The Conzilla Design

The definitive reference

Mikael Nilsson

The Conzilla Design: The definitive reference
by Mikael Nilsson

Beta 2 Edition

Table of Contents

Preface

1. Conzilla

Conzilla[1] is the first incarnation of the concept browser idea as developed by the Garden of Knowledge project at Centre for User Oriented IT-design, CID.

CID is an interdisciplinary research group and academy-industry collaboration centre, which is located at the department of Numerical Analysis and Computing Science (NADA) at KTH, Stockholm. At CID, there is a unique blend of knowledge from fields such as pedagogy, computer science, mathematics, art and design as well as cognitive and behavioral science. CID is engaged in a multitude of projects that aim to develop Information and Communication Technology based tools of various kinds.

Today, the activities at CID are divided into three major areas of research and development called respectively Smart Things and Environments, Digital Worlds and Interactive Learning Environments. The initial project within the latter field was the Garden of Knowledge (GoK) project, which was initiated by Ambjörn Naeve in 1996.

The Garden of Knowledge is the name of a multi-mediated learning-tool project which aims to develop IT-supported methods to create an interdisciplinary understanding of the perceptual and conceptual world, and to develop an educational framework for the creation of interactive and individualized forms of learning experiences.

Conzilla is the first prototype of the idea of a "concept browser", which is currently under continuous development at CID. In short, a concept browser tries to present the conceptual relationships between a set of concepts in the form of surfable "concept maps", and to allow the surfer to, separately, engage in content describing these concepts. The details and the merits of this idea are thoroughly described in [cid52], and there is no room here to further discuss this.

2. This article

This article is the definitive reference to 1.0 version of the design of the Conzilla browser for anyone interested in the functionality of the system. The content of this article has three main goals:

- To give a description of the technical functionality and architecture of the Conzilla browser, in

1. This section was adapted from [cid52].

as much detail as possible without entering into implementation details, philosophical discussions or user interface issues. This includes defining all important interfaces and standards as well as describing the level of fulfillance of the design goals. The purpose is to make it possible even for outsiders to understand the design and this way be able to contribute to the design discussion and development without being programmers.

- To give important examples and explanations of uses that have influenced and motivated the design.
- To give a picture of the design development. This involves presenting previous designs and why they were abandoned, as well as alternative designs that have been considered. As well, the goal is to describe the known problems of the present design and ideas for future designs that were in our minds while building the product. This way, it is much easier to understand the present design and very incomplete design ideas that have influenced it. It is also an important starting point for discussions of future design goals.

Thus, it is intended to be non-technical in the sense of not being a programming reference, and non-philosophical in the sense of not trying to motivate in other ways than technical and functional. Still, it is extremely useful as a starting point for any programmer interested in the project, and indispensable for anyone interested in how we manage to fulfill the design goals and what sort of beast Conzilla really is. It is, however, not a user manual; in fact, most users will be very happy not to know the details presented here.

Please note that this article is not just a passive piece of text. It takes an active part of the Conzilla project as an example of the usage of the system. The technical design and functionality is described in Part IV, Technical design of the article..

3. Acknowledgements

I would like to thank the project initiator, Ambjörn Naeve, for the support he has given me during the writing of this article, and for making it possible at all. His high philosophical demands has driven not only the process of contructing this work, but also the whole design it describes.

An important part has of course been played by Matthias Palmér, the co-designer of most of what is described here, and main designer of some. His comments have been worth gold.

I also want to thank CID and the people involved in the Garden of Knowledge project there: Bosse Westerlund, Hans Melkersson and all the others, who work intensely to give the project the cross-disciplinary form it is intended to have.

Lastly, I want to thank Linus Torvalds for giving us Linux, as well as the whole free software community and the GNU project for letting me use a superior SGML/XML authoring environment.

I. The database design

Chapter 1. Database structure

1.1. Scope

The database structure is the core of the Conzilla design. It contains the definitions of the fundamental objects that make up the raw material upon which concept maps are built, i.e., the concepts and relations, as well as the concept maps themselves. It also contains the ideas for how to identify and locate these. The details of the design is presented separately, even though the basic design goals are discussed here.

1.2. Functional requirements

The design goals we have developed for this structure are mostly motivated in [cid17] and [cid52]. The ideas described there have resulted in the following fundamental demands on the design:

- It must be able to describe mind-maps that strongly resemble Unified Modeling Language (UML) diagrams. UML, designed by the Object Modeling 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, which correspond to our concepts, and many other important types of diagrams. The class diagrams have been the most important inspiration for us, as they directly correspond to concept maps, but also activity diagrams are of fundamental importance, as they are able to describe the sequencing demands of, e.g., a course.
- It must contain all the information necessary to fully describe the concepts and their relations, and the concept maps in which they are located. This structure will be used in automated situations not unlike a relational database, such as searching for concepts having this or that relation to a given concept, or collecting lists of concepts related to a certain area. It is therefore necessary that no information is given only implicitly (via visual cues, for example).
- A good separation between the logical relations between concepts on the one hand, and the presentation of them in concept maps on the other, is a corner stone in the design. This is necessary not only to allow concepts to be present in several concept maps, but also to allow other types of concept maps, for example three-dimensional. The idea is that concepts form a complex interconnected network, of which concept maps present different views.
- It must be independent of the representation of the data one uses, be it XML, CORBA or something else, and therefore also independent of the particular concept browser in use.
- The concepts and concept maps must be readily locatable, separately, be it locally on your

computer, over a LAN or over the Internet. This is not as trivial as it sounds. It means that concepts on your local database server must be able to have relations with concepts located anywhere on the Internet. This is necessary to be able to, e.g., build your own concept maps using already existing concepts. Such functionality demands a well defined identification system.

- It must allow the introduction of content into the system, which includes linking concepts to content, and representing filters for this content to be able to filter it into different aspects.
- It must label the elements of the system according to a standardized metadata scheme. This is actually much more important than one may initially believe, as this information gives us the possibilities to search amongst concepts not only by contextual criteria (such as being related to a certain concept), but also by author, description, keywords etc..

1.3. Architecture

The system in many ways resembles a certain type of virtual, in the sense of distributed and disconnected, relational database. It consists of four types of components: neurons, neuron types, concept maps and content descriptions. Each instance of these types is a separate object in the virtual database, each with a separate explicit identity. Each component also has metadata information in the form of an IMS Metadata record, described in [imsmetadata].

The rest of the architecture is described in Chapter 2, Component Identity, Chapter 3, Neurons and Neuron Types, Chapter 4, Content, Chapter 6, Filters and Chapter 5, Concept maps.

1.4. Design Discussion

1.4.1. Distributability

It has yet to be proven that the wanted design is actually implementable in a realistically efficient and clean fashion. That is, it has been implemented, but not tested on a large scale. What could cause problems is that we actually implement a distributed relational database, which very well may cause problems (not least with performance) we were not aware of when designing the system. In particular, the search functions have not been satisfyingly studied.

The requirements described for the design places strong distributability demands on the concepts, and therefore marks a real departure from UML. UML, even though it deals with the logical information in diagrams, does not deal with the problem of distributing the logical information, 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.

1.4.2. The MOF

The Meta Object Facility (MOF) is a proposed standard for meta-models developed by the 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 [mofspec], 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. There are several problems with the MOF that makes it problematic to use for our purposes, the most important of which are described in the following sections.

1.4.2.1. Modeling objectives

The MOF Specification [mofspec] 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, which makes it possible to automatically generate programming interfaces for objects described by the MOF. This means that an object described by the MOF is intended to have a well-defined 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 with well-defined methods in a programming environment. So we actually want a more simplistic model that is not intended to be directly usable as a programming construct.

1.4.2.2. Modeling limitations

The MOF has several serious limitations with respect to their description of associations. Firstly, they only allow associations of degree two (see further Section 3.3), 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 Chapter 3, Neurons and Neuron Types, our design allows 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, where associations usually are of a simple character.

1.4.2.3. 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 [mofspec] 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, maybe even being MOF objects. But this must not be the primary design philosophy.

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

1.4.2.4. 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 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.

1.4.2.5. Conclusion

Thus, 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 the modeling of 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.

Additionally, nothing should prevent us from interacting with the MOF. Importing MOF models should be relatively straightforward and could probably be made on-the-fly using CORBA wrapper objects or XML exporters. Even importing parts of the concept world into a MOF model should be feasible using, for example, an XMI exporter. See the discussion in Section 8.4.

1.4.2.6. CORBA::RelationShips

`CORBA::RelationShips` is another standard interesting as inspiration. Being designed by the OMG and described in [corbarelationships], it 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 has several advantages over the MOF, the most important of which are:

- No object oriented analysis and design fixation. Indeed, `CORBA::RelationShips` has been used in an early attempt by IMS to describe learning resources (which is very close to our objective).
- No modeling limitations. Relations of any degree can be described, and relations are given a primordial role in the system.
- Relatively easy implementation thanks to its more simplistic approach.

Why is is not used by us? There are two important reasons:

- It is designed more for relationships between physical resources than for representing knowledge. With this package, you describe the relations between objects external to the system, which is a motive that differs slightly from our primary motive, which is to describe relations between objects inside the system.
- It suffers from the same distributability problems as the MOF, thanks to its connected CORBA object nature.

But `CORBA::RelationShips` is, in fact, not at all far from the design we want. It has been a direct influence for the present design when it comes to how to represent relations. What we have designed is in principle the design of `CORBA::RelationShips` with three important changes:

- Making the system independent from CORBA in order to facilitate the wished level of distributability.
- Making the system more self-contained, in the sense that the relations are between objects inside the system.
- Adding data to the nodes in the system.

It is obvious that it will be an important inspiration for the design of the CORBA binding.

Chapter 2. Component Identity

2.1. Scope

The definition of the identifier of the components defines much of the capabilities of the database structure. It not only acts as a unique label for the components, locally and globally, but also may affect the way the component is located and used. In fact, the definition of the identifier defines much of the infrastructure of the system.

2.2. Functional requirements

The scenario that has inspired the design of the identifier is one where the Conzilla user is using components from several places simultaneously. Realistic locations include: local harddisc, a database on the company LAN (the MOF case), and different sources on the Internet. One could imagine that the company database actually exports several components for use over the Internet, by different users using different browsers. This is already enough to describe several important characteristics of the identifiers:

- One component must have only one identity, i.e., if there are several methods of accessing the component, the identifier cannot depend on the method used. Consider a component located on the company database, that can be accessed both from inside the LAN and over the Internet. It is probable that you could want to access the database directly when inside the LAN, but use a web server for outside access. Still, other components referring to the component must use the same identity, as they do not know if they will be used inside or outside the LAN.
- The format of the identifier must be useful in several different environments, which means that it cannot depend on, e.g., the programming language.
- As the example above shows, some sort of resolving mechanism may be necessary for certain components. We really do not want to build this mechanism into the specification, so we instead want the identifier to allow future expansions in this direction.

