Master's Thesis in Telematics
for the Award of the Academic Degree
Diplom Ingenieur
at
Graz University of Technology

Aspects of Structured Component Spaces in Distributed Systems

---

submitted by:

Thomas Oberhuber May 2004

Institute for Information Systems and Computer Media

Supervisor: Univ.-Doz., Dipl.-Ing., Dr.techn. Klaus Schmaranz

Diplomarbeit aus Telematik
zur Verleihung des des Akademischen Grades
Diplom Ingenieur
an der
Technischen Universität Graz

Aspekte komplexer Komponentenstrukturen in verteilten Systemen

---

vorgelegt von:

Thomas Oberhuber Mai 2004

Institut für Informationssysteme und Computer Medien

Begutachter: Univ.-Doz., Dipl.-Ing., Dr.techn. Klaus Schmaranz

Abstract

Interconnecting information is one of the most important matters in modern information technologies. Mapping semantic relationships to a document structure properly is hardly never a trivial task. In many cases such relations are hyper dimensional; i.e. they cannot be described by simple, untyped one-to-one links.

This thesis deals with structured component spaces in distributed systems using hyper dimensional relations. This kind of relations is a key feature of *Dinopolis* – a distributed component framework. The practical work, this thesis is based on, was part of the development of *Dinopolis*.

The links in *Dinopolis* are called *Interrelations*. *Interrelations* are addressable, typed link-objects which provide for modeling robust, hyper dimensional relationships between objects of arbitrary type. Further these objects can be arbitrarily distributed over the network. This thesis presents the design as well as some aspects of the implementation of these *Interrelation* objects.

Kurzfassung

Das Verknüpfen von Informationen ist eine der wichtigsten Anforderungen an moderne Informationstechnologie. Semantische Beziehungen zwischen Daten in entsprechende Dokumentstrukturen umzusetzen, ist eine oft unterschätzte Aufgabe. Häufig sind solche Relationen hyperdimensional. Das heißt, sie können nicht durch einfache, untypisierte eins-zu-eins Links beschrieben werden.

Die vorliegende Diplomarbeit behandelt die Modellierung komplexer Komponentenstrukturen in verteilten Systemen mittels hyperdimensionaler Relationen. Derartige Relationen sind ein zentrales Charakteristikum von *Dinopolis* – einem verteilten Komponenten-System. Die praktische Arbeit, auf der diese Diplomarbeit basiert, wurde im Rahmen der Entwicklung von *Dinopolis* absolviert.

Interrelations, wie Links in *Dinopolis* genannt werden, sind adressierbare, typisierte Link-Objekte, welche der Modellierung robuster, hyperdimensionaler Objekt-Beziehungen dienen. Dabei können die zu verknüpfenden Objekte beliebigen Typs sowie beliebig über das Netz verteilt sein. Diese Arbeit präsentiert sowohl Design als auch Aspekte der Implementierung dieser *Interrelation*-Objekte.

Acknowledgments

First of all I want to thank my family, especially my parents, for supporting me all the time. Without them it would not have been possible to finish my studies. Further I want to thank all my friends just for being my friends.

Special thanks go to Klaus Schmaranz who enabled me to work on this thesis and who was a critical but fair reviewer of this thesis. I always enjoyed the lively discussions we had. I also want to thank all members of the DINO team for supporting me all the time.

I have to express my special gratitude to Stefan Thalauer for the delightful collaboration throughout the last two years. He became one of my best friends in that time.

Last but not least I want to thank Edmund Haselwanter and Jürgen Malin for proof-reading and especially for being the friends they are.

I hereby certify that the work reported in this thesis is my own and that work performed by others is appropriately cited.

Signature of the author:

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten Anderer unverändert oder mit Abänderungen entnommen wurde.

