KUNGL TEKNISKA HÖGSKOLAN
Royal Institute of Technology
Numerical Analysis and Computing Science

TRITA-NA-D9911 • CID-53, KTH, Stockholm, Sweden 1999

Conzilla - Towards a Concept Browser

Mikael Nilsson and Matthias Palmér

CID
Centre for
User Oriented IT Design

Mikael Nilsson and Matthias Palmér

Conzilla - Towards a Concept Browser

Report number: TRITA-NA-D9911, CID-53
ISSN number: ISSN 1403-073X
Publication date: August 1999

E-mail of authors: mini@nada.kth.se
matthias@nada.kth.se

Reports can be ordered from:
CID, Centre for User Oriented IT Design
Nada, Dept. Computing Science
KTH, Royal Institute of Technology
S-100 44 Stockhom, Sweden
telephone: + 46 8 790 91 00
fax: + 46 8 790 90 99
e-mail: cid@nada.kth.se
URL: http://www.nada.kth.se/cid/

`Conzilla` – Towards a concept browser

Mikael Nilsson
Matthias Palmér

January 28, 2000

Abstract

We have designed the basics of an architecture for working with distributed mind-maps containing connected concepts. This involves

- designing a static representation of concepts and mind-maps for storing and transmitting
- designing an application programming interface (API) for working with concepts and mind-maps
- designing a way to browse an interconnected network of such mind-maps.

The concepts and mind-maps are represented statically in Extensible Markup Language (XML), in a modular way. A general class library written in Java for manipulating concepts and maps has been developed, as well as a browser named "`Conzilla`" using the library.

Our work is meant to be used as part of of a distributed interactive learning environment that is being developed at Center for user-oriented IT-Design (CID), at the Royal Institute of Technology (KTH) in Stockholm, Sweden. It has been supervised by Tekn. Dr. Ambjörn Naeve, who is a senior researcher at CID.

Contents

Acknowledgments

We are most grateful to Ambjörn Naeve, who inspired us to get involved in this risky but rewarding business. He has been a perfect tutor with his great visions but willingness to let our creativity flow freely. He, together with Donald Broady, Hans Melkersson, Bo Westerlund and all the others involved in the Garden of Knowledge project have been a never-ending source of inspiration, ideas and confusion.

We also want to thank Stefan Pålsson at TDB, the Department of Scientific Computing, at Uppsala University for letting us realize our rather wooly ideas. There is no real room for a interdisciplinary subject like ours, but he has shown courage and great interest in our work.

Last, we want to thank Linus Torvalds for giving us Linux, the Blackdown team for giving us Java for Linux, and the whole Linux community and the GNU project for giving us a superior development environment.

Chapter 1

Introduction

There has been a lot of excitement in the last couple of years regarding different forms of computerized learning. There is a general consensus among the proponents of computerized learning that it has the potential to revolutionize the the way we learn and the way we teach. There are, however, many difficult problems to be resolved before everyone can take full advantage of the new medium. One of the most important aspects of the digital age, the Internet, has been largely ignored in the construction of computerized learning material. You could ask why you should be restricted to the material you have available locally, when there is a world of information out there.

This situation is changing rapidly, and new technologies are forming in order to facilitate net-based learning. However, solutions that take advantage of the Internet face major obstacles. The first problem is the diversity of platforms that are in use out there. Being totally platform-independent means limiting the features of the learning material.

There are other approaches. The IMS project, discussed in Section 7.5, is developing a platform-independent *framework* for web-based learning material. It does, however, allow the actual material to be as platform-dependent as necessary.

This thesis work falls in the same category as the IMS project. The goal of our work is not a specific technical solution to the problem of how to teach and how to learn using a computer. In contrast, our work is a way to tie together different technical solutions in a coherent and platform-independent way, a way to organize the knowledge rather than a way to present and use it.

This isn't the place to explain the complete theoretical framework behind this work. See [12] and [14] instead. In this chapter and the following we will concentrate on explaining only those aspect of the framework that are needed to understand what has been done.

1.1 Background

This thesis work is part of a project called the Garden Of Knowledge, which is an interactive learning platform that is being developed at CID at KTH under the coordination of Ambjörn Naeve.

The Garden of Knowledge project is aiming to provide an open-ended and flexible experimental platform which can be exploited in order to gain insights into how to create computer-supported learning environments with an increased potential for individualized learning. The project has resulted in several prototypes, of which the initial three are documented in [11] and [12].

The Garden of Knowledge is being designed and implemented as a so called Knowledge Manifold, which is a conceptual framework for individualized and interactive learning that has been introduced by Naeve in [12]. Here conceptual relationships are described in terms of concept-maps, expressed in Object Modeling Technique (OMT), the predecessor of Unified Modeling Language (UML), which is the main conceptual modeling language of today (see e.g. [4]).

As described in [12], a knowledge manifold is structured as a set of interlinked knowledge patches - each maintained by a dedicated knowledge-gardener. Each patch is equipped with its own set of concept-maps, which are constructed by the corresponding gardener as part of his or her personal presentation of the conceptual world. Such personal knowledge patches do not grow in isolation. On the contrary, they are collectively calibrated in various ways, in an ongoing process that creates an maintains the cultural consensus which we call the objective (= real) world around us.

The contents of a group of related concepts is structured as a so called knowledge component, which contains a description at multiple scales. As discussed in [12] and [14] a well designed knowledge component is somewhat like a skiing area, with different ways to get down, each one marked with a code that indicates the type of challenges that it offers the skier. There is nothing that prohibits you to choose a black pist even if you are at the green level of preparation, but you have a pretty fair idea of what to expect from the experience.

As pointed out in [12], the level of presentational complexity of a knowledge component should be interactively adaptable to the demands of the learner, just as in a computer game. A knowledge component is designed to separate between “what to teach” and “when to learn”, wheras a learning configuration (such as e.g. a traditional course) is designed to anticipate the most effective connections between “what to teach” and “when to learn”.

Within the context of a knowledge manifold, constructing learning configurations is done by what is termed “composition of components”. This activity is closely related to establishing a learning strategy for each learner involved, which

means eliciting individual answers to three basic questions: 1: What am I interested in? 2: What is there to know about it? and 3: What do I want to know about it?

Taken together, the collection of concept-maps define the so called atlas of the knowledge manifold. In [13] and [14] Naeve shows how to make use of this atlas in order to perform what he calls "conceptual navigation" and display the contents of a combination of concepts at multiple scales of resolution and depth.

1.2 The Problem

This thesis work is concerned with the organization of a Knowledge Patch, and with how to present the available knowledge in a Patch in a natural way. In more general terms

> we are interested in the organization of knowledge of any kind, and the presentation of this organization.

We will use special kinds of mind-maps for this purpose, that describe how different concepts within the subject field relate to each other. These mind-maps could simply be a way to show the structure of a Knowledge Patch, i.e., the organization of its Knowledge Components, but more often those maps convey significant knowledge on their own.

A mind-map thus does not in itself include any information to browse, presentations to view or tutorials to engage in etc. It only involves the *organization* of these things. In many learning resources in use today, including online material in the form of HTML pages, the structure of the knowledge is embedded in the document presenting and explaining the concepts. We want to separate the organization of knowledge from the presentation of it.

Knowledge in the form of mind-maps is, in this model, distributed between different and independent Knowledge Patches. This system becomes interesting when we allow Knowledge Patches to use knowledge from each other. Therefore, the representation of the mind-maps must be distributable and platform-independent, while remaining globally interconnected. The act of viewing mind-maps has now transformed into something different which we call *browsing mind-maps*, and which is called conceptual browsing in [14]. The purpose of this thesis work can now be formulated as:

> *Design a distributable way to represent mind-maps in such a way that it becomes possible to browse an interconnected network of them.*

This formulation is very vague. Making it concrete has consisted of three steps. First, to define a static representation of mind-maps, suitable for storage and transmittal, that is sufficiently general for the above to work. This has been the most important part of our work. Second, to design an API to manipulate mind-maps and third, to design a browser that can browse such mind-maps.

1.3 How to read this paper

Chapter 2, *The Meaning of Browsing*, explains in detail what is meant by "browsing". It explains the concepts involved and gives a high level description of the goals we have tried to reach. It is essential reading in order to understand what we have accomplished.

Chapter 3, *Elements of a Solution*, gives an overview of our solution and motivation for the design decisions that we have made, as well as possible alternative solutions. It gives the framework into which the following chapters fit.

Chapter 4, *Representing Concept-maps in XML*, Chapter 5, *The Class Library*, and Chapter 6, `Conzilla`, describes our solution in some detail, including problems that we have not solved.

Chapter 7, *Future extensions*, describes components of an extended type of solution that we have decided to postpone to some future date.

Chapter 2

The Meaning of Browsing

This chapter describes the specifications we have used for the project, and introduces the terms we will use later on. It represents specifications we received from the Garden of Knowledge group at CID and our own clarifications of those, together with results from discussions during the project. The project has been a moving target, where we have participated actively in the formulation of the project. This chapter represents the most recent version of that formulation. Because of the way this specification has evolved, the term *we* is used in this chapter to denote both us and the Garden of Knowledge group at CID.

2.1 A simple example

Suppose you want to know something about bicycles, and have decided to use the system we have designed and a small Knowledge Patch of a friend to find out more. You open the concept browser and in it, you find the diagram of Figure 2.1.

What can be seen in this diagram? You can see the concept “bicycle”, and other concepts related to that concept in different ways. These relations will be described in Section 2.3.2. Apart from just viewing the relations you can see here there are a few things you can do. If you want to find out more about bicycles, you can mark the “bicycle” concept and choose to view information of different kinds regarding bicycles. Or perhaps you want to know more about wheels. One way is to click on the “wheel” concept. You will be presented with a concept-map further analyzing the “wheel” concept.

This is the vague idea that we want to specify in detail in this chapter, and later present our implementation of.

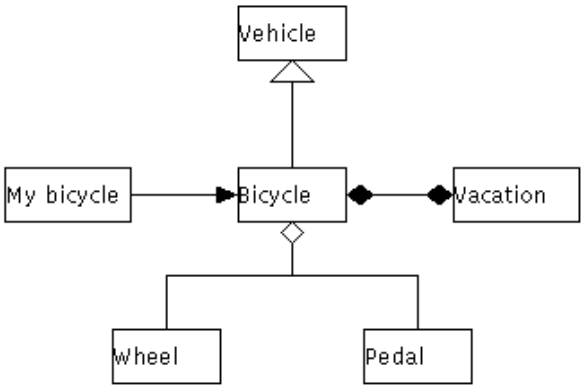

Figure 2.1: A diagram describing the “bicycle” concept.

2.2 The role of UML

The representation of mind-maps that we have used is heavily inspired by UML. UML, designed by the Object Management Group (OMG), is originally designed to be used as a modeling language for object oriented programming languages. It contains specifications for drawing class diagrams describing the relationships between classes, and many other important types of diagrams. The class diagrams have been the most important inspiration for us.