2.3. Architecture

The identity is a Uniform Resource Identifier (URI). The protocols currently supported are:

`http:`

The object is downloaded in XML format over HTTP. The identifier format is well known.

`file:`

The object is located in the local filesystem in XML format. The identifier format is well known.

`urn:path:`

This is the preferred type of identifier, conforming to the Path URN Specification draft [pathurnspec]. It has the structure `urn:path:/path/component`, where `path` is an abstract path to the component, and `component` the name of the component[1].

The URI is translated using a resolving mechanism into a URI list, as described in [pathurnspec]. These URIs are recursively tried until a URL in one of the forms `http:` or `file:` given above is found[2].

The resolver maps paths into base URIs, whereto the component name is appended. The path may match partially, and several paths of different lengths may therefore match. In this case, the longest (most specific) match will be tried first. The non-mathing part is appended to the base URI before the component name.

As an extension to the Path URN specification, the resolver may give the data format that components under the given base URI uses, which it describes with a MIME type. This enables the browser to choose the preferred data format, based on for example the domain of the server (if local, use direct access to database etc.). If no format is given, the format implied depends on the URI protocol used.

Thus, if your resolver maps the path `/math/geometry` to `http://www.nada.kth.se/cid/geomcomponents`, which uses the XML format (and MIME type `text/xml`), the URI `urn:path:/math/geometry/euclidean/circ` will become `http://www.nada.kth.se/cid/geomcomponents/euclidean/circle` which can be immediately downloaded.

These identifiers can be used from anywhere to refer to a certain component, with the only reservations being that certain URIs are relative to your current position (typically, `file:` URIs), and are therefore intrinsically local.

---

1. This way, the system indeed becomes a Knowledge Pathwork.

2. Until other protocols are supported, that is. Notably, other protocols will be needed to access CORBA components.

The resolving mechanism for Path URNs is currently a table with all known path names. This table is loaded upon startup of the browser.

2.4. Design Discussion

The choice of URI as the identifier was made very early in the design process, and has never really been questioned. It is based on the following characteristics of URIs:

- The existing URL protocols immediately usable: http, ftp, file etc., as well as the probability of useful future protocols. Especially in mind are the specifications for identifying single files inside JAR archives with a URI, which makes it simpler to download whole archives of components at once, as well as definitions for identification over IOCP (used in CORBA) with the help of URIs. URIs, and not least URLs, has been proved useful and functional.
- The existing standardized encoding in ASCII text, making it universally usable, as well as the universal support in different programming environments. A not small part of this point is that users actually are used to this type of identifiers. Note that the encoding of special characters in ASCII is not yet implemented.
- The hierarchical structure of the standard URI protocols. This is what makes the idea of a "base URI" work; it is a simple matter of appending one string to another.
- The possibility of defining your own protocols. In fact, URIs have exactly the flexibility wanted: it allows everything from completely abstract identifiers (such as URNs) to identifiers such as URLs which specify both location of the object and the protocol of access.

We see no real need for support for the FTP protocol, as this is mostly intended for large file transfer, which is not our case. Perhaps it could be interesting to use for saving components.

The `urn:path:` protocol is used in order to be able to specify the location of an object without having to specify the method of access, i.e., data format and physical location. Such a protocol is absolutely necessary as seen in the discussion in Section 2.2.

The table resolver mechanism must be replaced, as the idea of everyone having a complete table of paths is absurd. The idea is to replace the table mechanism with, e.g., an LDAP resolver, or even the DNS extension used in the Path URN specification [pathurnspec]. Such a resolver would, for a path, return a base URI and the data format to use, just as the table does today. This resolving represents a considerable overhead if it has to be done for each component downloaded (cf. DNS when browsing the web), which is why only paths are queried, not individual identifiers. This allows us, namely, to cache the resolved paths.

In short, we believe that we are on the right track using Path URNs, even though there are decisions left to take. There has gone a lot of thinking into the design, and we are satisfied with the resulting system.

In fact, we did not originally use Path URNs, but a home brewn protocol. After having experimented with this protocol and enhanced it to fulfill our needs, we recognized that we had mostly redesigned the Path URN protocol. So we decided to switch to this possible future standard.

Note that relative URIs in concept maps and neurons are supported.

Chapter 3. Neurons and Neuron Types

3.1. Scope

The neuron and neuron type objects define what we mean by a concept and a relation, and how we have chosen to describe them, and is therefore the part of the Conzilla design that makes it a concept browser. Neurons and neuron types constitute the logical network of concepts and relations, not tied to a certain visual presentation.

3.2. Functional requirements

The question of how to represent concepts and their relations is a difficult one, and much of the design work has therefore gone into figuring out for what, exactly, we want to use them. In the end, these are the important design goals we have considered:

- Concepts and associations must be incorporable in any concept maps, i.e., must be independent of any specific concept map. New associations must be able to connect concepts located anywhere, i.e., it must not be necessary to change the associated concepts. This is to allow reuse of components on a global basis.
- The concepts and associations must be typed. For associations, the motivation is clear: To be able to search for concepts having a certain relation to a given concept, the types of the associations must be known. For concepts, it is not that clear. In fact, the motivation is that one may want to have other types of objects represented in the system, such as events or states (to be able to draw activity diagrams), and these objects are not strictly concepts.
- Associations must be able to associate any number of concepts. This is motivated by the existence in UML of n-ary associations and in other relational systems such as `CORBA::RelationShips` of similar constructs. In reality, this reflects the need to represent relations such as a book loan, which has three parties: the loaner, the library and the book.
- Objects in the system should be equipped with a facility for adding arbitrary information to them, in order to represent, e.g., a date or a programming class (with a list of member functions) in the system.
- Association should preferably be able to relate not only concepts, but also associations to one another.

3.3. Architecture

The design consists of two types of components: neurons and neuron types. To understand the terms used here, let us define a few terms that are often used when discussing associations (The terminology is taken from `CORBA::Relationships`, as described in [corbarelationships]):

role

is an end of the association. We say that a concept "plays a role" in the association. Each role is of a certain role 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 a role in an association can be of. 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. The allowed multiplicity of the book role type may, however, be much larger, and maybe even unlimited.

3.3.1. Neuron

Neurons represent both associations and concepts, as well as other types of objects needed in other types of diagrams, such as events or states. A neuron has the following attributes:

- A type, in the form of a reference to (i.e., the identifier of) a neuron type.
- An IMS meta-data record as described in [imsmetadata] (as all database components have).
- A number of "data" tag-value pairs. Data allows concepts and associations to actually contain information that are inherent to their type. For example, a neuron of the type `historical event` may be equipped with an historical date. This is what makes the database we are designing actually contain data.

- A number of "axons", where each axons points to another neuron via its identifier. An axon has a type, and is allowed to have its own set of data tag-value pairs. They corresponds to roles for associations. For example, neurons that represent aggregations have axons of the types `part` and `aggregate` representing the two role types in this association type. Each axon is given an unique identifier within the neuron.

Further, the meta-data of a neuron may contain the following kinds of `relation`s that are treated specially:

content

Points to a component that represent a piece of content for this neuron. See Chapter 4, Content

filter

Points to a filter to be used to sort the content of this neuron.

context

Points to a concept map that describes a context of this neuron.

3.3.2. Neuron types

Neuron types defines the allowed values of the different attributes of a neuron, and therefore is what distinguishes concepts from associations and other sorts of neurons. A neuron type also contains hints as to how objects of the type are to be presented visually. The logical attributes are:

- An IMS meta-data record as described in [imsmetadata] (as all database components have).
- An enumeration of the allowed data tags.
- An enumeration of the allowed axon types, each with their own allowed multiplicity (as an interval between zero and infinity), as well as their allowed data tags.

A neuron is not allowed to use other data tags or axons types than these declared in its type. This is partially enforced in the current implementation; not, however, on the database level, but only when creating or modifying neurons.

The visual attributes are grounded on the assumption that the presentation of a neuron consists of a cell body and the axons. A concept will not have any axons, and is therefore represented only by the body, while associations in general have no visible body, but may very well have if they are important objects in themselves. The attributes in the current implementation are:

- Attributes for the axons: a thickness between 1 and 10 as well as a line type, such as `continuous`, `broken`, etc., as well as attributes describing how the end of the axons, the head, should look, described by a head type, e.g., `arrow`, `diamond`, as well as a size parameter between 1 and 10. It is also possible to specify if the head should be filled or not.
- Attributes for the body: a box type, which is a string describing the shape of the body, such as `rectangle`, `ellipse`, etc. There is also the possibility of connecting the body with the axons with the help of a line, which has the same attributes as the axon lines above.

The application is free to interpret these characteristics in the way it finds suitable for the presentation medium. The head, box and line types given above are examples only, as no formal specification of the allowed values has yet been produced.

3.4. Design Discussion

3.4.1. Terminology

The motivation behind the terminology "neuron" is the following: As what we have designed is a generalization of both concepts and associations, and indeed may be used to represent even other things, neither of these names would be appropriate. A neuron is an entity which connects other neurons with each other using axons[1], and thus serves both as link and as the subject of links. Our neurons connect to form a globally interconnected "neural network", which we visualize using computer tomography sections, that we call concept-maps (the name of which, however, may not be entirely in line with the metaphor).

This is a very abstract image, but this is what makes it possible to describe the different types of relations we want to describe. It is, we believe, an interesting new way of representing graphs that, in effect, comes down to assigning an owner to each edge in the graph among the two nodes it is connecting[2].

Note that when we say "neuron", we mean any neuron in the system, while the word "concept" is only used to talk about neurons that represent concepts or similar objects.

1. And dendrites, of course. But all analogies have their problems...

2. As it still is the case that a single axon connects two and only two neurons.

3.4.2. History

It has been a long way before we have reached the current design. Our first attempt at creating the structure, carefully described in [cid53] lead to failure on several points:

- We were fixated on the ideas of concept and association, and therefore failed too see how we were to expand the system to include other types of object to be able to draw, for example, activity diagrams. This also led to an underestimation of the importance of associations. Remember what Henri Poincaré said:

  > The aim of science is not things themselves - as the dogmatists in their simplicity imagine - but the relations between things. Outside those relations there is no reality knowable.