Table of Contents

Abstract i

Kurzfassung ii

Acknowledgments iii

1 Introduction 1
- 1.1 Chapter Overview 4

2 Motivation 6
- 2.1 A Real World Scenario 6
- 2.2 Key Features of the Dinopolis System 7
- 2.3 Why Interrelations? 9
- 2.4 Key Features of the Interrelation Mechanism 11

3 Background and History 14
- 3.1 History 14
  - 3.1.1 Hypertext and Hypermedia 15
  - 3.1.2 Component and Object Relationships 17
- 3.2 Link Concepts in Existing Information Systems 17
  - 3.2.1 The Web 17
  - 3.2.2 XML – XLink, Xpath, XPointer 19
  - 3.2.3 Hyper-G 20
  - 3.2.4 The History of *Dinopolis* 23

4 Interrelations in Dinopolis **26**
4.1 The Object Management Module . . . 27
4.1.1 Dinopolis Objects . . . 28
4.1.2 The Dynamic Type Mechanism . . . 29
4.2 The Address Management Module . . . 31
4.3 Requirements to the Interrelation Mechanism . . . 32
4.3.1 User Requirements . . . 32
4.3.2 Software Requirements . . . 34

5 Interrelations in Detail **37**
5.1 Interrelation Management Module . . . 37
5.2 Internal Structure of Interrelations . . . 37
5.3 Design Overview . . . 38
5.4 Interrelation Manager . . . 39
5.4.1 Lifecycle of an Interrelation . . . 40
5.4.2 Dynamic Type Mechanism concerning Interrelations . . . 40
5.5 Interrelations . . . 40
5.5.1 Types of Interrelations . . . 43
5.5.2 Endpoints (of Interrelations) . . . 43
5.5.3 Endpoint Container . . . 44
5.5.4 *EndPoint Classification Sets* . . . 45
5.5.5 Metadata . . . 45
5.5.6 Data held by an Interrelation . . . 45
5.6 The Interrelation Part of Dinopolis Objects . . . 46
5.6.1 Dinopolis Object side Endpoint . . . 47
5.6.2 The Dinopolis Object side Endpoint Container . . . 48
5.6.3 The Interrelation Handler . . . 49

6 Endpoint Classification Sets **51**
6.1 Concept . . . 51
6.2 An Example . . . 53
6.3 Design of *EndPoint Classification Sets* . . . 56
6.3.1 The Internal Structure of an Endpoint Classification Set . . . 57
6.3.2 The interface of *EndPoint Classification Sets* . . . 57
6.4 Set Container . . . 58

7 Processes and Algorithms **59**
7.1 Creating an Interrelation . . . 59
7.2 Attaching a Dinopolis Object to an *Interrelation* . . . 60
7.3 Detaching an *Endpoint* from an *Interrelation* . . . 62

7.4 Resolve 62
7.4.1 Request a Selection of Endpoints from an Interrelation 62
7.4.2 A more specific Resolve Operation 63

8 Summary and Outlook **65**

Bibliography **66**

References **66**

Glossary **69**

List of Acronyms **71**

Index **73**

List of Figures

3.1 The collection hierarchy – A directed acyclic graph 21
3.2 A document viewer first requests a document and then some links 25

4.1 Overall Dinopolis System Architecture 27
4.2 Dinopolis Object Proxy 29
4.3 The Dinopolis Object's Internal Structure 30
4.4 Static and Dynamic Types. 31

5.1 An Interrelation's logical-view 39
5.2 An Interrelation's Internal Structure 41
5.3 The *Dinopolis Object's Internal Structure* 46

6.1 Intersection of two sets 52
6.2 Union of two sets 52
6.3 Difference of two sets 53
6.4 Some documents that coud be connected via an Interrelation 53
6.5 select English destination document of final version 54
6.6 Intersection of three sets 54
6.7 Combination of union and intersection. 55

Chapter 1

Introduction

This master's thesis is focused on relations between arbitrary "objects" in general and on hyper-dimensional interrelations between Software components in detail. The thesis was authored in the course of the development of the *Dinopolis Middleware System*. This framework is developed by the Dino *Development Group* at the IICM (Institute for Information Systems and Computer Media), Graz University of Technology in cooperation with the DLR (Deutsches Zentrum für Luft und Raumfahrt (German Aerospace Center)). I joined the development at the beginning of the detailed design phase. The development of the single Modules of the DINO system was managed by small teams of two up to four members. I worked on the *Interrelation Management Module* mainly with Stefan Thalauer who was also working on his master's thesis. For a certain amount of time Gernot Hoebenreich participated in our team.