We will not enter a more detailed explanation of UML here, as it is not designed for being used the way we want our mind-maps to be used. It is, however, interesting to observe the differences between what we have designed and UML. These differences will be clearer as we proceed through this chapter,

2.3 Concepts

The most fundamental element of a mind-map is the *concept.* In UML class diagrams, the concepts are usually classes or objects in a programming environment. Our concepts are more general, and just like the regular meaning of the word “concept” they are meant to be used to describe any object of a thought process.

These concepts are meant to be placed in mind-maps. From now on, we will use the term *concept-map,* which is part of the terminology of a Knowledge Manifold, for the special form of mind-maps we are designing. The characteristics of concept-maps will be described later.

The definition of a concept given above does not suffice to design a representation of a concept, of course. We have focused on a few important characteristics of a concept, as described in the context of a Knowledge Manifold (see [12]):

1. A concept has connections to other concepts, which we call *associations*.

2. A concept has different ways to be described and explained, what we call *presentations*. Such a presentation describes a concept from a certain point of view, called an *aspect* of the concept.

3. A concept can refer to a concept-map that analyzes the concept further, which we call a *detailed map*.

A concept is associated with a Knowledge Patch, and is probably seen in the concept-maps defined there. But we want to be able to use the knowledge contained in a concept in other contexts that the original designer may not even be aware of. This leads to several important demands on a concept:

4. The representation of a concept should be platform independent.

5. It must be possible to incorporate a concept into any concept-map, together with concepts from other Knowledge Patches.

6. A concept must be independent of any concept-maps where it is located.

7. A concept must not have any visual attributes such as size, position etc., but only generic visual attributes that guides the presenting application in how to present the concept.

These requirements can be seen as the distributability demands on a concept, and marks the real departure from UML. UML does not deal with the problem of distributing the logical information in a diagram, but only with how to combine it visually.

In contrast, the design taking form here is primarily a logical design. Concepts do not have specific visual attributes, and are not placed in a specific concept-map. Therefore, the role of UML is to provide inspiration for the elements of the logical design of concepts and specifications for the visual design of concept-maps. But inspiration for distributability must be found elsewhere, such as the MOF, described in Section 3.5

2.3.1 Aspects

An aspect of a concept is a way to think about it and deals with certain types of questions regarding the concept. Take the concept “bicycle” as an example. Aspects of this concept could be:

operational: How do I use a bicycle?

historical: How did bicycles develop?

conceptual: What is a bicycle, anyway?

Each aspect has several presentations. A presentation describes that aspect of a concept in some detail. This amounts to a separation of *what* we want to know from *how* we want it presented. Examples of presentations of the operational aspects of the bicycle concept are:

visual: a picture of a person riding a bicycle

textual: a description of how to ride a bicycle

virtual: a 3D virtual bicycle simulator

The different presentations of a concept taken together are called the *content* of the concept. This term will be used extensively in the rest of this report. Content may be in the form of HTML pages, images, movies etc., but may as well involve running an application on the user’s computer. Therefore, we allow content to be as platform-specific as the concept designer finds necessary. The specification for aspects are:

1. Each concept has a number of aspects, preferably of standard types
2. Each such aspect has a number of presentations, preferably of standard types.
3. The description of available aspects and presentations of a concept should be platform independent, although the content itself may not be.

2.3.2 Associations

An association connects concepts. It describes how concepts relate to each other, and may be of many different types. [12] and [14] focus on three types of associations defined by UML that are important for describing the relationship between any types of concepts:

generalization connects two concepts where one of them describes a larger set of objects than the other. Example: “vehicle” is a generalization of “bicycle”.

aggregation connects two concepts where one is a part of the other. Example: "wheel" is a part of the aggregate "bicycle".

classification connects two concepts where one is an instance of the other. Example: "my bicycle" is an instance of the class "bicycle".

These are our predefined associations[1].

What characterizes an association is:

1. An association connects two or more concepts.
2. An association is a special form of concept in the sense that it too may have aspects with presentations and a detailed map.
3. An association has a type, which may be one of the above with predefined semantics, or any other that the designer wants to define the semantics of.

As was the case with concepts, associations also must be distributable. For an association to work in a distributed framework, we demand the following:

4. The representation of an association should be platform independent.
5. An association must be able to connect concepts from different Knowledge Patches.
6. An association must be incorporable into different concept-maps.
7. An association must be independent of any concept-maps where it is located.
8. An association must not have any visual attributes such as position etc., but only generic visual attributes that guide the presenting application in how to present the association.

2.4 Concept-maps

What has been defined above is an abstract world of interconnected but independent concepts distributed between different Knowledge Patches. Nothing has yet been said about how concept-maps fit into this scheme. This will be our next goal.

A concept-map is a presentation of a part of the abstract world of interconnected concepts. A concept-map is designed to emphasize certain associations

[1] For a discussion of why we have chosen these three as the most important, see [12] and [14].

between certain concepts, and this way provide a limited view of how the concepts relate to each other.

A concept-map is therefore necessarily a visual object, very similar to a UML diagram. UML diagrams contain both logical information about concepts, like name, type etc. and visual information like position of concepts and associations etc. In contrast to this, a concept-map should contain all the visual information in the diagram we want to construct, while concepts, associations etc. should contain all the logical information.

To be more precise, a concept-map should contain:

1. A number of concepts.
2. A number of associations between those concepts, but not necessarily all existing associations between them.
3. The visual information necessary to draw a diagram containing the above.

Usually, a concept-map is located in the same Knowledge Patch as the concepts it contains, simply because the designer wants to provide important views of his or her material. Concept-maps are, however, the last and most important step in combining concepts from different Knowledge Patches in a distributed way. Therefore we demand the following:

4. The representation of a concept-map should be platform independent.
5. A concept-map must be able to contain concepts from any Knowledge Patch.
6. It should be possible to generate simple concept-maps automatically from a set of concepts.
7. A concept-map must not contain any logical attributes of the concepts and associations.

2.5 Demands on a browser

It should now be much more clear what the role of a concept browser is. We must put together all the fragments we have specified so far, into a coherent whole. A concept browser should be able to:

1. Present a concept-map, which includes
    - Showing concepts and associations contained in the concept-map.

- Present the logical information contained in concepts in an intuitive way.

2. Present the available aspects of concepts.

3. Possibly provide a way to display the content of concepts. This will not be possible for all types of content anyway, as the content is allowed to be very platform dependent.

4. Allow the user to view the detailed map of a concept. This is the fundamental browsing step.

In addition, a concept browser should be able to keep several conceptmaps easily reachable. This could be concept-maps that are very close in what they describe, and which the user therefore often switches between. Another case is when the concept-maps are part of a course and you want course diagrams in the form of concept-maps to be present along with the concept-maps containing the course material, as described in Section 7.6. Still another case could be as bookmarks for a user.

Chapter 3

Elements of a solution

3.1 Overall structure

We have constructed a browser called `Conzilla` running as a Java applet inside the Netscape browser. This concept browser displays concepts, concept-maps and aspects loaded over the Web. The browser uses a Java API to retrieve and use what we call components. These components, which are our concepts, concept-maps, aspects etc., are structured in the form of XML documents, that combine to form a diagram that can be presented to the user. You browse from one diagram to another via identifiers identifying components. We use Netscape's functionalities as a Web browser to display contents in different formats.

3.2 The concept of a component

As described in Chapter 2, we needed to design a way to represent *concept-maps* containing a number of *concepts* with connecting *associations*. Each concept and association should have several *aspects* of content. We also wanted to be able to represent a collection of several concept-maps. A concept is typically contained in several concept-maps, and therefore a separation of concepts from concept-maps is necessary to avoid duplicated information. Modularization is necessary.

Our solution is to let a *component* in our system be:

1. a coherent entity
2. uniquely identified
3. distributed with respect to other components, but still having the possibility to be connected to them

4. easily created, stored and transferred over networks

Each component should also have *meta-data* describing the constructor of the component and other information as demanded by the IMS Metadata standard [19]

Right now, we have two components dealing with the abstract world of concepts, namely concept and contentdescription. The design of these two components will be fairly stable in the near future. Opposed to these are the components dealing with the visual representation. A 2-dimensional representation of the abstract world of concepts such as a concept-map, is only one visualization. Therefore we allow new components to form.

One desirable component would be a component describing a 3-dimensional view of the concept space, (see [14]). One could also imagine other 2-dimensional maps. These new components of course also needs a more or less new API and some compatible browser.

The components we have designed are of five different types.

3.2.1 Concept

A concept consists of a title, a detailedmap and a number of *aspects* containing contentdescriptions. A concept also contains associations, as well as references to other concepts that contain associations to this concept. Thanks to these references, it is possible to find the related concepts. This is not a problem within a given concept-map where all concepts and associations are known, but in the global conceptual environment it makes it much easier to search for related concepts this way.

3.2.2 ContentDescription

A contentdescription holds information about the MIME type (see [6]) and metadata about the content. A Uniform Resource Locator (URL) says where to find the content. Maybe it would be nice to have this as an Uniform Resource Identifier (URI) instead (see [3]), but right now we are limited by using the Netscape browser as a displayer, so an URL is a must. This component is intended as a preview of the content, so that you can browse without having to engage in content. This way you avoid actually loading a lot of data before you know if it is what you really want.

3.2.3 ConceptMap

A concept-map is a collection of references to concepts and associations. The concept-map further defines the layout of the concepts and associations. Each

concept is given a size and a position. Each association is described via a range of points and each type of association is given visual attributes like line thickness and arrowhead form etc. A concept-map, not the individual concepts, is the starting point for browsing of concepts for the regular user.

3.2.4 MapSet

A map-set is a collection of concept-maps, typically shown in a browser as a set of tabs. A concept-map always has a default map-set, but if this is used or not is up to the browser. A browser can choose to keep some extra maps not defined in the current map-set. These maps could be personal maps created by the user or maps describing the course context etc.

3.2.5 AspectSet

An aspect-set is an administrative component for handling the different aspects and presentation types that a concept-map's concepts can have. Every concept-map has an aspect-set. It defines the aspects and presentations that are interesting within this context. As it is a component of its own, the same aspect-set can be used in several concept-maps. Typically, the same aspect-set is used for a whole subject field.

3.3 Unique identification

Defining a way to access components is a complicated problem with three main parts. The components need a unique identity, but we also need means to locate the components and retrieve them. The identity of a component could typically be unique over the whole web or the local harddisc. The problem is to define these identities in a general way so that they can be resolved in the right way.

The means to locate components depends on how the identifier is constructed. It could point directly to the location of the component, or it could involve a more lengthly investigation via a database or some advanced directory search perhaps using Lightweight Directory Access Protocol (LDAP).

Still the problem of retrieving the component when you've found it remains. Possibilities include HTTP, FTP or indirectly via a downloadable archive such as a Java Archive (JAR) file. Another way is not to retrieve it at all, i.e., to use the remote object via a CORBA-IDL interface or similar.