- Associations were not separate components, but were instead each included in a concept, leading to severe problems and complexity when wanting to associate external concepts.
- Associations were only allowed to be of degree exactly two, which was motivated by the fact that the most common associations were of degree two, as well as by the fact that the MOF actually had the same restriction (which we now consider a severe defect in the MOF). In fact, higher degree association can be simulated by introducing a new concept acting as middle point. However, the current solution results in a much cleaner and more natural system.
- Associations had no type, resulting in problems when attaching common appearances to the same sort of associations, as well as problems of standardization when wanting to introduce new sorts of associations.

All of these problems were clear to us even before we had finished the implementation of the first Conzilla prototype, and the solution described here was already on the drawing board. By contrast, we strongly believe in the current system design. The system has proved itself to work very well in unexpected circumstances.

3.4.3. Concepts and associations

The elimination of the fundamental difference between a concept and an association actually has philosophical consequences that are important. It succeeds in elevating associations to the same status as have concepts. But it also has technical consequences, in the sense that everything you can do with a concept, you can automatically do with an association. For example, when implementing the assigning of content to concepts to aid in explaining them, this becomes an automatic feature of associations as well, which is not insignificant! This allows, namely, the explanation of associations by the same means, which results in exactly the emphasis on the relations between things that we were searching.

3.4.4. Designing types

It is expected that map designers in different areas construct their own neuron types for use in their own situations, but it is also expected that there will be a considerable amount of work dedicated to the standardization of types, as this is what allows large-scale searching. Regrettably, we have not ourselves done much work in this direction, and it is clear to us that before this design is set in stone, we will have to try to encode knowledge from several disciplines in the system so as to be sure that it is actually possible and that the system is flexible enough to cover all relevant needs.

When considering which axon types a new neuron type should have, the idea is that axons connect neurons with the neurons that are essential for its existence. As an example, a generalization does not exist before you have something to generalize and something to specialize, which is why it has degree two, while a concept exists by itself and therefore has degree zero. The same reasoning holds for data as well: it represents information without which the neuron does not exist. The difference between data and the axons is that the information given by a data tag is not represented by another neuron (meta-data, on the other hand, is not information internal to the neuron, but external information).

The reason for having data tags in axons is to avoid having to construct an extra layer of neurons between the neuron and the neurons it links to. For example, you may consider representing a sequence of events by letting them be attached to an `event-sequence` neuron. Then you realize that being connected by axons from this neuron prohibits you from marking the events with the time when they happened, and so you may consider adding the time information to the axons (as it is impossible to add it to the events themselves; they take part in several different sequences). The alternative would be lifting the relation to neuron status, by defining a `event-occurrence` neuron type, connected to the event with an axon and with the time it occurred as data. Then the event-sequence neuron may connect to this neuron instead. It is not at all clear which solution is the right one for this case, as this in fact depends on if you will need to do one of the following things with the `event-occurrence`:

- Allow it to have meta-data.
- Link to it from another neuron, or in any way use it separately.
- Link it to content explaining it.

Indeed, if you need to do one of the above, the `event-occurrence` clearly needs to be a neuron on its own. It is clear that not every relation can be a neuron, as this would lead to infinite recursion. The above points summarize the limit when a relation can be represented by an axon. Thus, the example shows the importance of considering carefully which neuron types you need.

3.4.5. Problems

The remaining problems in the design, in need of discussion, include:

- A more serious standardization of the visual attributes, as well as usage experiments to examine the relevance of the different attributes.
- A discussion of the role of the neuron type. Should you be able to inherit types? Which types are standard, and what identifiers should they have? Is the type an attribute of the neuron, or should it not in fact be part of the meta-data? Should the restrictions in the type be enforced by the system in some way?
- A discussion of the role of multiplicity. Is the concept well defined and useful, and is the design the right?
- Where should meta-data be located? Is it really necessary to download it together with the neuron? Perhaps a separate "meta-data" component is the solution?
- How are you supposed to find all the axons that point to a given neuron? Part of a solution is given by including `relation` informationint the meta data. But if you add a link to a neuron that you cannot modify, this does not help much. It becomes obvious that in the future, a standardized catalogue function will be necessary.

Chapter 4. Content

4.1. Scope

If neurons deal with the contextual organization of information, it is the content that contains the information that we organize. While concepts deal with the inner, mental world, content represents the link to the outer world of text, images and sound; the world of media and communication. The activation of content is one of the most important motivations for the creation of context at all.

4.2. Functional requirements

Content consists of explanations, examples and definitions of concepts used in learning environments, representing different aspects of the concept in question. It may also consist of physical or digital resources that have a representation in the system for lexical purposes. So the definition of content consists of two parts:

- Designing a representation of a piece of content in the system. This must include meta-data on the content, such as type information, and the location of digital content or the identification of physical content. The idea is that you should be able to search for and choose content without having seen the content itself, only this description.
- Associating such a content description with a concept or other type of neuron.

The ideas for the usage of content and aspects are carefully described in [cid52], and will therefore not be repeated here.

4.3. Architecture

The representation in the system of a piece of external content is made by a database component containing only meta-data, including type and location information. A concept map, being a component and thus having meta data, is in contrast represented by itself.

To allow these content descriptions to be treated as the content of a specific neuron, we use the `relation` meta-data entry, with a `kind` that equals `content`. While waiting for IMS to define the `identifier` element to be put here, we use an extension, `location`, to point to the component representing the content.

In addition, by adding `relations` with a `kind` that equals `context`, we allow the specification of concept maps that act as contexts for the neuron. Please note that concept maps may also serve as content, and that there is an important philosophical difference between the two.

4.4. Design Discussion

4.4.1. Adding content

One problem with the implemented solution is that it is impossible to add content or contexts to an existing neuron that you cannot modify. One could imagine doing a catalog search to add content to those specified in meta-data.

4.4.2. Complex content collections

A complex piece of content may generate large amounts of content descriptions. An example is an article like this, that would need one component per important section. One imaginable solution to this kind of problem is server-side generation of content descriptions on the fly from their identifiers, using for example XML markup in the article or other means. Quite generally, it is imaginable that the server stores the meta-data in a database of some sort, together with the content it describes. In this case, exporting the content descriptions over CORBA could be a reasonable idea.

Chapter 5. Concept maps

5.1. Scope

A concept map is a presentation of a part of the abstract world of interconnected neurons. A concept map is designed to emphasize certain associations between certain concepts, and this way provide a limited view of how the neurons relate to each other, without adding any actual information to the contextual network that the neurons constitute. There are many sorts of views possible, but the one described here is the one implemented - the standard two-dimensional UML-like view.

5.2. Functional requirements

A concept map contains the information necessary to present part of the neuronal context. This results in the following requirements for a concept map:

- It includes a number of neurons and certain of their axons.
- It defines positions for the visual elements that the neurons consists of.
- It allows the presentation of data in the map.

This is supposed to be done in a way that resembles UML as much as possible.

5.3. Architecture

A concept map is a component with the MIME type `application/x-conceptmap`. A concept map contains the following:

- An IMS meta-data record as described in [imsmetadata] (as all database components have).
- A bounding box for the whole map, which defines the coordinate system in which the rest of the coordinates are given.
- A filter to use as default sorting of the content of neurons. A filter given in a neuron (see Neurons) will override this.
- A list of neurons, referred to by their identity. For each neuron, a bounding box is given for the body, as well as a number of points constituting the line connecting the body with the axons. A title is given to each neuron, as well as a list of data tags that are to be displayed

together with their values. Each neuron may also have a so called detailed map, which is the map where you land if you surf this neuron.

- For each neuron listed, a list of axons that one wants in the map. Each given axon is given a list of points, where the last is the tip of the axon head. It is supposed that all axons start at the same point, although this is not enforced.

5.4. Design Discussion

In fact, given the structure of neurons already defined, defining simple concept maps is not very complicated; most elements are a must.

5.4.1.

That given, there are problems to discuss. One could ask why the detailed map is given here and not in the neuron itself. There are two reasons for this. First, this would constitute a breakage to the separation between the neuronal network and the presentation level (which may be replaced by other types of maps). The second reason is that which map one wants to link to actually is very dependent of which map one is regarding. Still, this should not stop us from considering placing a preferred detailed map in the meta-data of a neuron, as a sort of "primary context".

5.4.2. Double occurences

The implementation allows the same neuron to appear several times in the map. This feature is relatively new, but the need has been felt from early on in complex concept maps, where the otherwise resulting mess of axons is not pleasant. An axon thus points to only one of these appearances.

Note that the identifier given to each axon in a neuron (as described in Section 3.3) allows us to have several axons of the same type pointing to the same neuron. This will result in user interface troubles, when having to choose an axon to show in a map. It is expected that the vast majority of neurons will not use this feature, and that an axon type and the pointed-to neuron will usually uniquely identify an axon in a neuron, which is a natural way of identification. The exceptions will most likely mainly be machine-generated listings with double occurences such as histories etc.

5.4.3. Contained neurons

It has been discussed that one should allow concept maps to contain neurons. This would mean that one would not need to give very simple neurons definitive identities, as long as they are only used inside the map, and that one could distribute certain maps in an XML file, completely self-contained.

The neuron could, in this case, be referred to with the help of a "fragment identifier", which are introduced with the character `#` (see [rfcuri]).

Chapter 6. Filters

6.1. Scope

As described in [cid52], the different pieces of content of a neuron need to be sorted and filtered to make it possible to find the content fitting your wishes. These filters are fundamental to the realizing of content viewing, as content without order can be totally unmanageable. Defining filters includes defining how the term "aspect", used in the above article, is going to be used in our system.

6.2. Functional requirements

Filters should adhere to the following principles:

- They should work without the aid of concept maps, as they are useful in situations dealing only with the database structure, as well as in combinations with other sorts of concept presentations.
- Their definition should not depend on any data format.
- They should ultimately be able to produce several different aspect filtering and sorting systems:
  - One- or higher dimensional systems (list- or matrix-based aspect sorting), where the content is sorted into a grid of aspects. One axis could contain definition, location, use, history, etc., while the other contains the school level of the content.
  - Combinations with other type of filtering, corresponding to user profiles etc. that eliminates uninteresting content.

6.3. Architecture

6.3.1. Theoretical model

The filter implementation that we have used is based on a theoretical model that is independent of the aspect idea.

In this model, a "filter" consists of a number of "filter nodes" connected in a hierarchy. Each node takes a number of "packets" as input and gives as output only those packets which were accepted by that node. We call this "passing" the node. The packets themselves are in no way affected by the filter.

A node is usually not a complete filter even if this can be the case. Instead, a node can point to additional nodes which the packets must pass. Let us call these nodes which are pointed to by a node `A` for "direct refinements" of `A`, as they refine the filtration. If a filter node `B` is reachable from `A` through several steps of direct refinements, we call `B` an "indirect refinement" of `A`. If `B` is a direct refinement or an indirect refinement of `A`, we call `B` simply a "refinement" of `A`.