Since all the work on this thesis was done in close collaboration with Stefan Thalauer some considerations concerning *Interrelations* (especially in Chapter 4 and Chapter 5) can be found in a similar form in his thesis.

The head of the development team is Klaus Schmaranz. Development was started 'from scratch', based on the experiences with Hyperwave as well as with a Java library called Dino, which was presented at CeBit 1999 as the core platform of MTP (*Medical Telematics Platform*). *Dinopolis* is designed to be platform as well as programming language independent and aims integration of existing technologies and adheres to common standards.

As described at `http://www.dinopolis.org` *Dinopolis* is a massively distributable componentware framework. *Dinopolis* supports component-based development of arbitrarily distributed applications, either from scratch or by integrating and combining existing systems into a distributed environment. Further *Dinopolis* is designed to provide an abstraction on top of a heterogenous environment to build complex, distributed

business applications. Its layered architecture and its highly modular structure together with component technology turn it into a generic, flexible and extensible framework which can be adapted for all of today's common distributed application scenarios.

Aspects that come into play when developing such a powerful middleware system are the following:

- A completely platform and technology independent design which allows implementations in many different programming languages. Currently *Dinopolis* is implemented in C++ but implementation in other programming languages are thinkable. Therefore this thesis refers to C++ when talking about implementation details.
- A highly modular system design which allows loading of special modules like logging or versioning on demand. This provides for a very slim kernel.
- Dynamically typeable components. This allows extending components at runtime, which is one of the most astonishing features of *Dinopolis*.
- Complete separation of addressing and navigation. This is very important, since mixing up these two semantically completely independent aspects is one of the most problematic aspects of existing systems.
- A robust, highly sophisticated addressing mechanism which provides globally unique handles for each component. These handles have to be stable even in the case of moving components between instances of *Dinopolis* systems. See [Schmaranz, 2002a] for details.
- A highly flexible, runtime configurable component model allowing to distribute components transparently across the network.
- Arbitrary multi directional, hyper-dimensional, typed *Interrelations* between components.
- Full meta-information support for all components as well as for *Interrelations*.
- Full support of distributed security features. Since all components are accessible remotely the security mechanisms have to deal with this fact too.
- Elaborate access control mechanisms are of course an essential feature of each distributed system. *Dinopolis* as a middleware framework has to handle multiple different access policies depending on the systems it integrates.
- Operations have to be transaction based since there is never a guarantee that they cannot fail under certain circumstances. So a system like *Dinopolis* has to ensure at least that the system's state remains consistent after a failed operation.

Dinopolis as a middleware framework provides for integration of various "external systems" like filesystems, databases or even rather complex ones like full-blown hospital organization systems. Since these *External Systems* may provide rather different, arbitrarily complex functionality, this functionality is encapsulated within *Distributed Objects* called *Dinopolis Objects*. A *Dinopolis Object* is addressable by a GUH (Globally Unique Handle) which is stable during the whole lifetime of the *Dinopolis Object* (see [Bluemlinger et al., 2003] for details).

Another key feature of *Dinopolis* is the so called *Dynamic Type* mechanism. Dynamic typing means that the type of a *Dinopolis Object* is extensible at runtime. It does not mean that types are exchangeable without limitations. Each *Dinopolis Object* has its *Static Type* that – as the name says – is bound statically and doesn't change during an object's life cycle. Type "extensions" in the sense of dynamic typing are realized by so called *Definitions* which are small program units and represent *Components* in the sense of *Software Components*. Since newly attached *Definitions* may override already existing implementations – either coming from the *Dinopolis Object*'s *Static Type* or from other *Definitions* – this mechanism can be seen as sort of inheritance at runtime. For further information on this *Dynamic Type* mechanism refer to [Haselwanter, 2003].

Dinopolis Objects interact. They refer to each other. A *Dinopolis Object* may refer to a certain part of another *Dinopolis Object*. *Dinopolis Objects* may build arbitrary complex structures. These structures must not be limited by system borders. All these features are modeled by *Interrelations*. For details refer to Chapter 4.

Since *Dinopolis* is a distributed object framework, *network transparency* is a very important aspect. Therefore it is necessary that applications can access *Dinopolis Objects* in a unique way which has to be independent from the object's location. This can be realized by globally unique addressing and an elaborate proxy mechanism.