A solution to these problem should:

1. uniquely identify each component for the browser

2. allow but not demand the identifier to specify how to locate and retrieve the component

The first requirement is satisfied by conforming to the URI standard[1]:

> A Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource.

The second requirement is met by allowing conformance to the URL standard:

> The term "Uniform Resource Locator" (URL) refers to the subset of URI that identify resources via a representation of their primary access mechanism (e.g., their network "location"), rather than identifying the resource by name or by some other attribute(s) of that resource.

Note that the URL standard allows us to specify just the location and not the retrieve method (i.e., allows us to use a CORBA solution even for URLs)

> Although many URL schemes are named after protocols, this does not imply that the only way to access the URL's resource is via the named protocol. Gateways, proxies, caches, and name resolution services might be used to access some resources, independent of the protocol of their origin, and the resolution of some URL may require the use of more than one protocol.

All in all, we want to set up specifications for our type of identifier in such a way that it is always a URI, sometimes a URL and maybe defines a retrieve method. The application itself could be allowed to choose between different locator services and favorite protocols.

In addition, we want to specify some standard ways of component lookup. See Section 7.7 for details.

The current implementation is very primitive although it vaguely resembles an URI. But we strive for full compliance with the standard described in [3].

3.4 Implementation

Now when we know what a component should be, we need to find representations and a suitable programming environment. One trivial but very demanding solution could be to do everything from scratch, starting only with a programming language and primitive socket connections over the Internet. Obviously this would not fit an experimental project like ours, as we would be spending time on the wrong problems. The problem is to find the suitable tools for an implementation.

[1] This quote and the following ones in this paragraph are from [3]

3.4.1 The role of the Web browser

Since content is allowed to be in many different forms it would be a lot of work to implement a content displayer on our own. A content displayer would preferably be able to display images, sound, video, text etc. The most general content displayer is the Web browser.

In addition, much of the available computerized learning material is already accessible via the Web, and it would simplify the development of content immensely if it can be developed directly for the Web. So we have decided that using a Web browser as content displayer solves the problems of displaying content in the most general way.

Since we can't expect an immediate worldwide acceptance of our work, we also need a way to get our own application program to work on different platforms and for most users. This is also solved by somehow letting the application work inside a browser.

In short, the environment that fits our purposes best is provided inside a Web browser (see Section 6.7 for a further discussion of the Web browser). It is, however, important that the overall design is not dependent on browser functionality. If we some day want to develop a specialized content displayer outside of the Web browser environment, we should be able to use components easily in spite of this.

3.4.2 The role of XML

A solution for distributing and storing *concepts* and *concept-maps* is XML. XML is defined by The World Wide Web Consortium (W3C) as a meta-language for markup languages. A component will be represented as a XML-document, with one type of document for each type of component. These XML-documents will be loaded over the Web, parsed into a Document Object Model (DOM) (see Section 5.3), and finally displayed in the correct manner. An application will be responsible for the procedure, running inside or together with a browser. The next generation of browsers (Netscape version 5 and Internet Explorer version 5) will support XML directly as a way of representing arbitrary information. This could be used to retrieve and parse the component and then just let the application access the information. It is questionable if it ever will be possible or even preferable to let a web-browser take over the application's job completely. This could in theory be done by a complicated Extensible Stylesheet Language (XSL) document. XSL is a special form of stylesheet describing the presentation of XML documents. We have not thoroughly investigated whether this will be possible to do or not.

The design goals of XML that affect us are[2]:

[2]Taken from [5]

1 XML shall be straightforwardly usable over the Internet.

4 It shall be easy to write programs which process XML documents.

6 XML documents should be human-legible and reasonably clear.

9 XML documents shall be easy to create.

10 Terseness in XML markup is of minimal importance.

Number 1,4,6 and 9 fit our demands perfectly, while number 10 merits a brief discussion. The drawbacks with a non terse representation includes unnecessary download time and a user-hostile editability. But you seldom want to edit by hand[3] and the largest download time isn't due to concepts and concept-maps but rather depends on the actual *content* in form of images, movies, sounds and such. Although XML documents do not provide the shortest representation of concepts and concept-maps you can think of, we accept this fact since the advantages by far outweigh the disadvantages. We note that other representations of UML diagrams also use XML. See Section 3.5.2 and [18] for a discussion.

3.4.3 The Role Of Java

As discussed above, we want our application to:

1. be platform independent to improve acceptance
2. easily display content using a web browser
3. be extensible and flexible, so that new applications can be developed from our implementation in a simple and straightforward way.

To get an application to work on different platforms together with different browsers without rewriting the code is clearly a job for the Java programming language. The application we have developed can be run both inside a browser and as a stand-alone application remote-controlling a browser, if one exists, to display the content (see Chapter 6 for a discussion).

Java is very useful in other respects as well. Java is a very useful prototyping language for several reasons:

- using an object oriented language focuses the design on the structure rather that on the implementation.
- simple memory management.

[3] See Section 7.2

- abundance of useful libraries.

Java is also very well adjusted for usage together with XML, and there are several good parsers for XML written in Java. The reason for this is amongst other things the good Internet connectivity features, and the good international character set support of Java.

In short, the Java environment fitted us perfectly in every respect.

3.5 A comparison with MOF

The idea of a distributed object model for UML-like diagrams is not new. This Section will compare our design with that of another distibuted modeling system: the MOF.

The Meta Object Facility (MOF) is a proposed standard for meta-models developed by OMG in cooperation with several large software vendors. A meta-model is in essence a modeling language such as UML, and the MOF has a similar scope to that of UML. As described in [16], the main purpose of the OMG MOF is to

> provide a set of CORBA interfaces that can be used to define and manipulate a set of interoperable metamodels.

In practical terms, the MOF is a distributed modeling language, much like what we want to design. The MOFs initial purpose is to be used in object oriented analysis and design (similar to UML), but the OMG expects the MOF to be used to model other information systems. So the question arises: why not use the MOF? This question demands a lengthy discussion.

3.5.1 Problems with MOF

There are several problems with the MOF that makes it problematic to use for our purposes. The most important are described in the following sections.

Modeling objectives

The MOF Specification [16] states that the MOF designers provide

> a balanced model that is neither too simplistic (one that defines only classes, attributes, and associations) nor too ambitious (one that has all object modeling constructs as required in a general purpose modeling language like the UML). The designers have specified this model to be rich enough to define a variety of metamodels and precisely enough to be implemented in CORBA environments.

What they wanted to design was a model that would be immediately usable in object oriented analysis and design. One important part of the MOF is the MOF to CORBA IDL mapping, that makes it possible to automatically generate programming interfaces for objects descibed by the MOF. This means that an object described by the MOF primarily has a programming interface.

Our purpose differs from this in that the models we want to construct are models of any thinkable concepts. Such concepts are often not specific enough to be described as objects in a programming environment. So we actually *want* a more simplistic model that is not intended to be directly usable as a programming construct.

Modeling limitations

The MOF has several serious limitations with respect to their description of associations. Firstly, they only allow associations of degree two (see Section 3.6.2), even though this will change in future versions of the MOF specification. Secondly, and more serious, MOF does not view associations as being very important entities on their own. As described in Section 3.6.3, our design will allow associations to be full-fledged concepts, having all attributes the concepts of today have. We believe this is a fundamental flaw in MOF when it comes to usability in other contexts than object oriented analysis and design.

Distributability

As the MOF design is done in CORBA, the MOF describes a network of interconnected objects. There are serious problems with using this directly as a basis for a project like ours. The main problem is that MOF objects are directly connected to each other. This would imply serious stability problems if this was to be distributed globally. A large global network of interconnected CORBA objects is not yet feasible, even though this may be the case in the future.

The primary use of the MOF is inside a single development environment, called a Repository, and inside this Repository the objects are connected directly. The MOF specification [16] does allow references to other objects in other Repositories, but notes that

> it is recognized that the great majority of these object interactions will remain within one vendor's boundary

which is a position that we simply cannot accept for Knowledge Patches, which must be designed with the primary purpose of being used outside the Patch, interconnected to other Patches.

Our solution is to let the objects reference each other indirectly via identifiers, and to be independently distributed. In spite of this, nothing stops us from designing the objects in CORBA, using the CORBA Naming Service for locating components, and even letting certain CORBA objects be connected directly. But this must not be the primary design philosophy.

Another issue is CORBAs heavy-weight profile. By distributing objects packed in XML documents, we allow lighter operation of the whole system, especially when distributed over the Web.

Prototypability

We needed a simple implementation that would give us the relevant ideas for the future of this project. Implementing the system using MOF would distract us from fundamental design issues that arise when trying to model the mental abstractions of the human mind.

By using our own, much more simplistic implementation, it is possible to give a concrete form to our objectives and how they differ from the objectives of both the MOF and UML.

3.5.2 MOF and XMI

The XML Metadata Interchange (XMI) format, is designed as a serialization of a UML metamodel described using the MOF. This serialization is done using XML. Why isn't this a suitable representation of our objects? The reason is that it shares many of the problems with the MOF, namely:

- object oriented analysis and design fixation
- distributability problems. An XMI document describes all classes and associations contained in one closed model, typically one Repository. This is orthogonal to the modular design we want, and indeed, the goal of XMI as described in [17] is to ease

  > the problem of tool interoperability by providing a flexible and easily parsed information interchange format. In principle, a tool needs only to be able save and load the data it uses in XMI format in order to inter-operate with other XMI capable tools.

  That is, XMI is not designed as a way to decentralize information but as a way to transfer information.

- prototypability. The XMI Specification is truly a large one, and would distract us from more important tasks.

So we have decided not to use XMI. This is also the right place to mention UML Exchange Format (UXF) described in [18] that is a more lightweight XML description of UML diagrams which has inspired us in many ways.

3.5.3 Conclusion

We have concluded that the MOF is not optimal to use for our project. However, the overall tendency towards componentification, object oriented modeling, and information distributability give us hope that there will be a suitable generalization of the MOF available at some time in the future.

What we have designed can be described as a generalization of the MOF in the directions of

- usability outside programming environments, more precisely for modeling human knowledge in general,
- distributability, and
- ability to present parts of a very large model in small diagrams

Viewed in his way, it is obvious that MOF has inspired us in important aspects as a way to represent and generalize UML diagrams.

3.6 Future considerations

The component framework introduced in this chapter has not yet stabilized. This section will explain some of the problems and give some idea of what the components will look like in the future.

The main problem we have with the current implementation is concerned with associations. This problem has several independent parts, and we will now analyze them in turn.

3.6.1 The location of associations

In the current implementation, an association is not a component in itself, but instead it is placed within one of the concepts it connects. This organization has several problems. The most important is that this organization effectively makes it impossible to connect two concepts that you do not have the right to edit. Many

cases are imaginable where you wouldn't even *want* to include the association in the concept, for example when you construct your own personal maps connecting concepts that you are studying (and perhaps your class of 20 pupils are all doing the same).