Hence, a filter consists of a top filter node with an hierachy of refinements. A packet is said to "pass" a filter `F` if it passes the top filter node, `A`, and also passes one of the filters represented by the direct refinements of `A`.

Another way of expressing this is to say that a filter is a directed graph of filter nodes, and that a packet passes the filter if it manages to pass all nodes in at least one path from the top filter node to a leaf.

The recursive definition of filter poses serious problems, and there is need for a limitation on filters to avoid them. The problem is that the definition does not force an acyclic strucure, but rather allows a general graph strucure. A typical problem arises when two filter nodes are refinements of each other, in which case there is an obvious risk for infinite loops. Filters without loops are called "sound" filters. Ensuring that a filter is sound is, unfortunately, a very demanding task, so most implementations will probably just use a maximum recursion depth or similarly limit the filters.

The filtering done in a node can be as domain-specific as possible. One general type of filtering that is always possible is to let one node filter the packets with the help of a "sub-filter". In this case, passing the node (say `A`) would be equivalent to passing the whole sub-filter[1].

Please note that a filter also can be used to sort packets. All that is needed is to attach a separate output box to each leaf of the filter. This is what will be used when implementing aspect sorting, as described in Chapter 16, The Content Displayer.

6.3.2. Neuron Binding

We now turn to representing filters with the aid of neurons. In our implementation, an "aspect" is simply the result of a filtering process. This way, the "historical" aspect of a concept is defined by

1. This is equivalent to connecting all leaf nodes in the sub-filter with all direct refinements of `A` (and thus represents a significant simplification), except from the fact that the sub-filter additionaly becomes reusable.

filtering out the content dealing with the history of the concept, something that can be done by a simple meta-data filter or by much more complicated filters, depending on the available markup and the complexity of the aspect. The current implementation presupposes that the packages are any type of components.

Filter nodes are implemented as filter neurons. A filter is thus represented by its top level filter neuron. A filter neuron contains:

- An axon type `subfilter` which points to a filter neuron to use as sub-filter.
- Data tags of the form `AcceptFormat`, `AcceptCoverage` etc. Only keyword search has yet been implemented, using the data tag `FILTERTAG`.
- An axon type `refine` that points to the direct refinements.

Hence, the filter neurons filter in two ways simultaneously: using data tags that filter meta-data and with sub-filters pointed to by axons. The sub-filter idea is not yet implemented.

6.4. Design Discussion

6.4.1. The history of aspects

The original idea behind the representation of aspects was to have a data tag in the content description neuron named "Aspect", that named by which aspect the content described a concept. However, this design seemed unattractive when considering the multidimensional aspect filters that we wanted to realize. The realization that many of those other dimensions (such as school level) probably would be represented in the meta-data, led us to make the decision to put the "Aspect" tag amongst meta-data as well. This way aspect filtering becomes meta-data filtering, which in turn actually allows any aspect filtering system to work as a general neuron filtering and searching system, thanks to the fact that content descriptions are neurons like any others.

6.4.2. Filter presentations

The hierarchy represented by the filter nodes is essentially a directed acyclic graph. This graph could often resemble a tree-structure, but not always, as the branches of the tree are allowed to grow together (even for sound filters). For very simple filters with a single level of refinements, a list presentation is reasonable. With two levels and the same level-two refinements in all level-one refinements, a matrix presentation is the most obvious.

In general, a menu with recursive sub-menus can examine any sound filter "locally", meaning that you cannot be sure to get a complete overview of the filter in this way.

More fitted to get an overview would be to present the filter in a diagram, in such a way that each filter node is represented by a box, connected to the rest of the filter nodes using arrows. In this way, each filter is a diagram showing an acyclic directed graph, where each filter node in addition is connected to its sub-filter, if any. These sub-filters could be incorporated into the diagram and connected to the filter node by a different type of association. Alternatively, each filter node can be connected to new diagram showing these subfilters.

It should be obvious that filter neurons are perfectly adjusted to be presented in concept maps. Each such diagram could be arranged in different ways, naturally: as a star, tree etc. This concept map would probably be added as content for the filter. Note that some refinements to the top filter neuron will have their own maps if they are seen as top level filter neurons for their branch.

6.4.3. Other filtered elements

This far, only meta data filtering has been implemented. To extend the filter usability outside of content filtering, one should consider other types of criteria when the packets are neurons or even concept maps, such as axon information or contained neurons.

II. Data representations

Chapter 7. Defining bindings

7.1. Scope

There are different representations of the database components that use different data formats. Such a representation is called binding, and form the connection between the virtual database containing neurons and concept maps on the one hand, and the concept browser on the other hand. The two bindings for which support is planned are XML and CORBA, of which only XML is fully implemented. On the other hand, nothing should prevent the use of, e.g., RMI, , ODBC etc.

7.2. Functional requirements

The very idea of having different bindings is based on the fact that they allow different optimizations. Working over the Internet, over a LAN, or directly against a database each places very different demands on the communication protocol. For example, one could imagine a browser written for mobile phones that uses WAP, or another browser running in contact with your company database using database routines directly. With the data format independent specification in place, we are free to translate it into different formats. What is defined here is the canonical bindings using several formats, but by no means the only one possible. Also, one browser can support several of these bindings simultaneously, which is also true of the databases. So, our goals for each binding are the following:

- The binding should be as natural and clean as possible. It should not try to solve problems that the format is not well adjusted to solve. For example, the CORBA binding will have performance problems when used over the Internet, and should not try to solve these. Therefore, each binding should have a well-defined domain of optimal use.
- On the other hand, the binding should try to use the particularities of the technology used to allow optimizations within the domain of optimal use.
- As each binding defines a communication protocol, the design is expected to add, to the component representation, a layer that defines the capabilities with respect to editing and storing of changes.
- A binding should also define the possible means of locating the component, so that given the binding and a URI locating the component, it is clear how to access it.

7.3. Architecture

The architecture of the two bindings is described in Chapter 8, The XML binding and Chapter 9, The CORBA binding.

7.4. Design Discussion

The ideas behind the different data representations has been implemented only for XML, even though CORBA has been on our minds when designing the system. It is clear to us that more experimentation is necessary in order to declare the system a success.

That the XML binding is the most interesting and versatile format has been clear from the beginning of the project, but it has also been clear that this format will have problems in highly interactive situations, where a more direct contact with the data source will be necessary. This is the reason to allow several different bindings to coexist, even though this makes the system design a bit more complicated.

Chapter 8. The XML binding

8.1. Scope

The Extensible Markup Language (XML) binding is the only representation fully implemented in the Conzilla system and certainly the one most widely useful. The implementation does, however, lead the way towards implementing other formats, as many of the interesting problems must be solved already for the XML binding.

8.2. Functional requirements

The XML binding is intended to be used in a wide variety of environments. The format should be a totally disconnected format, in the sense that after having delivered the component, the server will no be longer involved. The analogy with a web server with passive pages is obvious, and the reason for this constraint is that it allows a world of users simultaneously using the components. Even if the component changes, the user will not notice before a reload.

8.3. Architecture

The DTDs are given in Appendix A. Note that they do not allow any free-text markup[1].

The access mechanisms allowed for this format are the standard HTTP and local harddisk accesses represented by the `http:` and `file:` URL protocols. Note that for component identifiers using one of these protocols, the XML binding will automatically be chosen, while other bindings in theory could use these protocols after having been through a Path URN resolver. See Chapter 2, Component Identity.

Updating a component using the XML binding can only be done by replacing it entirely via, for example, a file save command or a HTTP `PUT` command. If no saving destination is given or access to this destination is denied, the component may not be edited.

1. With one exception: the `extension` mechanism in the IMS meta data XML binding.

8.4. Design Discussion

8.4.1. XML

Luckily, translating the definition of the different components into XML is a relatively straightforward operation. In fact, the component design has been made directly in XML. XML was part of the very early design decisions, and therefore has much influenced the design. The reasons for choosing XML as the most fundamental data format are several:

- "XML shall be straightforwardly useful over the Internet" ([xmlspec]). Actually, XML fits perfectly into the scheme of using URIs and the existing URL protocols, being related to HTML as it is.
- XML also is immediately useful as a data storage format, immediately distributable using any web server. This is not a small point, as getting involved in the business of using a database could be unnecessary overhead for small content providers and for our prototyping.
- "It shall be easy to write programs which process XML documents" ([xmlspec]). And there are indeed a multitude of parsers for XML for any important programming language.
- "XML documents should be human-legible and reasonably clear" ([xmlspec]). This has much helped us in prototyping, as no special tools are needed to analyze an XML document.
- "XML documents shall be easy to create" ([xmlspec]). This point is incredibly important. You can create XML documents using nothing but a text editor, which was, by necessity, the case before we had constructed a complete editor. But it is also easy to create XML documents from inside a program, which has helped us in the prototyping.

This list could easily be continued. Of course, there are problems, such as the following:

- The updating procedure when you are not located on the same file system as the XML files turns out to be complicated, suffering from the same difficulties as updating HTML pages. But this is an expected side-effect of the extremely simple distribution. The components in XML format are not primarily meant to be edited from a distance, but rather to distribute ready-made components.
- XML is not the most compact format imaginable. However, we do not expect the components to constitute the largest part of the data used, but rather the content in the form of images, videos, etc. Also, this non-terseness in representation is the basis of many of the advantages of the XML binding, so it will be conditionlessly accepted. And it is still relatively lightweight, in comparison with, for example, CORBA.
- The canonical programmatic interface to XML, the Document Object Model (DOM), is a very heavyweight representation not suitable in all situations. We have solved this problem in the

class library. See Chapter 12, The Class Library.

8.4.2. 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 XML binding of our objects? The reason is that it shares many of the problems with the MOF as described in Section 1.4, more precisely:

- Object oriented analysis and design fixation in the modelling.
- Distributability problems. An XMI document describes all classes and associations contained in one closed model, typically one whole Repository. This is orthogonal to the modular design we want, and indeed, the goal of XMI as described in [xmispec] 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 complete models, a concept that in a sense does not exist in our system, as our models usually not are closed.
- 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 [uxfspec] that is a more lightweight XML description of UML diagrams which has inspired us in many ways.

Chapter 9. The CORBA binding

9.1. Scope

The CORBA binding is the first candidate for an extension of Conzilla's capabilities to other data formats. 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.