Finally *Dinopolis* provides a highly sophisticated, role based security concept which allows to intercept each operation on a *Dinopolis Object*.

1.1 Chapter Overview

Chapter 2 Motivation:
Relations between documents or even arbitrary objects are an important part of every modern information system. Since these relations are generally far from being trivial, modeling them isn't an easy task too. Features like stable links, subset references, and hyper dimensional relations are needed to model all thinkable dependencies. Further, *Interrelations* have to be typed. Another requirement concerning the usability of *Interrelations* is meta information. Existing systems do not provide all of these items in a satisfying way. Therefore *Dinopolis* aims to fulfill all of these requirements.

Chapter 3, Background and History:
Relations between objects and documents have been an important topic since the first information systems were developed. Even the first thought experiments concerning information management (like Vannevar Bush's *memex*) were heavily based on a concept we call *links* nowadays. During the last decades various information systems were developed. Some of them are theoretically interesting (e.g. Ted Nelson's *Xanadu*), some of the are just extremely successful and widespread (especially *The Web*). This chapter gives an overview.

Chapter 4, Interrelations in Dinopolis:
The *Interrelation* mechanism that is presented in this thesis is a core feature of the *Dinopolis* middleware framework. Of course this is only one of a whole bunch of characteristics of *Dinopolis*. Since the *Dinopolis* design is very modular all basic features are addressed by different kernel modules. This chapter outlines some fundamental kernel modules in quite briefly.

Interrelations are a useful mechanism for application programmers as well as for end users of applications based on the *Dinopolis* framework. Therefore *Interrelation* design has to fulfill requirements on two different levels of abstraction. Consequently this chapter describes *user requirements* as well as *software requirements*.

Chapter 5, Interrelations in Detail:
Dinopolis is a very modular system. Not only that additional fancy functionality can be introduced by adding appropriate modules. The base functionality of the system is provided by modules too. These modules are called *kernel modules* and one of them is the *Interrelation Management Module*. This chapter presents this module and all its submodules – including the *Interrelation* object.

Chapter 6, Endpoint Classification Sets:
One of the core feature of *Interrelations* is the possibility to establish hyper-dimensional **n:m:l:k:...** relationships. This represents a very powerful tool to model semantics of a document structure. Since this outstanding feature can be seen as the core of this thesis a whole chapter is dedicated to it.

Chapter 7, Processes and Algorithms:
The *Interrelation Management Module* heavily interacts with other kernel modules of the *Dinopolis* system; especially with the *Object Management Module*. This includes some rather sophisticated processes. Further, some of the *Interrelation* internal processes are also worth mentioning. This chapter outlines just the most interesting ones.

Chapter 8, Summary and Outlook:
At the moment the final phase of development of the *Dinopolis* middleware framework is reached. In the near future a prototype of an application based on *Dinopolis* will be implemented. This will be the first practical test for the system. Especially concerning *Interrelations* the next moths and years will be very interesting since nobody knows if users will exploit the new features that *Interrelations* provide.

Chapter 2

Motivation

This chapter outlines the problems *Dinopolis* aims to solve by means of a fictitious but realistic scenario. Further it describes which key features of *Dinopolis* help to deal with these issues. Finally Section 2.3 presents an overview about *Dinopolis*' improvements concerning *Interrelations*.

2.1 A Real World Scenario

This scenario describes one of several fields of applications *Dinopolis* is defined for, namely a medical information system that interrelates arbitrary information during a whole anamnesis. In the following sections will refer to this example when outlining the fundamental requirements of a system that is able to manage the tasks.

Consider a patient who is suffering from a heart disease. Further think of him having a heart attack while spending his holidays abroad. After he was found in his hotel room an ambulance with an emergency doctor on board arrives at the scene. The doctor has a special gadget in his disposal that allows him to connect to the local hospital's medical information system. Further, this system is interconnected with an international medical information system which provides security checked lookups of patient records.

The doctor – knowing the patient's identity – now is able to request the appropriate patient record. He doesn't have to deal with any complications caused by the patient being a foreigner. Fetching the appropriate information is processed completely transparently. After a successful request the medic knows the patient's complete anamnesis including all information about his heart disease. This enables him to make an exact initial diagnosis on site. Moreover, he is able to append his diagnosis to the record

immediately.