In the future we want to allow associations to be added to an already existing, and for the creator of the new association uneditable, concept. The solution is to allow the associations to be components of their own instead of being part of the concept component. Actually, the existence of this presumably new externally added association won't be known to everyone else. Only if the association is explicitly included in a concept-map or something similar would the rest of the world know of its existence.

This does not stop the Knowledge Patch designer from wanting the concepts to know which associations they are playing a role in. This simplifies finding the concept's neighborhood enormously (see Section 7.7 for a discussion of searching).

This is achieved in the current implementation by allowing each concept to contain references to other concepts that contain associations to it. When associations are components on their own, this is solved by allowing references to the association directly instead. This is a much more natural design.

3.6.2 The structure of associations

During the course of our work, we have found it necessary to put further demands on associations. This section tries to explain the problem and the reason for our decisions, and point to a better solution of the problem.

Let us define a few terms that are often used when discussing associations[4]:

role is an end of the association. We say that a concept *plays* a role in the association. Each role is of a certain type, and an association may be able to hold roles of several different role types.

degree or **arity** is the number of different role types that an association can have. In the case of a *generalization* there are only two role types: the general and the specific. We can imagine more complicated cases. Take a book loan as an example. There are three role types involved: the person loaning the book, the library and a number of books. So this association has degree three.

multiplicity of a role type is the number of concepts that play roles of that role type in an association. In the book loan example, one person can loan two books at the same time from a library. So the multiplicity of the "book" role type in this book loan is two, even though more books are allowed

[4]The terminology is taken from `CORBA::Relationships`, as described in [15]

The problem

Our implementation limits the degree of an association to be exactly two. There are several reasons for this. First, the three fundamental associations we use are all of degree two. Second, associations of higher degree would introduce much complexity that we did not find necessary when designing the first implementation. Thirdly, higher-degree associations can be imitated by introducing a new concept. In the book loan example, you can introduce the concept "book loan" and construct three associations to the other concepts.

Our implementation allows one of the two ends in an association to have multiplicity higher than one. The reason for this is completely visual. When for example looking at the "wheel" and "pedal" concepts as parts of the "bicycle" concept, it would not be especially intuitive to draw the "part of" association using separate arrows for each of them. Instead we want to group them together so that they have a common end touching the bicycle concept. This behavior is illustrated in Figure 2.1.

Our decisions differ from that of the UML designers. They allow degrees higher than two, but only at the cost of introducing a rhombic "association symbol" for what they call *n-ary* associations, that is located between the concepts. The difference between this and introducing a new concept we find negligible. Interesting to note is that the MOF has only allowed associations of degree two (see Section 3.5.1), something that is also true of UXF (see Section 3.5.2).

Our decisions also differ from those of the `CORBA::RelationShips` designers (see Section 5.2 for a discussion). `CORBA::RelationShips` is, however, designed more for relationships between physical resources than for representing knowledge, but it is an unquestionable fact that their design is more general than ours when it comes to associations.

The solution

The upshot of all these considerations is that our present design of associations is actually not very good. The concept component can certainly not be called coherent with the above discussion in mind. The question is how to solve the problems with the current implementation without introducing new problems.

We have already designed a new implementation that we believe solves all these problems as well as being much more consistent. It is heavily inspired by the `CORBA::RelationShips` interface. We will now turn to describing the new design. The description cannot be very detailed as the new design is not yet implemented, but we believe implementing it is fairly straightforward.

The solution is to let association be a component of its own as described in

Section 3.6.1, and to introduce a new component, the *association-type*. Each association must be of some association-type. The association-type contains information about the degree of the association and the names of the different role types in the association, as well as the allowed multiplicity of each role. For example, the "book loan" association-type has the following attributes:

- Degree: 3
- Roles:
  - loaner; allowed multiplicity: 1
  - library; allowed multiplicity: 1
  - book; allowed multiplicity: 1-∞

The association-type also defines abstract presentation attributes, such as color, line thickness and arrowhead type. The presenting application is free to interpret these attributes in a suitable way, depending on the presentation medium. The concept-map browser will use a presentation that closely resembles UML diagrams.

Every association then contains roles, each being of one of the role types defined in the association-type. The multiplicity must be within the limits set by the association-type. Each role declares which concept it represents and possibly other attributes.

Finally, a concept-map includes the concepts and associations it wants to display and gives them visual attributes, much like today. One important addition is that a concept-map will allow visually grouping together associations of the same type, with one concept playing the same role in all of them as was discussed earlier. All in all, the design results in the same visual appearance in most cases but with a much better architecture and separation between abstract and visual information.

A natural question in this context is if concepts also need a type. This question needs to be examined in more detail.

3.6.3 The essence of associations

Associations and concepts actually share many features. The obviuos similarity is of course that they both have content, as discussed in Section 2.3.2. Visually, they are very dissimilar, but there are acually reasons for making them share features. In UML, an AssociationClass is the type of an object that represents an association. Many interesting relations between objects are often represented using an AssociationClass. Examples include a library application, that introduces the BookLoan

class, which is clearly an AssociationClass between a Customer, a Library and a number of Book objects.

An AssociationClass is a Class (=type) on its own, and similarly we view associations as being concepts on their own. The weakness in the current implementation is that it would be natural to allow associations between associations, such as an instance of the BookLoan class. This is not allowed.

So we have decided to extend our redesign of associations even further. We want to introduce a generalization of both association and concept. It would not be right to call this generalization either concept or association, so we have named it *neuron*[5].

A neuron has the following features:

- it has a type which is essentially identical to the association-type above, but which will be called the *neuron-type.*
- it connects zero or more other neurons.
- it has aspects.
- it has meta-data.

Typical neuron-types are:

- "concept" is a neuron-type of degree zero.
- "generalization" is a neuron-type of degree two (with the obvious roles).
- "attribute" is a neuron-type of degree one.
- "event" is another neuron-type of degree zero.

As was the case with association-types, the neuron-type will give an indication of the presentation of the neuron. This will now include the possibility of presenting association with a "box" like concepts have, and just as AssociationClasses can be presented in UML.

This way we have solved the problem of giving concepts a type, by simply letting them be neurons, and allowing the neuron-type to contain information about the characteristics of that type.

Note that this leads the way towards introducing other kinds of maps than concept-maps, such as activity diagrams (see [4]) with different semantics than concept-maps. This was illustrated by introducing the "event" type above.

[5] A neuron connects other neurons. Our neurons connect to form a globally interconnected neural network, which we visualize using computer tomography (concept-maps)

Chapter 4

Representing Components in XML

4.1 A short introduction to XML

XML is a language describing a class of data-objects called XML documents. How such a XML-document is written is determined by what is called a Document Type Definition (DTD). The structure of a XML-document resembles that of HTML, i.e., it is a markup language, and like HTML is derived from SGML. But HTML is only described by one DTD. In contrast, XML provides a way to define a language like HTML by defining its corresponding DTD. We have designed one DTD for each component, and hence every component can be transported as a separate XML-document.

4.2 The component representation

The components described in Section 3.2 will here be described as XML-documents. Since just listing every component DTD is not very pedagogical (and indeed does not convey all the semantics you need to know) we will present a simple example for each component. The graphical presentation of the examples is shown in Figure 4.1.

4.2.1 The `<MetaData>` tag

The `<MetaData>` tag is contained within every component and presents a collection of meta-data tags and values. Every attribute is contained within the tag `<Tag>` with an attribute `NAME` describing the type of meta-data. This is now a

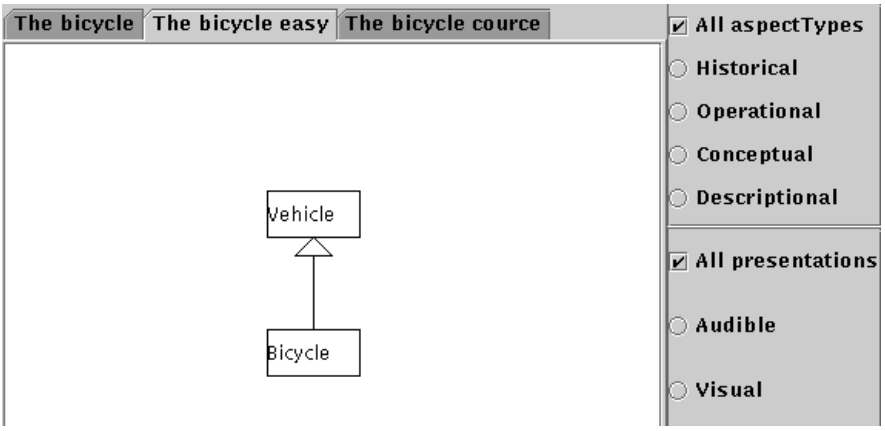

Figure 4.1: The concept, concept-map, map-set and aspect-set examples graphically illustrated. This interface closely resembles the conceptual browsing interface described in [14]

rather loose set of data but will in time conform to the standard defined by the IMS Metadata standard (see 7.5).

The data should clearly specify who created the component, a short description of it, how and in which context it should be used etc. A special case is the content-description component which does not describe itself, but describes a piece of content in some form somewhere.

The `<MetaData>` tag won't be further commented in the examples.

4.2.2 An example concept

This concept represents the mental picture of what a vehicle is. It has been shortened and simplified for presentability here.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE Concept PUBLIC "-//CID//DTD Concept 1.0//EN" "concept.dtd">

  <Concept CONCEPTID="CID:vehicle">
    <MetaData>
        <Tag NAME="Creation_details">Matthias Palmer, 990531</Tag>
        <Tag NAME="Keywords">material | transportation | pollution</Tag>
        <Tag NAME="Description">A human manufactured
                                transportation object.</Tag>
    </MetaData>
    <LinkData>
        <DetailedMap MAPID="CID:vehicletypesmap"\>
        <Aspects>
```

```
        <Aspect TYPE="Conceptual">
              <AspectPresentation TYPE="Visual"
                                  CONTENTID="CID:whatisvehicle">
              <AspectPresentation TYPE="Audible"
                                  CONTENTID="CID:vehiclesounds">
        </Aspect>
      </Aspects>
   </LinkData>
   <Presentation>
     <Title>Vehicle</Title>
   </Presentation>
  <Generalization ASSOCIATIONID="1" ROLE="specific">
    <MultiEnd CONCEPTID="CID:humanobject">
      <Multiplicity HIGHEST="1"/>
    </MultiEnd>
  </Generalization>
  <AssociationRef CONCEPTID="CID:bicycle"/>
 </Concept>
```

The first two rows are of a technical nature and will be ignored. Let us only notice that the second row specifies the DTD describing the XML-document for the concept component.

<Linkdata>

is also optional, containing the concept's different ways of presentation. First a <DetailedMap> is given, where a component ID, CID:vehicletypesmap (a concept-map), is given as an attribute. Second there is several <Aspect-Presentation>s within the <Aspect> tags. The component IDs CID:what-isvehicle and CID:vehiclesounds has the same type conceptual (see Section 2.3.1). But they have different presentations: Visual and Audible.

<Presentation>

is optional and holds a <title>, Vehicle. This is not necessarily the right place for this information. See Chapter 2 for a discussion about separation of abstract and visual information.

<Generalization> and <AssociationRef>

are both associations where <Generalization> is one of the three predefined associations. The attribute ASSOCIATIONID is a unique ID for the association within the concept scope. When several associations are placed within a single concept, this is necessary to distinguish associations located in the same concept

from each other, for instance in concept-maps. According to the discussion in Section 3.6.2, an association has a `<SingleEnd>` and several `<MultiEnd>`s. The optional tag `<SingleEnd>` has been left out but all `<MultiEnd>`s are necessary. The `<Multiplicity>` tag describes the multiplicity of the concept in each end. There is two integer attributes `HIGHEST` and `LOWEST`, both optional. See Section 3.2.1 in order to understand the phenomenon. `<AssociationRef>`.

4.2.3 An example concept-map

