="property[@name='name']" />
        </h2>
      </td>
    </tr>
    <tr height="2" />
    <tr>
      <td>
        <xsl:apply-templates select="property[@name='resource']" />
      </td>
    </tr>
    <tr height="2" />
    <tr>
      <td>
        <xsl:apply-templates select="property[@name='target']" />
      </td>
    </tr>
  </table>
</xsl:template>
```

C.2 Generated Document

When putting all these templates together and applying them to the XML data presented in chapter 3, the HTML document given below will be created. The entire file is included here for reasons of completeness and to help the reader with understanding how the update mechanism works as a whole.

```html
<html>
<head>
  <title>Sample Website</title>
  <meta content="no-cache" http-equiv="pragma">
</head>
```
<body bgcolor="#FFFFFF">
<form method="post" action="">
<input value="true" name="update" type="hidden">
<table align="center" cellspacing="3" cellpadding="3"
border="0" width="100%" bgcolor="#CCCCCC">
<tr>
<td>
<h2>index.html</h2>
Page Title<br>
<input size="90" type="text" name="o927@title"
value="Sample Website">
</td>
</tr>
<tr>
<td>
<table align="center" cellspacing="3"
cellpadding="3" border="0" width="100%"
bgcolor="#BBBBBB">
<tr>
<td>
<h2>samplelink</h2>
</td>
</tr>
<tr height="2"></tr>
<tr>
<td>
<table align="center" border="0"
cellspacing="3" cellpadding="3"
width="100%" bgcolor="#AAAAAA">
<tr>
<td>
<h2>samplepicture</h2>
Image Source<br>
<input size="90" name="o929@image"
type="text" value="CMServerLogo.jpg">
<br>
Image Border<br>
<input size="90" type="text"
name="o929@border" value="0">
<br>
Image Width<br>
<input size="90" type="text"
name="o929@width" value="">
<br>
Image Height<br>
<input size="90" type="text"
name="o929@height" value="">
<br>
Alternate Text<br>
<input size="90" type="text"
name="o929@alternate" value="CMServer">
<br>
</td>
</tr>
</table>
</td>
</tr>
<tr height="2"></tr>
<tr>
<td>
</td>
</tr>
</table>
</td>
</tr>
</table>
</tr>
</form>
</body>
The page as rendered by an HTML capable user-agent is displayed in figure C.1 using Internet Explorer. As the screenshot shows, the user can now change the data very comfortably, without worrying about database concepts.
Figure C.1: Update Page Rendered By Internet Explorer
Acknowledgements

I would like to thank Beat Signer for his generous and continuing support during my diploma thesis. Discussions with him were always very interesting and helpful. I am also very grateful to Moira Norrie for the opportunity to do this diploma project in her group. Moira has always been very encouraging and supportive of all my work at the ETH.

For bringing me in touch with the topic of content management and offering me the best possible internship, which finally led to this thesis, I would like to thank Obtree, Inc.

Pedro Gonnet has improved this work with his never ending criticism of my work and some helpful remarks about my English. Lisa von Boehmer has always provided me with a very comfortable working environment through Kinder Überraschungen, discussions about things other than computer science and delicious meals at the VIS office. Alex de Spindler convinced me that it is sometimes necessary to stop working and get away from it all. Our snowboarding trip to Davos helped this work to thrive by allowing me to take a look at the big picture. Paul Groenewoud has read my report over and over again and pointed out several errors and misunderstandings. Biörn Biörnstad provided support with technical problems concerning the VIS infrastructure. I am thankful to all of them.

Finally, my I would like to express my deepest gratitude to my family for their support of my ideas and projects throughout my entire life. This thesis would never have been possible without them.
Bibliography

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, 1995.