At the hospital all necessary preparations can be done based on the emergency doctor's initial diagnosis. Further the hospital staff can request the patient's record themselves and study it in detail. They can investigate all relevant information from birth to present. During the patient transport all essential examinations like CT (Computer Tomography) or even a potentially necessary surgery could be prepared.

After the patient is transfered to his home town's hospital the staff there automatically has access to the updated patient record. So there is no need for expensive and bureaucratic call backs to the foreign hospital.

2.2 Key Features of the Dinopolis System

The scenario sketched above outlines some of the requirements a very special but full-blown information management system has to fulfill. Since no existing component framework satisfies all needs for building such an information system, *Dinopolis* was developed to fill this gap. Even if it is not in the scope of this thesis to present the design of the *Dinopolis* middleware framework in depth, this section shows that *Dinopolis*' general approach meets all requirements arising from a scenario as outlined in Section 2.1.

On the basis of this example we can define the features a general platform like *Dinopolis* has to provide. Always keep in mind that the main purpose of such a middleware system is to allow to write applications that are based on transparently distributed components and objects. In our example we want to design and implement a virtual, distributed, electronic patient record.

- The way an object is accessed has to be independent from the object's location on the network. Further it must not be influenced by the type of external system (e.g. database, filesystem, ...) the object is stored to. Therefore *Dinopolis* has to provide a completely transparent accessed mechanism.

- Object movement must not influence the virtual address of that object. Consequently *Dinopolis* has to provide an addressing mechanism that is based on globally unique handles which are robust against object movement. (see Section 4.2)

- It must be possible to compose virtual patient records from many objects of arbitrary type. Therefore an elaborate inclusion mechanism has to be provided. *Dinopolis* implements composition by providing arbitrarily typed *Interrelations*. These *Interrelations* have to be robust against object movement too. With this mechanism it is possible to implement inclusions, bookmarks, hyperlinks, nav-

igation mechanisms and any other relationships that can be modeled by hyper dimensional *Interrelations*. (see Chapter 4)

- Another feature that eases the organization of objects is the possibility to add arbitrary descriptive and administrative *Metadata*. Such *Metadata* could be the content type of some data encapsulated in an object or an object's creation date. (see Section 4.1)

- It must be possible to access the content of an object in a uniform way. It is not sufficient to provide a raw stream to that content and leave its treatment to the applications. Therefore it is preferable to provide a specialized content handler that provides a unified interface. (see Section 4.1)

- An object isn't restricted to providing passive data. It may also be a wrapper of dynamic information or even external devices. In our example this could be a MRI (Magnetic Resonance Imaging) device. Therefore objects in *Dinopolis* provide so called operations. Operations provide an interface applications can use to access the external data or device respectively. Adding operations to an object makes it a *component* in the sense of componentware. (see Section 4.1)

These are only the most obvious and most important requirements to a middleware framework like *Dinopolis* that can be deduced from a simple real world scenario like the one sketched above. But even these few requirements reveal that the system may become arbitrarily complex if we do not allow for a simple and clear design. This implies a very modularized system based on a flexible, adaptable, runtime reconfigurable and network transparent software architecture.

Another important aspect when talking about a complex system like *Dinopolis* is abstraction. Similar to the object oriented programming paradigm, that allows to reduce complexity by hiding implementation details, the components *Dinopolis* provides, introduce a new abstraction level. They enable applications to communicate with some arbitrary objects, external devices, or even complex external systems via a clear interface. This interface is built up by operations and services of a *Dinopolis Object*. Of course this is much more convenient than communicating directly via a complex, specialized API.

Based on these considerations the design of *Dinopolis* has to fulfill the following general requirements:

- Dependencies from a certain platform or a special technology have to be avoided by all means. Therefore the design and system architecture have to be platform and technology independent.

- According to the KISS principle the design has to be as rich as necessary but as

slim as possible. Consequently the *Dinopolis* system kernel is very slim. But of course it is extensible by loadable modules.