```
<ConceptMap MAPID="CID:bicyclemap">
  <MetaData>
    <Tag NAME="Title">Bicycle Overview</Tag>
    <Tag NAME="Author">Matthias Palmer</Tag>
    <Tag NAME="Date">28, may, 1999</Tag>
    <Tag NAME="Version">0.1</Tag>
  </MetaData>
  <MapSet MAPSETID="CID:cycledemo"/>
  <AspectSet ASPECTSETID="CID:materialworld"/>
  <BoundingBox WIDTH="400" HEIGHT="400"/>
  <ConceptStyle CONCEPTID="CID:bicycle">
    <BoundingBox WIDTH="60" HEIGHT="30"/>
    <Position X="170" Y="215" RELATIVE="lowleft"/>

    <AssociationStyle ASSOCIATIONID="1">
      <SingleEndStyle>
        <Line TYPE="polygon">
          <Position X="200" Y="155"/>
          <Position X="200" Y="185"/>
        </Line>
      </SingleEndStyle>
      <MultiEndStyle CONCEPTID="CID:vehicle">
        <Line TYPE="polygon">
          <Position X="200" Y="155"/>
          <Position X="200" Y="125"/>
        </Line>
      </MultiEndStyle>
    </AssociationStyle>
  </ConceptStyle>
  <ConceptStyle CONCEPTID="CID:vehicle">
    <BoundingBox WIDTH="60" HEIGHT="30"/>
    <Position X="170" Y="125" RELATIVE="lowleft"/>
  </ConceptStyle>
  <VisibleConcept CONCEPTID="CID:bicycle"/>
</ConceptMap>
```

<MapSet> and <AspectSet>

refers to a map-set component and an aspect-set component, respectively. See Sections 3.2.4 and 3.2.5 respectively.

<BoundingBox>

is the size of the concept-map given in pixels.

<ConceptStyle>

is a number of tags describing the graphical representation of a concept and its associations. A <BoundingBox> and a <Position> gives the size and position of the concept, which will be displayed as a rectangle. In the future, concepts and associations will have several possible visualizations, possibly following UML-standards.

In the example the association is between the vehicle and bicycle concepts. Since the association lies within the bicycle concept it makes sense to have the <AssociationStyle> within the bicycle <ConceptStyle>. The <AssociationStyle> contains a <SingleEnd> and one or several <MultiEnd>s describing the ends of the association. In this case the association is a generalization where vehicle is at the general end and bicycle at the specific end. Of course there can be several generalizations or some other association from the bicycle concept. To include styles for them, you only need to identify which association you want to describe via the ASSOCIATIONID, (see Section 4.2.2).

An association is described by several <Line>s, one for each role. Each such line contains <Position>s. The last <Position> is the one ending at the concept playing that role. The attribute TYPE defines how the line should be drawn, polygon is just simple straight lines between the <Position>s. Another type could be spline, a smooth interpolating curve algorithm. The three built-in types of associations use arrow heads as specified by UML.

4.2.4 An example map-set

```
<MapSet MAPSETID="CID:cycledemo">
      <Map TITLE="The bicycle course" MAPID="CID:bicyclecourse"/>
      <Map TITLE="The bicycle" MAPID="CID:bicycleoverview"/>
      <Map TITLE="The bicycle easy" MAPID="CID:bicyclemap"/>
</MapSet>
```

This component is easy to understand. The <MapSet> just contains several <Map>s, each containing a title and an optional icon belonging to the concept-

map identified by the component ID given in the MAPID attribute. See Sections 2.5 and 3.2.4.In the picture 4.1 we see <MapSet> as a set of tabs on top of the concept-map. The concept-map choosen is CID:bicyclemap.

4.2.5 An example aspect-set

```
<AspectSet ASPECTSETID="CID:materialworld">
  <AspectType NAME="Descriptional"/>
  <AspectType NAME="Conceptual"/>
  <AspectType NAME="Operational"/>
  <AspectType NAME="Historical"/>

  <PresentationType NAME="Visual"/>
  <PresentationType NAME="Audible"/>
</AspectSet>
```

Both <AspectType> and <PresentationType> are explained in Section 3.2.5.

4.2.6 An example contentdescription