It will highlight the most fundamental problems of designing such an "active" binding, and therefore constitutes an important challenge for Conzilla in the direction of internal company use, as an alternative or complement to a MOF model. As nothing of this is yet implemented, this discussion is still a bit hypothetical, even though a first implementation actually would not be too difficult.

9.2. Functional requirements

A CORBA binding will probably have to adhere to the following guidelines:

- Each object should be active (connected to the data source), but the connections between components should not become CORBA object references, but instead remain simple identity references, i.e., the components should be individually distributed. This is because other components actually might be in the XML format.
- Changes in an attribute should be immediately distributed to all others using the same component. Thus there is no saving step involved. The component may deny any editing if the user is not authorized.
- The API should be immediately usable in an application, without adding extra layers.

9.3. Architecture

There will exist four main CORBA interfaces, namely `Component`, the base class of `Neuron`, `NeuronType`, and `ConceptMap`. These include an interface to the meta data record. These are mostly already defined, as interfaces in the Java class library, and will be converted to CORBA IDLs in the future

9.4. Design Discussion

The primary design influences for a CORBA solution has been `CORBA::RelationShips` and the MOF, both discussed in Section 1.4 . The most important difference in design is the decision not to use active connections between objects, which is an absolute demand to allow other data formats.

This decision is problematic, as the advantages of CORBA are not fully exploited. We imagine at least two solutions:

- One could imagine a sort of semi-direct connection, that allows direct connection if possible, but do not force it. This is, however, difficult to implement. The whole problem can be summarized as a problem of associating local data to a CORBA object without using, e.g., a hash table (which, on the other hand, definitively solves the problem, and which is also the solution implemented).
- Perhaps we will need to make the decision not to mix the different data formats at all. Only the future can tell if this rather radical solution will be necessary.

III. The Implementation

Chapter 10. The Conzilla Prototype

10.1. Scope

The Conzilla browser is the first prototype implementation of a conceptual brower, and therefore forms the testbed for all our ideas concerning concept browsing. Thus, what is described here should not be understood as implementation specifications for future browsers, but rather as proof of concept (no pun intended) and as a source for inspiration. But still, nobody would be happier than us if the prototype turned out to be practically usable.

10.2. Functional requirements

The Conzilla browser has been designed with several thoughts in mind:

- The choice of programming environment must be done with care. This is discussed in Chapter 11, Programming environment.
- It should be based on a library dealing only with the representation of the components and the different data formats, to allow the reuse of this code in different situations. This is discussed in Chapter 12, The Class Library.
- It should adhere to exiting standards to the largest amount possible. By following standards we ensure compatibility now and in the future, but we also increase the amount of existing implementations immediately usable.
- Trying not to tie the browser to a certain platform is a necessary design principle to ensure that the design of the database structure does not itself become platform dependent, something which could obviously not be accepted.
- A good separation between the different components will be necessary, to simplify replacing them with new implementations, something that will happen.

10.3. Architecture

The choice of programming environment is described in Chapter 11, Programming environment, while the base library is described in Chapter 12, The Class Library. The different parts of the browser are:

- The browser itself, containing the map management and browsing functions. It is described

in Chapter 13, The Browser.

- The editor, used to construct neurons and maps graphically. It is described in Chapter 14, The Editor.
- The library, used to maintain collections of neurons for purposes such as history, bookmarks etc. It is described in Chapter 15, The Library.
- The content sorting and viewing mechanisms. They are described in Chapter 16, The Content Displayer.

Each of the above are separate and, to the greatest extent possible, independent entities.

10.4. Design Discussion

The browser is an ever changing construct, and necessarily so. Changes in specifications, user design wishes, and implementation design has forced several rewrites of fundamental components. This way, it has evolved into a highly modular structure to allow these changes to take place without needing to redesign other parts.

This structure is mostly inspired by technical needs. There has been too few user studies, something that should be initiated as soon as possible as it could profoundly affect the design of certain parts.

Future additions include adding a help desk system, as described in [cid52]. Hopefully, this can be done without affecting the current code.

The name "Conzilla" has been inspired by the Mozilla (http://www.mozilla.org/) browser, with the twist that it browses the context and content of concepts. The interpretation "CONZeptual ILLuminAtor" has been suggested, and a relation with "consilience" as used by Edward O. Wilson in [wilson] is also proposed. Wilson uses the words of the 19th century philosopher, William Whewell, who defines consilience as "a 'jumping together' of knowledge by the linking of facts and fact-based theory across disciplines to create a common groundwork of explanation."

Chapter 11. Programming environment

11.1. Scope

Deciding which tools to use in order to implement the browser is a problem that affects many parts of the design. The decisions to make include which programming language to use, which programming libraries to use, and which other external software to depend on. Once these choices are made, we have severely limited the directions in which the browser may develop without a rewrite.

11.2. Functional requirements

The principal design goal when it comes to the choice of programming environment has been: a maximum of functionality with a minimum of effort. Here, functionality means different things:

- Availability of useful libraries and functionality.
- Portability to different platforms.
- Availability for ordinary users.
- Possibility of implementing content activation for a maximum of content formats.

but not primarily performance, notably, which will be a later problem[1]. Low effort means:

- Prototypability, i.e., the ability to quickly try out ideas. This demands a low level of implementation overhead and nonfundamental details, and enables us to focuse the design on the structure rather that on the implementation.
- Extensibility and flexibility, to be able to maintain a certain structure in the code in spite of large rewrites.

One must, however, have in mind that the choice of environment must not influence the design in a way that would cause troubles that day when we decide to change the environment.

11.3. Architecture

1. Donald E. Knuth: "Premature optimization is the root of all evil."

The Conzilla browser is being implemented in Java, based on a Java class library to deal with all basic database functionality. The application will work either standalone or as an applet in a web browser. In either case, it can use a web browser to provide content activation for a large number of data types: hypertext, pictures, sound, video etc. (in the applet case, the same web browser). It uses Java 1.2 with the JFC Swing toolkit for all GUI code.

11.4. Design Discussion

11.4.1. 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. The most general content displayer, for content in standard formats, is surely 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 decided at an early stage in the design process that using a Web browser as content displayer solves the problems of displaying content in the most general way.

It is interesting to note that the current generation of browsers (Netscape Communicator version 6 and Microsoft Internet Explorer version 5) does support XML directly as a way of representing arbitrary information. This could be used to retrieve and parse the components 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 Style Language (XSL) document. We have not thoroughly investigated whether this will be possible to do or not.

11.4.2. Java

The problem of getting an application to work on different platforms cooperating with different browsers without rewriting the code is clearly solved by the Java programming language. This way, the application can actually be run both inside the browser, started from a website, or as a stand-alone application, remote-controlling a browser to display the content. Running as an applet undeniably increases the availability to users, as there is nothing to download and install.

Java is a very useful prototyping language for several reasons:

- It is strictly object oriented.
- Its memory management is simple for most simple cases.

- There is an abundance of useful libraries, all automatically portable. Not least, the GUI libraries are cross-platform, which saves an enormous amount of work.

The choice of GUI library deserves a short discussion. In the Java environment there are two real possiblities: either you use the standard AWT-package, or you use the more recently developed JFC Swing package. We have choosen to work with the JFC Swing package, as it is, in comparison to AWT, better designed, more extensible and more flexible. 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, as is the case with AWT.

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 the design goals very well. There are, of course, problems.

11.4.3. Security problems

Firstly, as our application can run as an applet inside the browser, it is subject to applet security restrictions. These involve (taken from Frequently Asked Questions - Java Security (http://java.sun.com/sfaq/)):

> 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. These restriction are not acceptable for anything else than a simplistic demonstration. There are several imaginable solutions to this problem:

- Implement a proxy on the host where the applet is located, that can fetch components that the applet asks for.
- 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.

The two browsers we have studied 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 unstandard and tied to the Windows platform, and has therefore been avoided for the prototyping stage.

Netscape Communicator 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, as it tries to be cross-platform and standards-adhering. Fortunately, it would 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, but for the moment, the security support only works with Netscape Communicator.

Finally, as browsers start to support Java 1.2, they will also start to support the Java 1.2 security model, which is also based on the idea of signing. Then we will have a less platform-dependent solution. At the time of designing the security handling code in Conzilla, we were using Java 1.1, and so this was not a possible path. When writing this document, Java 1.2 support is arriving, so it may be time to reinvestigate the issue.

11.4.4. Java versions

The second problem is that the decision to use Swing to construct the GUI sadly limits us to newer versions of Netscape Communicator (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 initially seem. We have hopes that future browsers will support the Open JVM Integration (OJI) 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.

11.4.5. JavaBeans

There has been some discussion of simplifying the work for course designers and teachers by making the whole Java implementation JavaBeans compatible. 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 our editor probably will never be and was not intended to be. But, as this seems to be relatively far into the future, the details of such a project remain vague at this point.

Chapter 12. The Class Library

12.1. Scope

The class library is a set of Java classes that together form an API for dealing with components and the different data formats. It represents the most stable and reusable part of the implementation, as well as the border line between the database design implementation and our particular concept browser.

12.2. Functional requirements

The class library is designed after the following goals:

- It should contain
  - the component and identity handling routines,
  - the data format specific routines,
  - the neuron, neuron type, and concept map representations,
  - the content description handling routines, and
  - the filter implementation.
- It should be immediately usable and practical in the implementation of a concept browser.
- The API should be usable in other contexts than the browser, such as a search engine or other software using the system. In particular, the library should not depend on any GUI or interactive code.
- The API should abstract the different data formats, and thus must not be dependent on the XML or the CORBA binding. Internally, there should of course be support for the abstraction of each format, as well as the possiblity to add new formats without affecting the external interface.
- The neuron, neuron type and concept map representation should represent a Java binding of the specification, and preferably in such a way that the defined interfaces are easily translatable into CORBA IDL definitions. In practice, this means among other things that the interface must use only basic data types or CORBA types, but not Java-specific types. This way, the Java and CORBA bindings are made simultaneously, much aiding a CORBA implemetation at a later stage.

- The library should also be implemented in a way that separates the representation of the completely non-visual global conceptual environment from the visual parts concerned with concept maps, on order to enable the addition of other types of concept maps.

12.3. Architecture

The library consists of several main packages.

12.3.1. XML

This package contains the functionality to deal with simple XML documents in a simple way. Features:

- A document object model which can handle XML documents with the following restrictions:
  - Free text markup is allowed only under the condition that the position of subelements in the surrounding text is not recognized. That is, each element contains a single CDATA element in addition to its subelements.
  - If several different types of elements are allowed as sub-elements for a given element, they may be mixed, but the mixing is not recognized. Only the order of sub-elements within each sub-element type is recognized. When saving, they are not mixed at all.
- A parsing mechanism using the Ælfred (http://www.microstar.com/aelfred.html) XML parser.
- An exporting mechanism.