- Even if *Dinopolis* is a rather lightweight framework it has to provide for implementation of very complex applications like the electronic patient record described in our example. This implies that *Dinopolis* must include a mechanism to combine arbitrary data, metadata, whole objects, and components. This mechanism has to provide a unified but extensible interface. This is best done by addressable *Interrelation* objects.

We can see that the first two items are very general and apply to all mature software designs. The third point, in contrast, is rather special for a middleware framework. Therefore we will have a look at the need for such a mechanism in more detail.

2.3 Why Interrelations?

Interconnecting information is one of the most important matters in modern information technologies. Mapping semantic relationships to a document structure properly is hardly never a trivial task. In many cases such relations are hyper-dimensional, i.e. they cannot be described by simple, untyped, one-to-one links. Further it is obvious that every single relationship between two documents is bidirectional, since each reference from *document A* to *document B* implies that *document B* is referenced by *document A*. Although you don't need to be much smarter than the average bear to comprehend this fact, modeling these multi directional links is not trivial at all. Further it is often desirable to annotate interrelations. Therefore arbitrary metadata (text, images, ...) may be useful in many cases. Another important point when thinking about interconnecting information are types of relations. A relation isn't just a relation. It can be a bookmark, a unidirectional one-to-one link, a multi directional n:m link or even a hyper-dimensional n:m:l:k... relation. Of course technically similar relations can also be of different types because of different semantics. Therefore distinction between arbitrary types is an important feature of a modern interrelation implementation.

Another very important aspect of relations is robustness. It must never happen, that a link becomes invalid when moving the destination object. But this is exactly what happens when moving a document in classical hypermedia environments. The major reason for this undesired behavior is the mixture of addressing and navigation in such systems. This also applies to all filesystems currently used. So separating addressing and navigation is an inevitable condition for robust interrelations.

Classical HTML *Hyper Links*, as well as most other link implementations, do not fulfill most of the requirements described above. The only relation that can be modeled by a

simple *Hyper Link* is a unidirectional, one to one reference. Furthermore a *Hyper Link* has no semantics. Therefore *Hyper Links* are a very poor tool for modeling complex relationships between arbitrary documents. For this reason several alternatives have been developed during the last few years. But mostly all of them fail to provide at least one important feature needed when modeling arbitrary relationships. For details see Section 3.2. *Dinopolis* aims to fill this gap.

2.4 Key Features of the Interrelation Mechanism

As we have seen in the previous section *Interrelations* have to fulfill several very important tasks within the *Dinopolis* middleware framework. Consequently the following key features can be outlined:

- **An *Interrelation* allows for connecting arbitrary many objects of arbitrary type.**

  Connecting two objects is a well known feature of all link mechanisms. But this mechanism is by far too weak for modeling complex relations between a various number of pieces of information or objects. Therefore *Dinopolis Interrelations* are not limited in respect to the number of objects they connect.

- ***Interrelations* are always multi directional.**

  To avoid the well known problem of so called dead links, *Interrelations* provide a mechanism that allows each object to know which *Interrelations* it is attached to at any time. So there is no danger that an object tries to follow a link to another object that doesn't exist anymore.

- ***Interrelations* are robust against object movement.**

  Dead links do not only result from object deletion but can also be caused by cause is object movement. The reason for this problem is that most known systems are mixing up addressing and navigation. Consequently *Dinopolis* solves this problem by a clear separation of addressing and navigation.

- **Objects attached to an *Interrelation* can be grouped together semantically.**

  It isn't enough to simply connect a bunch of objects when we want to model complex relationships between them. It is necessary that we can denote *how* the objects correlate. Therefore *Interrelations* provide for grouping objects within sets of semantically corresponding elements. For example one could want to denote the sources and the destinations of a directed n:m dependency.

- **The semantic grouping of connections can be changed at runtime.**

  The meaning of an object in the context of an *Interrelation* is not static. Therefore it is possible to move an object from one semantic set to another at runtime.

- **Objects can belong to multiple semantic sets at a time.**

  Objects are not restricted to having a single meaning in the context of an *Interrelation*. Therefore an object can participate in several groups of objects with different meaning. For example in the context of an *Interrelation* modeling a multi

lingual relationship between several versions of documents an object may be in the set of English documents as well as in the set of up-to-date documents.