```
<ContentDescription CONTENTID="CID:bicycleparts" MIMETYPE="text/html"
 URL="http://www-lexikon.nada.kth.se/skolverket/bilder/teman/tema18.JPG">
     <MetaData>
        <Tag NAME="Language">Swedish</Tag>
        <Tag NAME="Content">En cykel och andra fordons beståndsdelar</Tag>
        <Tag NAME="TargetGroup">Schoolchildren,
                        students in the swedish language</Tag>
        <Tag NAME="TargetAge">any</Tag>
        <Tag NAME="TargetMentalAge">curious child</Tag>
     </MetaData>
</ContentDescription>
```

Before you load the content of a concept you can examine a <Content-Description> where the attributes URL and MIMETYPE describes where to get and how to treat the actual content. The <MetaData> consists of relevant and searchable descriptions of the content.

Chapter 5

The class library

In order to use the XML components described in the previous chapter, we wanted to design an API for accessing components that we could use when developing a concept browser. The API could be usable in other contexts than the browser, though, and it is important that the API is sufficiently general to enable the construction of any kind of browser. Therefore we have formulated a few design goals that have led us through the design process. They are presented here.

This chapter can be read as an introduction to using the library, but it is not meant as a reference manual. We refer the interested reader to the source code for that purpose.

5.1 Design goals

The overall design of the class library is meant to reflect the component idea introduced in Section 3.2. A natural design is to let each component be represented by a Java object. This is essentially what we have done. But there are important differences between a static representation as described in Chapter 4 and a dynamic representation in an object oriented programming language. This API must be practical to use directly in an application, and therefore it must hide many of the details. For example, it would be practical if one could traverse the relations between concepts as a graph directly, i.e., replacing the reference via ID to other concepts, with an object reference in Java. This and similar considerations demand some consideration.

It is obvious that there must be an strong correspondence between the elements of the static representation and the structure, names etc. in the Java representation. A typical scenario would be that the objects are constructed from the static representation, and possibly later saved into the static representation. But it is also

possible that components are represented in some other way, such as some unknown form of database storage or a more dynamic approach like CORBA, where the objects already reside on remote computers. Therefore the Java representation must not be too heavily dependent on the XML representation.

These considerations has lead to a few important design goals:

1. The API should not be dependent on XML or on the actual XML representation of components. It should be possible to add other static or dynamic forms of representations.

2. The API should be easy to use in an application.

3. The API should not contain any graphics code or interactive code. It should be possible to implement any type of user interface using the API.

These goals have been mostly fulfilled.

When it comes to how the API should deal with the different components, it is important that the API mirrors the design philosophy behind the component framework. In particular, it is important that the idea of a completely non-visual global conceptual environment is reflected in the API. Therefore, the API should consist of two layers, where one deals only with the abstract concept information without using a concept-map, while the other takes care of providing access to concept-maps. Ideally, the API should be able to use the same concept in several concept-maps at the same time, to enable different forms of presenting concept-maps. This is not the case with our API at this time, even though the separation between the two layers have, in principle, been carried out successfully.

5.2 `CORBA::RelationsShips`

`CORBA::RelationShips`, described in [15], is a CORBA interface package containing a standardized API for accessing objects and their relationships, that in some ways resembles our configuration of related concepts. It is, however, designed more for relationships between physical resources than for representing knowledge, and shares the distributablility problems of MOF, as described in Section 3.5.1.

We decided that this first prototype was a too early implementations to focus on a CORBA solution, but the package has been an important source of inspiration for the design of the API. `CORBA::RelationShips` is worth looking into when designing a CORBA solution, as discussed in Section 7.4.

5.3 XML and Java

XML is very well adjusted to work in a Java environment, as described in Section 3.4.3. There are several ways to use XML in Java. One way is to use the standard DOM as defined by W3C, and a DOM-capable XML parser. The DOM is intended to be able to describe any type of XML document containing arbitrary markup. However, our decision to use our browser in a web environment has led to demands of simplicity and smallness.

A DOM parser constructs a very complicated and heavyweight object representation, and is in itself complicated. This DOM representation is much more general than we need, though, as our DTDs do not allow any free-text markup.

We have therefore decided to use a simpler XML parser called Ælfred, which is freely available (see [1]). It is extremely small and simple even though it is complete, and using it we have developed a very simple document model that fits our very simple DTDs, but would not fit for example an HTML DTD.

5.4 Packages

We will now try to describe the purpose of the different Java packages that we have implemented, in order for the reader to understand how the API is structured. In accordance with the Java Language Specification (see [9]), the name of the root package of all our packages is constructed from the name of the organization which has developed it. The root package is called `se.kth.cid.kt.Conzilla`.

5.4.1 The concept-related packages

This section describes the packages that are part of the abstract conceptual layer of the API. These are for the most part independent of the concept-map part of the API.

`Concept`

This is the central package, containing classes that represent the abstract information contained in concepts and associations. The types in this package are all interfaces, meaning that there is no implementation of the types in this package.

`Concept.Xml`

This package contains the implementation of the interfaces defined in the `Concept` package. The implementation uses XML for loading concepts.

`Content`

This package contains the representation of content. As described in Section 3.2.2, we use a component called contentdescription for describing content. This package implements this component.

`Content.Xml`

This package contains classes for loading objects defined in `Content` from XML documents.

5.4.2 The concept-map-related packages

These packages contain the implementation of concept-maps. They build on the concept-related packages to connect concepts with their visual attributes.

`ConceptMap`

This package contains classes for dealing with the information in a concept-map component. It does not deal with the visual information of each concept, but only descriptive information belonging to the concept-map, such as the map-set and the aspect-set.

`ConceptMap.Xml`

This package contains classes for loading objects defined in `ConceptMap` from XML.

`ConceptStyle`

This package contains classes that implement the visual attributes of concepts and associations in a concept-map.

`ConceptStyle.Xml`

This package contains classes for loading objects defined in `ConceptStyle` from XML.

5.4.3 Miscellaneous packages

These are packages that do not really belong to any of the above layers.

`Component`

This package contains the implementation of the unique identifiers discussed in Section 3.3, including classes for locating and downloading components. It also contains classes that implement other features common to all types of component. At this time, this is limited to a representation of meta-data.

`Component.Xml`

This package contains classes for loading objects defined in `Component` from XML documents.

`XmlLoader`

This package contains our interface to the Aelfred XML parser to simplify loading and interpreting XML documents. The design hides Aelfred-specific details of the parsing so that the library can easily be adjusted to use any XML parser.

`XmlLoader.DTD`

This package consists of classes containing pre-loaded versions of the DTDs we use for the components. The motivation behind this is that having to load a DTD across the net every time an XML document is loaded is an unnecessary waste of time. As we have decided to put default values and fixed values of certain attributes in our DTD, actually having access to the DTD is important.

`util`

This package contains various utility classes we have found necessary to develop, but which are not specific for concept browsing.

5.5 The future of the class library

The class library will probably change in many ways before it stabilizes. These are the main issues to be solved in the near and more distant future:

- It will adjust to changes in the XML DTDs and the component framework.
- Better separation between the abstract concept layer and the concept-map layer.
- Extending the API to allow editing and storing components. See 7.2.

- It will probably be adjusted to Java 1.2 when there is better support for 1.2 in browsers. This includes several things:
  - The `Collection` framework.
  - The 1.2 security model.

  and more.
- CORBA interfaces for the central objects. See Section 7.4.
- JavaBeans conformance.

Chapter 6

Conzilla

6.1 Background

We have developed the Conzilla browser in parallel and as a source of inspiration for the different components and the class library. From the beginning it has been clear that our browser is just one of many possible applications displaying concepts and associations from different approaches. Therefore the browser is primarily intended to prototype some of the possibilities provided by a concept browser.

Our approach has been straightforward and has hopefully resulted in a very natural concept-map browser, that displays human-generated concept-maps. Automatically generated maps is a possibility left to the future.

6.2 The Graphical User Interface

To write a Graphical User Interface (GUI) doesn't have to be such a mess as it often turns out to be. It heavily depends on the graphichal package you choose, as a well structured starting point can save you from many grey hairs. In the Java environment there isn't much of a choice: either you use the standard AWT-package[1] which is rather primitive, or you use the more recently developed JFC Swing package[2]. We've choosen to work with the JFC Swing package, even though it's not a stable version yet, as it is comparably well designed. It also solves the problem of keeping the design platform independent, as graphical components (buttons etc.) are drawn by the package, not delivered by the operating system.

[1] Abstract Window Toolkit is a standard part of the Java Development Kit

[2] Swing is part of the Java Foundation Classes which is a standard extension of Java

We've choosen to divide the Conzilla browser into several parts, or better, modules. These modules should be as independent of each other as possible. Visually they often occupy a certain area of the screen. A module can invoke actions on other modules by a well defined set of functions (what we call an interface[3]). A superstructure is responsible for all the modules, something that includes creation, layout and communication. The superstructure is simply called the *Browser*.

We've implemented two modules so far: the *ConceptMapBrowser* and the *AspectDisplayer*. A third module is ready for implementation, the *ContentDisplayer*. Even though the names of the modules are choosen to be intuitive, let's have a short description of each one of these three and where they are located in Figure 6.1

6.2.1 Introducing the ConceptMapBrowser

This module is the core of the Conzilla browser and therefore occupies the largest part. In Figure 6.1 the ConceptMapBrowser shows a concept-map, which is actually the same concept-map about a bicycle as discussed in Section 2.1.

On top of the concept-map three tabs are visible. These correspond to the mapset component.

6.2.2 Introducing the AspectDisplayer

The AspectDisplayer module to the right of the ConceptMapBrowser contains two areas of buttons: a list and on the bottom a toolbar. The buttons inside these areas are defined by the aspect-set component, hence, when a new concept-map is loaded the buttons will change or reappear. The button 'show' at the bottom is the connection to the ContentDisplayer.

6.2.3 Introducing the ContentDisplayer

At present the ContentDisplayer module is simply a Netscape frame. See the discussion in Section 3.4.1. To make it a real module we just need to wrap it so that it can be replaced by something else than a Netscape window for some types of content.

6.2.4 The responsibility of the Browser

Since the modules have well defined interfaces, it is possible to lift out some functionality from them. This functionality is placed in the Browser. Right now it

[3]This exactly corresponds to the Java programing concept named `interface`

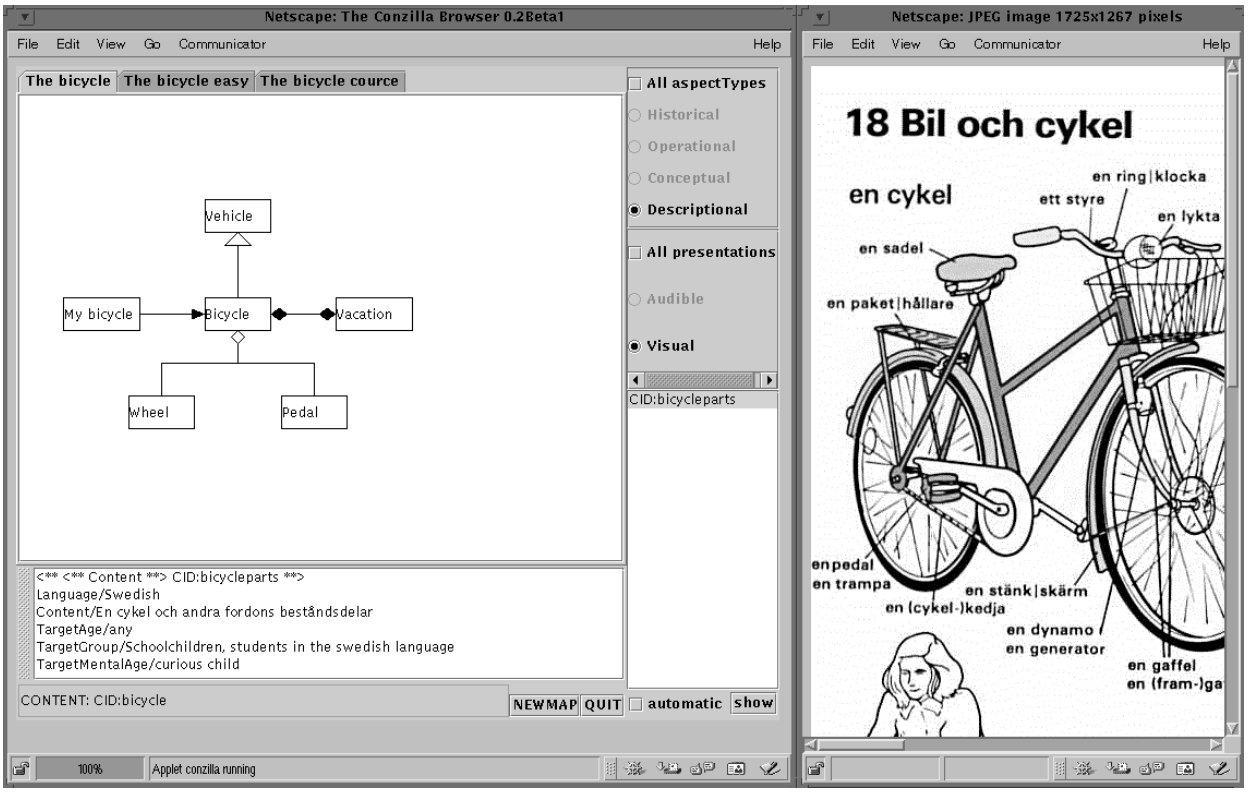

Figure 6.1: A snapshot of the browser (running inside Netscape) displaying content in a separate window. The sample content displaying a bicycle and it's parts is found on http://www-lexikon.nada.kth.se/skolverket/bilder/teman/tema18.JPG and belongs to the swedish-english online dictonary provided by the swedish national school-organization, Skolverket. It is copyrighted SÖ-83 Lidman Information.

handles the meta-data window and the status bar, as well as some buttons controlling the ConceptMapBrowser module.

The meta-data window is placed below the ConceptMapBrowser. Here all forms of meta-data are displayed. Since all components have meta-data there is a lot of different information that can be diplayed here. The first row is automatically generated and indicates whose meta-data is displayed.

6.3 The browser superstructure

As mentioned above Conzilla consists of three modules communicating with each other through a superstructure. The advantage of this is that each module can be exchanged. That is, you can write a new AspectDisplayer without even touching the implementation of the ConceptMapBrowser.

To make this possible, you need a toplevel object, handling the different parts of the browser. This toplevel is provided by the browser class, which is a class with primitives for setting, getting and creating[4] the modules. Whenever the ConceptMapBrowser needs to impose an action on AspectDisplayer it does so by asking the browser class for a reference to AspectDisplayer and then calls it directly.