12.3.2. Identity

This package contains the URI parsing routines as well as the standard Path URN resolving routines.

12.3.3. Component

This package contains the generic component handling routines.

- IMS Meta data representation, including a Java interface and a listener mechanism along the same lines as for the Neuron and ConceptMap packages.

- Resolver functionality and handling of identifiers. The implementation separates the following categories of functionality:
  - The resolver engine, mapping a URI to a URI and a data format. For `http:` and `file:` URIs, this is the identity mapping, adding XML (as `text/xml`) as format. For Path URNs, this is implemented using a table, mapping path names to base URIs and giving the data format that is used by that path, as described in Chapter 2, Component Identity
  - The handler for each data format. This handler knows how to use the URI given by the lookup routines.
  - The actual handler for a certain URI protocol. The XML handler (the only one yet existing) passes the URI to the XML parsing subsystem for both `file:` and `http:` URIs. When saving, only `file` is yet sopported, but HTTP `PUT` requests would be simple to implement, and even FTP saves should be interesting.
- Packing and unpacking routines for the XML binding. This package uses the XML package to parse the downloaded data into the simple document model defined there, which is then traversed and the component Java objects created. Inversely, when saving, such a document model is constructed and then written to the URI used to save to.
- A cache subsystem. The implementation is currently pretty simplistic, but the architecture is in place.

12.3.4. Neuron and ConceptMap

These packages contain the implementation of the neuron and neuron type components, and the concept map component, respectively. They consist of two main parts:

- The Java interfaces for the components, inteded to be used when supporting other formats, and for constructing the CORBA IDL.
- The Java implementation of the interface, used for downloaded components (in contrast to remotely used components), be it in XML or other formats.

The most important characteristics of the design are:

- Meta-data is represented by a separate object, which is of the same type for all components.
- Each attribute is associated with a listener mechanism. That is, whenever the attribute changes, all registered listeners are notified.

The Neuron package is completely independent of the ConceptMap package.

12.3.5. Content

This package contains wrappers to simplify the loading of components referred to in the meta data of a neuron.

12.3.6. Filter

* What is there to say about the filter implementation??

12.4. Design Discussion

12.4.1. XML

The reasons for implementing a more simplified document object model when there are several fully compliant DOM XML parsers available are several:

- The DOM parsers are in general very heavyweight, which we found unattractive for use in an applet. By contrast, the Ælfred (http://www.microstar.com/aelfred.html) parser is extremely lightweight in spite of being a conforming XML parser.
- The resulting object model is very complicated and heavyweight. The DOM is intended to be able to describe any type of XML document containing arbitrary markup. Again, our decision to enable the use of our browser in a web environment has led to demands of simplicity. In fact, as our XML DTDs are used as data structures rather than documents, the needed object model can be made much simpler without loosing any necessary functionality. The use of this simpler object model makes the code using the model much simpler as well.

Thus, we decided to use a simple parser and design a simple object model that fits our realively uncomplicated DTDs, but would not fit for example an HTML DTD. The decision still seems to have been the right one.

12.4.2. Component

The component package design is extremely modular. Designing the package in this modular way has made clear the responsabilities of the different parts of the design, which in turn has influenced the design of the data format and component identity specifications. The structure will help when adding resolver functionality and other data formats.

12.4.3. Neuron and ConceptMap

The need for a listener mechanism is motivated by two cases of use:

- When the same neuron is used several times in the same browser (in different maps), and is edited in one of them, it is immediately updated in the other.
- Even when several users access the same CORBA object, they will all be notified when someone modifies the object.

The risk of not using a mechanism like this is to create inconsistencies when editing, something that is most problematic when using CORBA.

The need for full independence of the neuron level and the concept map level is motivated by the possibility to use the neuron level in other types of concept maps, but also by the fact that this library may be used when constructing server-side search engines and similar tools.

12.4.4. Content and Filter

These parts of the library are not strictly part of the fundamental API, but still seem general enough to belong here. It is interesting to note that these constructs are implemented solely with the help of the API, using the components as a data structure.

Chapter 13. The Browser

13.1. Scope

The browser is the primary goal of the whole design, and the construct the most visible to the user. It is the point where all concept map, neuron and neuron type information come together, and with the addition of user interaction it becomes the starting point for the utilization of all the functionality in the Conzilla system.

13.2. Functional requirements

With the browser, we want to be able to do the following:

- View a concept map and all information supposed to be displayed therein.
- Surf a neuron to arrive in its detailed map.
- View the metadata of the neurons in the map.

The responsability to sort, choose and display content is given to the content displayer.

13.3. Architecture

The browser consists of three principal layers:

- A module responsible for the graphics in the map. This module will examine all information given in the concept map, neurons, and neuron types and construct graphical objects that operate independently of these components. These objects localizes mouse- and keyclicks and delegates them to interested listeners.
- A map controller system, which is responsable for the structures attached to a map, such as filters, libraries and content viewers, and contains a general interface for adding tools. It performs the surfing actions, and emits history event accordingly.
- The different tools and modules that use the above layers to interact with the user.

13.4. Design Discussion

The three different layers are motivated by several wishes:

- The displaying module could be (and is) used in a content viewer for viewing maps without being able to surf them, or even for superposing maps.
- We will add tools not originally thought of. By forcing all functionality to use the general interfaces, we assure that they are usable to the needed extent. By carefully separating responsability, we ease the development of new features by allowing experimentation without altering existing code.
- By separating the functionality from the user interaction, we allow different implementations of the user interfacing, which is an important part of the project.

Different ways of making concept maps more interactive have been discussed. There are several ways in which this may be possible:

- allow the user to move certain concepts inside the map, to increase readability
- allow the user to hide parts of the concept-map temporarily, to increase clarity

This kind of functionality becomes even more important if we introduce automatically generated concept-maps, such as search results, and showing the result as a concept-map, or even included in the current concept-map. Being able to "clean up" the map would be a much wanted feature.

Chapter 14. The Editor

14.1. Scope

The editor is a fundamental part of the prototype, as it (ideally) hides from the user all the technical details of constructing neurons and maps. But the editor functions are useful for more than that; it can also be used to increase the level of interactivity of concept maps.

14.2. Functional requirements

The editor should fulfil the following goals:

- It should be independent of the browser in the sense that the browser should work without it.
- It should be independent of the data format the edited component happens to use.
- It should allow the editing of all fields in editable neurons and conceptmaps, and possibly neuron types. This editing should be graphical or aided by graphical hints if possible.
- It should be responsible for the saving of changes.

14.3. Architecture

The editor is implemented directly in the browser system, so that you are able to edit a map in the very same window you use for browsing it.

The editor is implemented using tools that attach mouse and key listeners to the map graphics elements. In turn, when editing, the editor modifies the actual concept map and neuron objects, and the map display will be automatically updated using property change listeners.

The editor consists of three main parts:

- The map editor, which does all the graphical editing of conceptmaps.
- The neuron editor, which does all the editing of neurons.
- A metadata editor, which edits metadata for neurons and for concept maps. This is currently not functional.

14.4. Design Discussion

The intense use of listener mechanism has made it possible to make the edit functionality one tool amongst others. As the editor is heavily dependent on user interaction, and therefore user interface design, it is expected that it will develop considerably in the way it works with the edited elements. Thus, the current method of implementation ensures minimal impact on other parts of the code, notably the browser.

The separation of editing functionality is necessary because being able to edit a neuron directly in the concept map would most probably be confusing, as one actually edits different objects.

It is noted that we have no method of creating and editing neuron types. This should be added.

Also note that the functionality in the editor for moving and otherwise manipulating concept maps could be used to increase the interactivity of concept maps in the sense described in Section 13.4.

Chapter 15. The Library

15.1. Scope

* This chapter must be completed with the help of Matthias.

15.2. Functional requirements

15.3. Architecture

15.4. Design Discussion

Chapter 16. The Content Displayer

16.1. Scope

The content displayer completes the browser with a method of accessing content, which, after all, is one of the two fundamental browsing steps. It contains the methods for viewing and interacting with content, as well as the application of filters on content.

16.2. Functional requirements

The content displayer has the following responsabilities, all described in detail in [cid52]:

- Use the specified filters to sort content for a neuron into aspects, and present the result of this sorting to the user.
- Present content according to the user choice.
- If the content is another concept map, allow the "contextification" of this map, meaning that the map is "sucked in" into the browser to allow it to be browsed (and edited, of course).

16.3. Architecture

The content is sorted according to the filter specified in the concept map, or if the neuron itself specifies a filter, this filter is used. The sorting is displayed as a meny with submenus as outlined in Chapter 6, Filters, and when a leaf is selected, the contents reaching this leaf are listed, and the user is allowed to choose among them.

After having chosen a content to view, the content displaying is multiplexed with respect to the MIME type of the content. The MIME type `application/x-conceptmap` is displayed using a simple map displaying window without any tools (and is thus completely uninteractive). It can still be contextified using a tool in the originating map. Other MIME types are handled by a web browser. There are two possibilites:

- Conzilla runs as an applet in a web browser. Then a frame named `content` is used to display the content.
- Conzilla runs as an independent application. Then Conzilla tries to use a running browser to display content.

16.4. Design Discussion

The content displaying is an area where there are many future possibilities when it comes to increased interaction with content. They are discussed in Chapter 17, External Cooperation.

It is worth noting that we have only implemented support for Netscape Communicator as external content displayer. Adding further support is, however, trivial.

The URI given in the content description, and which is used to locate content, may be of any general form. When viewing a concept map as content, the displaying is handled by Conzilla itself, and thus the complete machinery of `urn:path:` resolving is used. Unfortunately, this is not used for other types of content. Instead, it is up to the web browser used to interpret the URI. This could cause problems in the future. See further Chapter 17, External Cooperation.

Chapter 17. External Cooperation

17.1. Scope

The Conzilla browser does not operate alone. Instead, it depends on external tools for, notably, the displaying of content. This interaction must not necessarily be limited to just one direction. Instead, by allowing the content to interact with the browser we lift Conzilla to a whole new level of interactivity and enable many intriguing possiblities.

17.2. Functional requirements

The reason why interacting with content could be interesting is explained by the example of the current article. The concept maps describing this article will have sections of the article as content. The text in these sections will contain hypertext links to other parts of the article. But when using these hyperlinks, the context described in Conzilla will not automatically be updated to show the context of the link end. Thus, it would be desirable to be able to either automatically update the context when following a link, or to have a button such as "Show context" in the article display that shows the context of the current text. This would demand that Conzilla be controllable using, for example, JavaScript.