- **It is possible to interconnect parts of *Dinopolis* Objects.**

  Often it is not sufficient to refer to an object as a whole. For example it may be necessary to point to a certain section of a document. The *Dinopolis Interrelation* mechanism makes it possible to relate to a certain part of an object.

- ***Interrelations* can be arbitrarily typed.**

  Interrelations can be used to model various relationships between objects. This has to lead to distinguishable types of *Interrelations*. For example an ordinary hyperlink is semantically completely different from an inclusion of an image within a document. This is true even if they are technically similar.

- **New types of *Interrelations* can be introduced at runtime.**

  As one can not know all *Interrelation* types that will be necessary during the lifetime of a *Dinopolis* system it has to be possible to register new *Interrelation* types at runtime.

- **The type of an *Interrelation* can be changed at runtime.**

  It has to be possible to configure the types of an *Interrelation* if the semantics of an *Interrelation* changes at runtime. Therefore the types of *Interrelations* must not be hard-coded.

- ***Interrelations* are uniquely addressable.**

  Since *Interrelations* have to deal with objects that are distributed arbitrarily over the network, they are addressable uniquely. This requires globally unique handles for all *Interrelations*. As we will see in Section 4.2 this is no problem at all since *Interrelations* are objects in the sense of the *Dinopolis* object model.

- ***Interrelations* can hold arbitrary meta information.**

  The more meta information is assigned to an *Interrelation*, the easier it is for humans to understand the – maybe very complex – relationship that is modeled by an *Interrelation*. Therefore *Dinopolis* provides the possibility to attach metadata of arbitrary type to each *Interrelation*.

- ***Interrelations* provide extensible functionality.**

  Modeling complex relations would be rather useless if we couldn't work with these *Interrelations*. For example we will always have a need to attach and detach objects from an *Interrelation*. Another example would be to follow a link from source to destination. Of course various further functionality is thinkable depending on the type of the *Interrelation*. Further it may be necessary to extend an *Interrelation*'s

functionality at runtime. The *Dinopolis* object model provides a mechanism to achieve all these features. For details refer to Section 4.1

- **A *Dinopolis* system can make its *Interrelations* persistent.**

  Instances of the *Dinopolis* system can be shut down and restarted from time to time. Therefore its state information – which of course includes *Interrelations* – is stored at shutdown and reloaded when started next time.

- ***Interrelations* can be moved from one *Dinopolis* system to another.**

  There are various reasons that can cause a need for moving an *Interrelation* to a different *Dinopolis* instance. For example an instance may be shut down but the *Interrelation* is still needed by objects residing in another *Dinopolis* instance. In this case *Dinopolis* provides a possibility to move the *Interrelation* to this instance.

Chapter 3

Background and History

In this chapter we will discuss some existing systems that implement technologies similar to *Dinopolis*' *Interrelations*. To be able to focus on the most important aspects of link systems we will look at the possible fields of applications at first. Based on these discussions we will examine several technologies from the well known WWW (World Wide Web) – based on HTTP (HyperText Transfer Protocol) [Berners-Lee et al., 1996] and HTML (HyperText Markup Language) [Berners-Lee and Connolly, 1995] and [W3C, 2004b] – to early versions of *Dinopolis*.

3.1 History

Interrelation-like technologies are used in several fields of application. Generally we can distinguish between two basically different applications. On the one hand *hypertext* systems make heavy use of different kinds of linkage technologies. On the other hand we have the wide area of component and object relationships.

Information systems that are based on various methods of linking are called *hypertext* or *hypermedia* systems. As we will see in the following section this technology was conceived long before computers entered everyday life. In contrast the idea of implementing *object relationships* by higher level concepts is rather new and still not very widespread.

3.1.1 Hypertext and Hypermedia

The primary idea about *hypermedia* is to create a simple, large document "containing" a gigantic amount of content. *Hypertext* in its original meaning is a collection of linked pieces of plain text. *Hypermedia* systems on the other hand may contain data of arbitrary type including images, sounds, videos, texts of course, and many more. Vannevar Bush is said to be the first who was speculating about a system like that. In an article published in 1945 (see [Bush, 1945]) he wrote:

> Consider a future device for individual use, which is a sort of mechanized private file and library. It needs a name, and, to coin one at random, "*memex*" will do. A memex is a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory.