The Browser is written as an applet, which means that it can be run within web browsers over the Internet. This doesn't prevent it from being used as a standalone application, but then the ContentDisplayer will have to exist or `Conzilla` will be crippled.

6.4 The ConceptMapBrowser module

The interface defines the functionality demanded for every ConceptMapBrowser we construct. The functionality is of three different types: the first is concerned with how to browse through concept-maps, the second with how to work with history listeners and the third with how to access internal states.

6.4.1 Browsing concept-maps

The interface defines the following four functions:

jump to a specified arbitrary concept-map. This automatically displays the new map-set found in the new concept-map. If the new concept-map can't be found, an error messaage is displayed, otherwise a 'JUMP' history event is triggered.

[4] In terms of design patterns, it's an abstract factory (see [8])

zoomIn on a specified concept in the current concept-map gives a new concept-map with corresponding map-set, just like a jump. If the concept-map is not found, an error message is displayed, otherwise a 'ZOOMIN' history event is triggered.

zoomOut is not implemented yet, but it is intended to take place on a concept-map level. Triggers a ZOOMOUT history event.

refresh is just a visual refresh which happens when stuff for some reason gets out of date and needs a repaint. This wouldn't be needed if there were no bugs in the Swing package.

6.4.2 History listeners

The browsing process emits a lot of history events, which can be intercepted by history listeners. The interesting functions are:

addHistoryListener just adds a HistoryListener.

removeHistoryListener removes a HistoryListener if it exists.

fireHistoryEvent emits a specified HistoryEvent to all Listeners.

6.4.3 Accessing internal state

The browser has a current concept-map and map-set, which are changed by the browsing procedure. Another module of the Conzilla Browser (typically an editor) could be interested in getting hold of what is currently displayed. Right now the following functions exist:

getCurrentConceptMap returns the current concept-map.

getCurrentMapSet returns the current map-set.

It's possible that further functionality should be demanded from a concept-map browser and therefore here be accessible via functions. Typically this could be a set of maps specific for the application and/or the user. This would work like a second map-set which, in contrast to the first map-set, would not change when browsing.

6.5 The AspectDisplayer

Here we also have an interface describing the basic functionality. The filtering procedure is rather complicated and needs a thorough understanding of a concept's aspects (see Section 2.3.1 and 3.2.5).

setAspectSet initalizes the AspectDisplayer from an aspect-set component. This means constructing the right button representing aspect and presentation types.

showAspects loads in the aspects from a concept. They are displayed in a list according to the filtering by aspect and presentation types.

setFilter sets aspect and presentation type filter and updates which contentdescriptions that matches this filter. If no aspect or presentation type is given, the show-all option is choosen.

setSelected selects one of several possible contentdescriptions. Note that the selection is performed from a given list and defined to be unique, i.e., two entries can't be selected at once.

isAutomaticShowContent returns true or false depending on whether the ContentDisplayer is automatically updated whenever a new selection is done. Default is false.

setAutomaticShowContent sets the isAutomaticShowContent option.

showSelected shows the content belonging to the currently selected contentdescription. If the selection is empty nothing happens.

6.6 Remote control

Let us investigate a use case: a knowledge patch with information concerning the famous Swedish writer August Strindberg. A book is probably represented by a concept, and this concept can probaly be seen in several concept-maps. One possible form of content of such a concept could be the book in digital format.

Let us suppose that in this book we have several phenomena that can be associated to other books or other resources, such as commentaries. Such a resource can be seen in another concept-map and constitutes in itself the content of some concept. When we browse using regular hyperlinks to such a resource in the diplayed text (all in the ContentDisplayer) it would be rather natural to have the

ConceptMapBrowser change to the relevant concept-map automatically or when you press a 'focus' or 'synchronize' button.

This is actually possible since both ConceptMapBrowser and AspectDisplayer can be externally controlled via the functionality presented above. Some simple JavaScript code inside an HTML-page can invoke actions on our applet if it can only reach it. So remote controlling is possible whenever you can find your remote.

This of course raises many possibilites, that cannot be discussed here.

6.7 Platform and security considerations

What has been described in this chapter gives an impressive picture of the possibilities of using a Java application from inside of a browser. This Section will show another side of the story by briefly explaining some of the problems we have had with this solution.

6.7.1 Java and security

As our application runs as an applet inside the browser, it is subject to applet security restrictions. These involve[5]:

> In general, applets loaded over the net are prevented from reading and writing files on the client file system, and from making network connections except to the originating host.
>
> In addition, applets loaded over the net are prevented from starting other programs on the client.

This means that our browser is not allowed to fetch components from anywhere on the Internet, only from the host where the applet is located. In addition, an editor used inside the browser would not be able to store the XML documents on the local harddisk. There are several imaginable solutions to this problem:

1. Implement a proxy on the host where the applet is located, that can fetch components that the applet asks for.
2. Let the user run the applet from the local harddisk. Applets loaded this way are usually seen as trusted, and have no security restrictions.

None of these solutions are really satisfactory. The first will decrease speed and place a possibly heavy load on the server, while the second will mean a lot of hazzle for the user. We wanted to avoid that, and let the applet do the work all by itself.

[5] taken from [7]

The two browsers of interest both have their own solutions to this problem. Both depend on the idea of digitally *signing* applets, and letting the user grant the applet privileges based on this signature.

Microsoft Internet Explorer uses a solution where the applet must be stored in a CAB archive and signed. This solution is totally tied to the Windows platform, and has therefore been avoided.

The Netscape browser uses a solution where the applet is stored in a JAR file and signed. The applet may then ask the user to grant it additional privileges. This is the solution we have used. Fortunately, it is possible to use this solution in parallel to a Microsoft solution, as the impact on the code is minimal. This will possibly be done in the future.

Hopefully, as browsers start to support Java 1.2, they will also start to support the 1.2 security model, which is also based on the idea of signing. Then we will have a less platform-dependent solution. But that time is not yet here.

6.7.2 Java and Swing

As described in Section 6.2, we have used Swing to construct the GUI. Sadly, this limits us to newer versions of the Netscape browser (exact version depending on the platform, but 4.5 should work) that include version 1.1.5 or later of the Java Development Kit.

So, our solution is not as platform independent as it might seem. We have hopes that future browsers will support the Open Java Integration API, which allows the Java virtual machine to be separated from the browser. Then it will probably be much easier for everyone to get hold of a compatible Java implementation.

6.8 Future

The Conzilla user interface is just a prototype, and has not been been reviewed by either graphical designers or by real users. This means that our focus here is largely the technical design. Here are some possible changes:

- Small design details such as changing buttons, styles, centering text etc.
- Increase the modularisation. Here is a list of a number of suggestions of new modules. Some of these will be looked upon as basic modules and will be demanded as standard.
  - Add the module ContentDisplayer to the superstructure. As mentioned above, it's planned for but not implemented yet.

  - Add an editor module. See Section 7.2 for a discussion.
  - Add a HistoryListener module.
  - Add tools as a module. Tools could be a better interface to much of the functionality, and new modules should be able to add their own tools.
  - Add preferences as a module. New modules should also be able to add their own preferences configuration.

- Remove some functionality of the superstructure so that it only controls the layout and organization of the modules. The creation of new modules, i.e., the factory, has to be done somewhere else. A factory module is of course the solution. For instance, suppose you want to build a new application with a different HistoryListener. Then you will only have to build a new factory and override some layout functions in the superstructure and everything will work together as planned.

With growing stability of metadata we would like to add functionality for choosing between contentdescriptions using such data. In the present implementation the aspectdisplayer filters out a couple of contentdescriptions and then automatically selects the first. This list isn't sorted so it is an arbitrary choice.

Instead it would be nice if the selection was done by some smart algorithm taking as arguments the users personality, the current learning objective etc. This could be done by a decisionmaker where you put in the options, some hints and ask for a decision. The hard part will be to implement an algorithm that is relevant in all situations. Therefore, as you probably guess, we want it to be a module which then of cource can be easily replaced depending on the browsing situation.

Moreover, it could be interesting to use the decisionmaker to choose between concept or other components as well, but this would happen in another sort of browsing, for instance when performing searching (see Section 7.7).

Chapter 7

Future Extensions

We have already described some of the directions along which this work will develop in the future. The discussion has been limited to adjusting and enhancing the existing functionality, but now we turn instead to a number of future extensions of this project that are large projects in themselves.

These extensions are ideas that have grown out of the design work and research we have done, and that have influenced the final design in subtle ways. Therefore we find it important to discuss them here. They are not listed in order of importance.

7.1 User interface design

As has been described in this thesis, there is a lot of functionality hidden in the concept representation. It is an interesting problem to try to present all relevant functionality to the user. Our prototype described in Chapter 6 presents much functionality, but being a prototype it is not perfect from a usability perspective. Once this prototype exists, however, it is feasible to involve graphic designers and real users in the design of the browser.

Usability is extremely important, as the whole idea behind the project is to design an environment sufficiently friendly to have a pedagogical advantage over traditional learning. Therefore the technology must be advanced enough to be transparent. Achieving this goal requires sophisticated user interface design.

7.2 Editor

One of the more important future projects is to design a concept and concept-map editor. The purpose of an editor is to create new and edit existing concepts

and concept-maps. It would probably use XML for storage, and it is therefore important that the design of the XML representation stabilizes a bit before that project is started.

As noted in Section 5.5, the current version of the class library does not implement any functionality for editing concept-maps. This is the first part of constructing an editor.

The second step would be to create the editor. One of the more intriguing possibilities we have examined is to integrate the editor in the browser. Besides reusing code and user interfaces, this would open up completely new possibilities. It would allow the user that browses ready-made concept-maps to construct personal concept-maps explaining in subjective ways the material under study. These personal maps can be shared with friends or be used as part of a course. Some of these possibilities are already under consideration for other projects at CID.

In short, the editor is an important and interesting project.

7.3 Dynamic concept-maps

One interesting area that has been discussed is different ways of making concept-maps more interactive. There are several ways in which this may be possible:

- allow the user to move certain concepts inside the map, to increase readbility
- allow the user to hide parts of the concept-map temporariliy

These possibilities becomes much more important if we allow the user to view automatically generated concept-maps, such as the result of finding all instances of a certain concept (see Section 7.7), and showing the result as a concept-map, or even included in the current concept-map.

7.4 CORBA

As described in Section 5.4, the `Concept` package contains interfaces for the concept representation, which are implemented using XML loading of concepts. One interesting project would be to examine the possibility to implement concepts in CORBA. What would this mean?

CORBA is a standard for distributing objects and allowing them to work together in an application. This fits our situation very well. Our components are nothing less than distributed objects. But with the current implementation, the objects need to be downloaded in XML format and used entirely locally. With

CORBA we would never need to download the component. Instead we would execute the component's methods directly on the component where it is located.

Introducing this feature would probably result in a redesign of important parts of the Java API. Hopefully it would not affect the browser at all.

7.5 The IMS Project

The IMS project is an organization for setting technical standards within the world of Internet-based learning resources. This includes standards for meta-data, user profiles, integration with enterprise systems and much more.

This raises interesting possibilities for concept browsing. As the IMS project has not yet produced many of the important standards that they intend to, it is difficult to say exactly what needs to be done in this matter.