It should be clear that allowing this type of manipulation of context from inside of content could be immensely useful when navigating complexly interlinked content collections, which is probably not at all going to be an unusual case.

The requirements on Conzilla for it to be able to interact with external tools include:

- It has to have a standardized interface. Note that the specification of this interface may very well have be made completely concept browser independent, as content is not supposed to know which browser the user is using. This places strong demands on the interface, which must be carefully designed to meet the following criteria:
  - It must not be dependent on Java specific features or types or features specific to the Conzilla browser.
  - It must be able to perform the fundamental tasks of concept map browsing, which include surfing to a given map, showing a given content, etc. The details of this functionality is not yet clear.

- There must be a means to locate Conzilla and the interface. This includes specifically from inside of content, where at all possible.

17.3. Architecture

None of this functionality has really been implemented, even though there has gone some thinking into the map controller interface, which makes it probable that this class will serve as model for the external interface.

17.4. Design Discussion

It must be noted that already, the Java classes are usable from inside a browser using either Java or JavaScript. It has been demonstrated that a small piece of JavaScript is able to control the browser. The interface is, however, not yet clean enough.

The external interface will probably be designed in at least two ways: in Java, to be used from inside a browser, and in CORBA, for general external use.

Possibly, we will implement support for content displaying using the Mozilla (http://www.mozilla.org/) browser. We will be able to embed Mozilla into Conzilla, and the other way around, use Conzilla as displayer of concept maps in HTML pages, allowing interaction between Conzilla, the embedded Mozilla, and the concept map displayer inside Mozilla.

IV. Technical design of the article

Chapter 18. Structure

18.1. Scope

This article is not simply a large piece of text; it is an active article intended to demonstrate the very system it tries to describe. This chapter describes the structure and usage of the article itself, i.e., how it is written and how it it intended to be used.

18.2. Functional requirements

The usage of this article is not limited to printed format. It is intended to be used in several different environments:

- As a printed reference.
- As an online reference in HTML, well integrated and linked into the other parts of the Conzilla Web pages (http://www.nada.kth.se/cid/il/conzilla).
- As content for a collection of Conzilla maps describing the functionality of the Conzilla browser. This content should actually interact with the Conzilla browser, as the article will contain concept maps to describe its interconnections.

This combination, in fact, results in a revolutionary new way of writing documentation which places high demands on the surrounding technology. This same text must not only be able to exist and be updateable in three different environments simultaneously, but it must also be so well organized that the "componentification" necessary for use in the Conzilla environment becomes possible.

But why is all this complexity worth the while? The real advantages comes when the article can be placed in a proper context. In our case, this article plugs right into the concept maps describing the project in general, the partners, etc. It also gives us the possibility of actually linking directly from the relevant descriptions into the API reference or the philosophical motivation, thereby making the complete context of the article immediately accessible. This is the revolutionary aspect of this document.

Additionally, experimenting with this kind of usage of a document is an irreplacible source of experience for the Conzilla project.

18.3. Architecture

The document format is XML with the DocBook XML DTD. It is possible to generate plain text, HTML, TeX (and therefore DVI, PS, PDF etc.) and other output formats, as well as directly use SGML/XML tools like sgrep to extract parts of the document. The document can be used even though it is not completed, which allows early web access to and experimentation with the parts already completed.

The article consists of structured text and concept maps. The text has been structured in chapters, where each large concept, and possibly also important associations between concepts, has its own chapter. The single sections in each chapter serve as content for the concept described, e.g., this section describing the architecture of this article serves as content for the concept representing this article, under the aspect of "architecture" (...read slowly). Smaller concepts, not described in a chapter of their own, may have content that consists of a smaller part of a section describing another concept. For example, this paragraph may be the content of the concept "structured text" under the aspect of "architecture". The aspects in which we have chosen to separate the content are:

Scope

: Describes what is contained within the chapter: the limits with respect to other chapters and an introduction to the components contained therein.

Functional requirements

: Describes how the design is intended to be used, i.e., the possible uses and wanted functionality which have influenced the design.

Architecture

: Describes the internals of the present design, i.e., how we manage to produce the wanted functionality.

Design Discussion

: Discusses the usage of the current design. Presents the motivations for the design and discusses previous designs and why they were abandoned, as well as alternative designs that have been considered. Points to problems in the present design and design goals for the future.

The concept maps are different things in different output formats. In printed form, they are simply pictures, while online, they will be used to interact with Conzilla.

18.4. Design Discussion

The only reasonable choice as document format that adheres sufficiently to the principles of polyvalence and structure is XML (or SGML, of course). The choice of DocBook as DTD is based on its status as a growing and well-proven standard for technical documentation.

As this is the first reference work prepared to be used directly in a Conzilla environment, it is interesting to note the experience it has given, as we hope that the article may form a model for how to write reference documentation with the help of Conzilla. For a technical document like this, that tries to be very systematic in its descriptions, it is impossible to even start considering what to write before having decided under which aspects to describe the concepts, as these represent the classification of information. Therefore, it is very much worth it to spend time analyzing the wanted aspects. This is in turn impossible until you have an idea of which concepts you want to include. So the process of writing becomes:

1. Draw all the concept maps (or nearly all) that you want to include. Decide which concept are worthy their own chapters. This becomes the scope and the structure of the article.
2. Consider the aspects you want to describe. This reflects the depth and target audience for the article.
3. Decide the order of chapters and possible appendices. This is important for the logical structure of the article. Without the right linear order, it becomes unbearable to read.

Please note that this "bondage & discipline"[1] way of writing is applicable primarily to reference documentation. Other types of documentation, such as reports etc. need another, freer style of writing. This does of course not stop them from being introduced in a Conzilla environment as well. It just lessens the level of structure.

1. Attributed to Eric S. Raymond

Chapter 19. Presentation

19.1. Scope

After having defined the structure and technology to use when writing this article, there remains the problem of designing the presentation. This chapter describes what has been done in order to achieve the result you as a reader (hopefully) are seeing.

19.2. Functional requirements

The goals for the presentation has been clear:

- Produce a serious, professional, printed reference.
- Produce an online version that can be customized by adding pieces of JavaScript etc. to activate content.

Both of these include being able to import concept maps in a, for the presentation medium, well adjusted format.

19.3. Architecture

I have used the Modular Docbook DSSSL Stylesheets (http://nwalsh.com/docbook/dsssl/) for the formatting of this article. I have had to change two things in the stylesheets:

- Comment to Remark in `dbblock.dsl` and `dbfootn.dsl` in order to support DocBook 4.0.
- "Chapter %n" to "Chapter %n, %t" in `dbl1en.dsl`.

Apart from this, I have made some minor customizations to adjust the presentation, but these do not affect the stylesheets themselves.

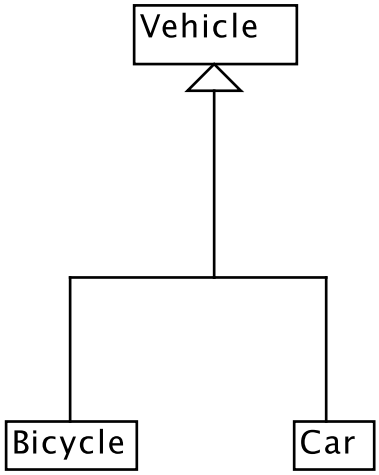
19.4. Design Discussion

It has still to be shown that all the wished functionality can be easily implemented. The problems still remaining are more precisely the following:

- The format of concept maps for printing. Optimally, they should be exported to EPS from

Conzilla, using the Java Printing API, for example, to allow highest possible resolution (no visible pixels in fonts etc.). That this works is displayed by this example image:

Example 19-1. Figure example



- The exact meaning of links in the document in a Conzilla environment: a link to the corresponding concept, a link to the corresponding content, or in some way both? Another problem is the inclusion of suitable JavaScript code to talk to Conzilla. See Chapter 17, External Cooperation.
- The meaning of a concept map in the online version. It should probaby be used to start the Conzilla browser on the map, either inline or via a link. As content in a Conzilla environment, they would be importable (and therefore surfable) into the Conzilla browser. the corresponding JavaScript code remains to be written.

Appendix A. XML DTDs

A.1. The Component DTD

```
<!- Component ->

<!ELEMENT Component             (MetaData,
                                 (Neuron
                                 |NeuronType
                                 |ConceptMap)?)>

<!ELEMENT MetaData              (record)>



<!ENTITY % MetaData SYSTEM "IMS_METADATAv1p1.dtd">
%MetaData;

<!ENTITY % Neuron SYSTEM "Neuron.dtd">
%Neuron;

<!ENTITY % NeuronType SYSTEM "NeuronType.dtd">
%NeuronType;

<!ENTITY % ConceptMap SYSTEM "ConceptMap.dtd">
%ConceptMap;
```

A.2. The Neuron DTD

```
<!- Neuron ->

<!ELEMENT Neuron                (Data?,
```

```
                                    Axon*)>
<!ATTLIST Neuron
        TYPEURI                     CDATA                   #REQUIRED>

<!- END Neuron ->


<!- Data ->

<!ELEMENT Data                  (Tag*)>
<!ELEMENT Tag                   (#PCDATA)>
<!ATTLIST Tag
        NAME                    CDATA                   #REQUIRED>

<!- END Data ->



<!- Axon ->

<!ELEMENT Axon                  (Data?)>
<!ATTLIST Axon
        ID                      ID                      #REQUIRED
        ENDURI                  CDATA                   #REQUIRED
        TYPE                    CDATA                   #REQUIRED>


<!- END Axon ->
```

A.3. The NeuronType DTD

```
<!- NeuronType ->

<!ELEMENT NeuronType            (DataTags,
                                 BoxType,
                                 LineType,
                                 AxonType*)>
```

```
<!- END NeuronType ->


<!- DataTags ->

<!ELEMENT DataTags              (DataTag*)>

<!ELEMENT DataTag               EMPTY>
<!ATTLIST DataTag
        NAME                    CDATA                   #REQUIRED>


<!- END DataTags ->


<!- BoxType ->

<!ELEMENT BoxType               EMPTY>
<!ATTLIST BoxType
        TYPE                    CDATA                   "rectangle"
        COLOR                   CDATA                   "black">

<!- END BoxType ->


<!- LineType ->

<!ELEMENT LineType              EMPTY>
<!ATTLIST LineType
        TYPE                    CDATA                   "continuous"
        THICKNESS               (0|1|2|3|4|5
                                |6|7|8|9|10)            "1"
        COLOR                   CDATA                   "black">

<!- END LineType ->


<!- AxonType ->

<!ELEMENT AxonType              (DataTags,
```

```
                                HeadType,
                                LineType)>
<!ATTLIST AxonType
        NAME                    CDATA                   #REQUIRED
        MINIMUMMULTIPLICITY     CDATA                   "0"
        MAXIMUMMULTIPLICITY     CDATA                   "infinity">

<!ELEMENT HeadType              EMPTY>
<!ATTLIST HeadType
        TYPE                    CDATA                   "arrow"
        FILLED                  (true|false)            "true"
        SIZE                    CDATA                   "10">


<!- END AxonType ->
```