However, it is clear that it will be important that the browser fits into a future IMS environment. This includes communicating with other tools, messaging, using user profiles, course management (see Section 7.6) and much more. Therefore, examining the possibilities of IMS conformance and integrating the browser with the IMS environment will be a highly interesting project in the not so distant future.

We have already started adjusting to IMS standards. All our components are intended to conform to the IMS Metadata standard, as described in [19], which was released during the course of this thesis work. This is not enforced at this point, but our intention is full compliance.

7.6 Courses

The browser is not intended to be used for planlessly browsing concept-maps. The intention is to build a course framework upon the concept-map design. The idea, as described in [12],[13] and [14] is to separate the following roles in a learning environment:

1. The Knowledge Patch designer that constructs a Knowledge Patch with concept-maps, concepts and content, intended to be used in many different contexts

2. The course designer, that essentially constructs a path through a number of concept-maps,

3. The teacher, assisting the student, answering questions etc.

4. The student

What we have designed is essentially the basics of the first item. Designing a way to construct courses through our material is a very important project that can actually be started at any time, as the basic design of the Knowledge Patches is finished. Just as concept-maps can be based on UML activity diagrams; which state the necessary sequential requirements between the different parts of the cource. This technique for modeling courses has been introduced in [14]

Such a cource-modeling project probably would need to investigate some of the connections to the IMS project, as described in Section 7.5.

7.7 Searching

When browsing concept-maps, it can happen from time to time that one wants to find all concepts that relate to a certain given concept. For example, it could be interesting to find all specializations of the concept “vehicle” in order to examine which types of vehicles there are. These concepts could then be displayed in an automatically generated concept-map. Or perhaps you want to find all instances of the “Author” concept. One intriguing possibility is to allow concepts and associations to act as filters for this search, so that one can use the concept “Swedish” to look for Swedish authors. This must be examined in more detail.

Another scenario could be that one wants to search for concepts based on their meta-data. All in all, searching for concepts is an essential part of a complete concept browser. Searching is however complicated by the fact that it needs nontrivial support from the Knowledge Patch server. Everything we have designed so far only involves finding files, which can be implemented on the server side using a simple web server. Searching involves generating documents based on the request, and therefore is much more complicated.

Another aspect of searching is locating components. The simple system we have implemented uses static URL paths for locating and retrieving components. In the future, one would probably want to ask a directory server where to find a component. This would be even more interesting if a CORBA solution is implemented, as one could then search for a component and then choose between several ways to retrieve it, either as an XML document or via CORBA.

In short, searching for components needs to implemented somehow. This would include standardizing the server side of concept browsing. Probably several protocols will exist in parallel, but some form of overall design is necessary. And most of all, as the searching phenomenon has not been thoroughly investigated, there is a need for a prototype to experiment with. This is also a project ready to start at any time.

7.8 JavaBeans

There has also been some discussions of simplifying the work for the Knowledge Patch designer and for the course designer by making the whole Java implementation JavaBeans compatible (see [10]). This would mean that it would be possible to reuse the different Java objects in new environments such as a highly sophisticated Knowledge Patch construction environment, which the editor discussed in Section 7.2 probably will never be and was not intended to be.

As this seems to be farther into the future, the details of such a project are still vague at this point.

7.9 Conclusion

The above extensions are all goals that have emerged during the course of our work. It has become more and more obvious that an existing prototype to play with is an irreplaceable source of inspiration. Many of the ideas in this chapter would not have existed without our implementation. Therefore, we believe that these projects extending our work will bring equally intriguing insights into what can be done with this kind of learning environment, and that constructing prototypes are of fundamental importance, even if they do not survive.

We do hope that this upward spiral will continue.

Appendix A

Glossary

A.1 Special terms

These are some of the terms used in this paper in a sometimes non-intuitive way.

aspect A way to describe a *concept.* We make a distinction between *what* we want to present and *how* to present it. We might want to present historical aspects of the concept or how to use the concept, and any of those can be presented visually (e.g. a picture) or via a verbal description, etc.

association A connection between two *concepts.* It has two ends, and sometimes a direction. Three types are predefined, but new types can be defined locally. It has *meta-data* and *aspects,* so that it acts in many ways just like a concept.

component An independent part of a *concept-map.* Every component has a unique identifier that is used to locate it. *Aspect-sets, map-sets, concepts, content-descriptions* and *concept-maps* are all components.

concept In everyday life, *concept* is used to denote anything that can be the object of a thought process. For us, a concept has *associations* to other concepts, *meta-data* to describe what it is and descriptions or presentations in the form of *aspects.*

concept-map To simplify the visualization of *Concepts* and their *associations,* we've created the conceptmap *component* for a two dimensional layout.

content A *concept* is a representation of a mental object. Further on this concept needs an explanation and the actual explanation is an instance of type content. This could typically be a document, a video, sound or something else

that has learning potential and which is possible to give a reference to. We are at the moment limited to URLs though.

meta-data This is a group of data each with a describing tag, typically it indicates who, when, why, a short description etc. It will be used for searching, classifying, marking ownership etc.

map-set This is a *component* that collects a set of *conceptmaps.* It's meant to group together alternative layouts, related explanations, parts of something too big to display on one conceptmap or just conceptmaps that someone decided belong together.

A.2 Abbreviations

API (Application Programming Interface) – a set of classes and functions for performing a certain task, while hiding the underlying details.

AWT (Abstract Windowing Toolkit) – Java GUI components implemented using platform-specific code, providing functionality common to all platforms.

CID (Centre for user-oriented IT design) – an inter-diciplinary competence centre at KTH. Activities include a three-part collaboration between IT-industry, user organizations and university researchers. For more information, see http://www.nada.kth.se/cid.

CAB (Cabinet Format) a compressed file format developed by Microsoft and used in many of their applications. Internet Explorer, for example, can use CAB to download large Java applets from the Internet.

CORBA (Common Object Request Broker Architecture) – an open distributed object computing infrastructure being standardized by the OMG. CORBA automates many common network programming tasks such as object registration, location, and activation etc. See the OMG home page at http://www.omg.org.

DOM (Document Object Model) – from the DOM spec [2]: "a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents." Also, "the goal of the DOM specification is to define a programmatic interface for XML and HTML".

DTD (Document Type Definition) – a text file that defines the allowed elements of a markup language such as HTML or XML-based markup languages.

FTP (File Transfer Protocol) – a very common method of transferring one or more files from one computer to another

GUI (Graphical User Interface) – the interface to an application that the user sees, and which uses graphical elements such as buttons and menus for interaction.

HTML (HyperText Markup Language) – a language used to describe WWW pages so that font size and color, hypertext links, nice backgrounds, graphics, and positioning can be specified and maintained.

HTTP (HyperText Transfer Protocol) – the underlying system whereby Web documents are transferred over the Internet.

IMS (Instructional Management Systems) – a global coalition of academic, commercial and government organizations, working together to define the Internet architecture for learning. IMS abbreviates Instructional Management Systems, which they have noted raises more questions than answers. So they prefer to be called just IMS. For more information, see http://www.imsproject.org.

JAR (Java Archive) – a compressed bundle of Java classes, similar to ZIP files. It is used to distribute Java applications and their related files.

JFC (Java Foundation Classes) – a set of Java GUI components extending the original AWT. The most important part is called Swing.

LDAP (Lightweight Directory Access Protocol) – a simple protocol that allows you to access and search all forms of directories, containing information such as names, phone numbers, and addresses, over the Internet.

MOF (Meta Object Facility) – as described in the MOF Spec [16]:

> The main purpose of the OMG MOF is to provide a set of CORBA interfaces that can be used to define and manipulate a set of interoperable metamodels. The MOF is a key building block in the construction of CORBA-based distributed development environments.

OJI (Open Java Integration) – an API that allows a Web browser to use any Java Virtual Machine installed on the local harddisk instead of a built-in Virtual Machine.

OMG (Object Management Group) – as described on the OMG home page (http://www.omg.org), an organization that was formed

to create a component-based software marketplace by hastening the introduction of standardized object software. The organization's charter includes the establishment of industry guidelines and detailed object management specifications to provide a common framework for application development.

OMT (Object Modeling Technique) – the predecessor of UML.

SGML (Standard Generalized Markup Language) – a standard for describing markup languages. Used to define both XML and HTML.

UML (Unified Modeling Language) – a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. It has been developed by the OMG. See [4].

URI (Uniform Resource Identifier) – as described in [3], a URI is "a compact string of characters for identifying an abstract or physical resource."

URL (Uniform Resource Locator) – as described in [3], the term URL refers to "the subset of URI that identify resources via a representation of their primary access mechanism (e.g., their network "location"), rather than identifying the resource by name or by some other attribute(s) of that resource"

UXF (UML Exchange Format) – a set of XML DTDs that describe UML diagrams. See [18].

W3C (The World Wide Web Consortium) – as described on the W3C home page (http://www.w3.org), the W3C was founded to "lead the World Wide Web to its full potential by developing common protocols that promote its evolution and ensure its interoperability."

XMI (XML Metadata Interchange) – a serialization of a UML metamodel described using the MOF. The serialization is done using XML. See [17].

XML (Extensible Markup Language) – a metalanguage defined by W3C, used to define specialized markup languages (like HTML) that can be used to transmit data in a portable way. See [5].

XSL (Extensible Stylesheet Language) – a language defined by the W3C used to transform XML documents into HTML or some other presentation format, for display in i.e. a Web browser.

Bibliography

[1] *Ælfred XML Parser*, http://www.microstar.com/aelfred.html

[2] Apparao V., Byrne S., Champion M., Isaacs S., Jacobs I., Le Hors A., Nicol G., Robie J., Sutor R., Wilson C., Wood L., *Document Object Model (DOM) Level 1 Specification* (REC-DOM-Level-1-19981001), http://www.w3.org/TR/REC-DOM-Level-1/.

[3] Berners-Lee T., Fielding R., Masinter L., *Uniform Resource Identifiers (URI)* (RFC 2396).

[4] Booch, G., Jacobson, I., Rumbaugh, J., *The Unified Modeling Language – A Reference Manual*, Addison Wesley Longman Inc., 1999.

[5] Bray T., Paoli J., Sperberg-McQueen C. M., *Extensible Markup Language (XML) 1.0 Specification* (REC-xml-19980210), http://www.w3.org/TR/1998/REC-xml-19980210.

[6] Freed N., Borenstein N., *Multipurpose Internet Mail Extensions (MIME) Part One* (RFC 2045).

[7] *Frequently Asked Questions - Java Security*, http://java.sun.com/sfaq/.

[8] Gamma E., Helm R., Johnson R., Vlissides J.,*Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing 1995

[9] Gosling J., Joy B., L. Guy, Steele Jr., *The Java Language Specification* (ISBN: 0201634511), Addison-Wesley Pub Co, 1996.

[10] Hamilton G. (editor), *JavaBeans API Specification* (version 1.01), http://java.sun.com/beans/docs/spec.html.

[11] Linde, R., Naeve, A., Olausson, K, Skantz, K, Westerlund, B., Winberg, F., Åsvärn, K, *Kunskapens Trädgård,* Centre for user-oriented IT-Design (CID-18), TRITA-NA-D9708, KTH, Stockholm, Sweden, 1997.

[12] Naeve, A., *The Garden of Knowledge as a Knowledge Manifold – A Conceptual Framework for Computer Supported Subjective Education,* Centre for user-oriented IT-Design (CID-17), TRITA-NA-D9708, KTH, Stockholm, Sweden, 1997.

[13] Naeve, A., *Den IT-baserade Utbildningsevolutionen på KTH - några tidigare och några pågående projekt,* Centre for user-oriented IT-Design (CID-51), TRITA-NA-D9909, KTH, Stockholm, Sweden, 1999.

[14] Naeve, A., *Conceptual Navigation and Multiple Scale Narration in a Knowledge Manifold,* Centre for user-oriented IT-Design (CID-52), TRITA-NA-D9910, KTH, Stockholm, Sweden, 1999.

[15] The Object Management Group, *CORBAServices: Common Object Services Specification* (formal/98-12-09), http://www.omg.org/library/csindx.html.

[16] The Object Management Group, *Meta Object Facility (MOF) Specification* (ad/97-08-14), http://www.omg.org.

[17] The Object Management Group, *XML Metadata Interchange (XMI)* (ad/98-10-05), http://www.omg.org.

[18] Suzuki J., Yamamoto Y., *Making UML models exchangeable over the Internet with XML: UXF approach,* 1998, http://www.yy.cs.keio.ac.jp/˜suzuki/project/uxf/.

[19] Wason, Thomas D., *IMS Meta-Data* (draft), http://www.imsproject.org/work_public/meta-data_did188.html.