A.4. The ConceptMap DTD

```
<!- ConceptMap ->

<!ELEMENT ConceptMap            (Background,
                                 Dimension,
                                 NeuronStyle*)>

<!- END ConceptMap ->



<!- Background ->

<!ELEMENT Background            EMPTY>
<!ATTLIST Background
        COLOR                   CDATA                   "white">

<!- END Background ->
```

```
<!- Dimension ->

<!ELEMENT Dimension              EMPTY>
<!ATTLIST Dimension
        WIDTH                    CDATA                  #REQUIRED
        HEIGHT                   CDATA                  #REQUIRED>

<!- END Dimension ->



<!- NeuronStyle ->

<!ELEMENT NeuronStyle            (DetailedMap?,
                                  BoxStyle?,
                                  AxonStyle*)>
<!ATTLIST NeuronStyle
        ID                       ID                     #REQUIRED
        NEURONURI                CDATA                  #REQUIRED>


<!ELEMENT DetailedMap            EMPTY>
<!ATTLIST DetailedMap
        MAPURI                   CDATA                  #REQUIRED>


<!ELEMENT BoxStyle               (Dimension,
                                  Position,
                                  Title,
                                  DataTagsStyles?,
                                  Line?)>

<!ELEMENT Position               EMPTY>
<!ATTLIST Position
        X                        CDATA                  #REQUIRED
        Y                        CDATA                  #REQUIRED>


<!ELEMENT Title                  (#PCDATA)>


<!ELEMENT DataTagStyles          (DataTagStyle*)>

<!ELEMENT DataTagStyle           EMPTY>
```

```
<!ATTLIST DataTagStyle
        NAME                    CDATA                   #REQUIRED>


<!ELEMENT AxonStyle             (DataTagStyles?,
                                 Line)>
<!ATTLIST AxonStyle
        AXONID                  NMTOKEN                 #REQUIRED
        NEURONSTYLE             IDREF                   #REQUIRED>


<!ELEMENT Line                  (Position*)>


<!- END NeuronStyle ->
```

Glossary of terms

Application Programming Interface (API)

A set of classes and functions for performing a certain task, while hiding the underlying details.

Abstract Windowing Toolkit (AWT)

Java GUI components implemented using platform-specific code, providing functionality common to all platforms.

See Also: Swing.

Centre for user-oriented IT design (CID)

An inter-disciplinary competence centre at KTH. Activities include a three-part collaboration between IT-industry, user organizations and university researchers. See http://www.nada.kth.se/cid/.

Cabinet Format (CAB)

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

Common Object Request Broker Architecture (CORBA)

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 http://www.omg.org

Distributed Component Object Model (DCOM)

Microsoft's extension of their Component Object Model (COM) to support objects distributed across a network. DCOM has been submitted to the IETF (http://www.ietf.org/) as a draft standard. Since 1996, it has been part of Windows NT and is also available for Windows 95.

DocBook

DocBook is a very popular set of tags for describing books, articles, and other prose documents, particularly technical documentation. DocBook is defined using the native DTD

syntax of SGML and XML. Like HTML, DocBook is an example of a markup language defined in SGML/XML.

Document Object Model (DOM)

From the DOM spec [domlevel1]:

> 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

Document Type Definition (DTD)

A text file that defines the allowed elements of a markup language such as HTML or SGML- and XML-based markup languages.

File Transfer Protocol (FTP)

A common method of transferring one or more files from one computer to another.

Graphical User Interface (GUI)

The interface to an application that the user sees, and which uses graphical elements such as buttons and menus for interaction.

Hypertext Markup Language (HTML)

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.

HyperText Markup Language (HTTP)

The underlying system whereby Web documents are transferred over the Internet.

Interface Definition Language (IDL)

A language defined by the OMG used to define CORBA interfaces.

Instructional Management Systems (IMS)

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/

Java Archive (JAR)

A compressed bundle of Java classes, similar to ZIP files. It is used to distribute Java applications and their related files.

Java Foundation Classes (JFC)

A standard package of extensions to the base Java libraries. The most significant part is called Swing.

Kungliga Tekniska Högskolan (KTH)

Kungliga Tekniska Högskolan, i.e., The Royal Institute of Technology, located in Stockholm, Sweden.

Lightweight Directory Access Protocol (LDAP)

A protocol for accessing on-line directory services. LDAP defines a relatively simple protocol for updating and searching directories running over TCP/IP. See RFC 1777 (http://www.ietf.org/rfc/rfc17

Meta Object Facility (MOF)

As described in the MOF Spec [mofspec]:

> 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.

Open DataBase Connectivity (ODBC)

A standard for accessing different database systems. An application can submit statements to ODBC using the ODBC flavor of SQL. ODBC then translates these to whatever flavor the database understands.

Open JVM Integration (OJI)

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.

Object Management Group (OMG)

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.

Object Modeling Technique (OMT)

The predecessor of UML.

Path URN

A URN scheme that defines a uniformly hierarchical name space. This URN scheme supports dynamic relocation and replication of resources. DNS technology is used to resolve a path into sets of equivalent baseURIs, and then one URI is resolved into the named resource. See [pathurnspec].

Remote Method Invocation (RMI)

Part of the Java programming language library which enables a Java program running on one computer to access the objects and methods of another Java program running on a different computer.

Standard Generalized Markup Language (SGML)

A standard for describing markup languages. Used to define both XML and HTML.

Swing

A set of Java GUI components extending the original AWT. Part of JFC.

Unified Modeling Language (UML)

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 [umlref] and the UML home page (http://www.omg.org/uml/).

Uniform Resource Identifier (URI)

As described in [rfcuri], a URI is "a compact string of characters for identifying an abstract or physical resource".

Uniform Resource Locator (URL)

As described in [rfcuri], 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.

Uniform Resource Naming (URN)

A URI scheme which is under development by the IETF (http://www.ietf.org/), which should provide for the resolution using Internet protocols of names which have a greater persistence than that currently associated with Internet host names or organisations (as used in URLs). Uniform Resource Names will be URI schemes that improve on URLs in reliability over time, including authenticity, replication, and high availability. See http://www.w3.org/pub/WWW/Addressing/

See Also: Path URN.

UML Exchange Format (UXF)

A set of XML DTDs that describe UML diagrams. See [uxfspec].

The World Wide Web Consortium (W3C)

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."

Wireless Application Protocol (WAP)

A lightweight protocol to allow mobile phone users to use Internet services.

XML Metadata Interchange (XMI)

A serialization of a UML metamodel described using the MOF. The serialization is done using XML. See [xmispec].

Extensible Markup Language (XML)

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 [xmlspec].

Extensible Style Language (XSL)

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

W3C (http://www.w3c.org/) Specifications

[domlevel1] Document Object Model (DOM) Level 1 Specification, V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, L. Wood, REC-DOM-Level-1-19981001 (http://www.w3.org/TR/REC-DOM-Level-1/).

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

OMG (http://www.omg.org/) Specifications

[corbarelationships] CORBAServices: Common Object Services Specification, The Object Management Group, formal/98-12-09 (http://www.omg.org/cgi-bin/doc?formal/98-12-09).

[mofspec] Meta Object Facility (MOF) Specification, The Object Management Group, ad/97-08-14 (http://www.omg.org/cgi-bin/doc?ad/97-08-14).

[xmispec] XML Metadata Interchange (XMI), The Object Management Group, ad/98-10-05 (http://www.omg.org/cgi-bin/doc?ad/98-10-05).

IETF (http://www.ietf.org/) Documents

[rfcuri] Uniform Resource Identifiers (URI), T. Berners-Lee, R. Fielding, L. Masinter, RFC 2396 (http://www.ietf.org/rfc/rfc2396.txt).

[rfcmime] Multipurpose Internet Mail Extensions (MIME) Part One, N. Freed, N. Borenstein, RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt).

[pathurnspec] The Path URN Specification, Daniel LaLiberte, Michael Shapiro, draft-ietf-uri-urn-path-01.txt (http://www.hypernews.org/~liberte/www/path.html).

Other Specifications

[uxfspec] Making UML models exchangeable over the Internet with XML: UXF approach, J.

Suzuki, Y. Yamamoto, http://www.yy.cs.keio.ac.jp/~suzuki/project/uxf/, 1998.

[imsmetadata] IMS Meta-Data Specification, Version 1.1 (http://www.imsproject.org/metadata/index.html), IMS Project (http://www.imsproject.org/) .

CID (http://www.nada.kth.se/cid/) reports

[cid17] The Garden of Knowledge as a Knowledge Manifold: A Conceptual Framework for Computer Supported Subjective Education, A. Naeve, CID-17 (http://cid.nada.kth.se/sv/pdf/cid_17.pdf), TRITA-NA-D9708, KTH, Stockholm, 1997.

[cid18] Kunskapens Trädgård, R. Linde, A. Naeve, K. Olausson, K. Skantz, B. Westerlund, F. Winberg, K. Åsvärn, CID-18 (http://cid.nada.kth.se/sv/pdf/cid_18.pdf), TRITA-NA-D9708, KTH, Stockholm, 1998.

[cid49] IT-baserade matematikverktyg: några tidigare och några pågående KTH-projekt, A. Naeve, Centre for user-oriented IT-Design, CID-49 (http://cid.nada.kth.se/sv/pdf/cid_49.pdf), TRITA-NA-D9907, KTH, Stockholm, 1999.

[cid52] Conceptual Navigation and Multiple Scale Narration in a Knowledge Manifold, A. Naeve, CID-52 (http://cid.nada.kth.se/sv/pdf/cid_52.pdf), TRITA-NA-D9910, KTH, Stockholm, 1999.

[cid53] Conzilla: Towards a Concept Browser, M. Nilsson, M. Palmér, CID-53, TRITA-NA-D9911, KTH, Stockholm, 1999.

Other Works

[umlref] The Unified Modeling Language: A Reference Manual, G. Booch, I. Jacobsson, J. Rumbaugh, Addison Wesley Longman Inc., 1999.

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

[wilson] Concilience: The Unity of Knowledge, Edward O. Wilson, Knopf, April 1998.