But on the other hand he has also denoted the most important problem of such a system:

> The difficulty seems to be, not so much that we publish unduly in view of the extent and variety of present day interests, but rather that publication has been extended far beyond our present ability to make real use of the record. The summation of human experience is being expanded at a prodigious rate, and the means we use for threading through the consequent maze to the momentarily important item is the same as was used in the days of square-rigged ships.

Twenty years later the term *hypertext* was coined by Ted Nelson when he presented his idea of a system he called *Xanadu*. The system evolved over the years and was finally presented in [Nelson, 1987]. The idea was to introduce a new way of writing text. About conventional writing he notes:

> Spoken language is a series of words, and so is conventional writing. We are used to sequential writing and so we come easily to suppose that writing is intrinsically sequential. It need not be and should not be.

And therefore he suggests a new, nonsequential way of writing:

> There are two outstanding arguments for breaking away from sequential presentation. One is that it spoils the unity and structure of interconnection. The other is that it forces a single sequence for all readers which may be appropriate for none.

Nelson identifies two fundamental "tools" that are necessary to write nonsequentially.

- **Hyperlinks:** *Hyperlink* is a well known term to all internet users nowadays. But what we all know as links in the WWW is a rather truncated version of what Nelson defined as *Hyperlink*. They are unstable, untyped, uni-directed, and not extendible.

  > A link is simply a connection between parts of text or other material. Links are made by individuals as pathways for the reader's exploration; thus they are part of the actual document, part of the writing.
  > ...
  > The link facility gives us much more than the attachment of mere odds and ends. It permits fully non-sequential writing, or hypertext.

  Further he already stated a need for arbitrarily typed links:

  > A proper system should allow any types of link whatever, and there is a myriad of possible types.

- **Transclusion:** In his first approach Nelson calls this technique *inclusion* (later he preferred the term *transclusion* to underline its bidirectional character.). It describes a way of windowing documents. This means that a part of some document is virtually copied into another document. This allows for creating new documents that contain an old one without changing the original. Further it has an enormous advantage compared with the traditional "copy & paste" approach of quoting parts of documents. The quoted document parts are alway kept up-to-date because they are not copied into the new document but they are "updated" every time the new document is read.

  > Every document has an owner. The integrity of this document is maintained; no one can change it but the owner.
  >
  > But someone else may create a document which quotes it as much as desired. This mechanism we call the *quote-window* or *quote-link*. Through a "window" in the new document we see a portion of the old. We may also call this an *inclusion*.

  Another advantage of this technique is accounting. Imagine the author of a document charges each access to his or her masterpiece. If he would allow other authors to copy and paste parts of his work he would have no chance to control further read access to these parts. But with *transclusion* he would keep full control over his intellectual property.

3.1.2 Component and Object Relationships

Hypertext and *hypermedia* are describing interrelated documents containing arbitrary but *passive* content. This section describes similar concepts in the field of component frameworks and object oriented middleware frameworks. Even if the concepts are rather similar, semantics may differ tremendously because components and objects are *active*; i.e. on the one hand they can perform various operations and on the other hand – in the case of objects – they can alter their content. This leads to much more variety in modeling data representations.

Of course the concept of *Interrelations* isn't meaningful for all objects and components – if we define components to have a state at all. Nevertheless modeling relationships between objects by using explicit definitions of interrelations is very powerful. For example one could implement various datastructures on a higher – non programming language – level. Of course we need relationships of different types again.

Object relationship isn't a very widespread feature in nowadays middleware and component frameworks. CORBA defines a so called *relationship service* but as Szyperski notes in [Szyperski, 2002]

> However, the relationship service is rarely used or even implemented and is likely to be replaced by support for relationships among business objects based on CCM (CORBA Component Model).

3.2 Link Concepts in Existing Information Systems

This section sketches some concepts of interrelating documents in existing information systems.

3.2.1 The Web

The Web is definitely the most widespread information system. Its growth rate is without precedent; not only in the history of information technology. The history of the WWW ([Cailliau, 1995], [Berners-Lee, 2000a], [Gromov, 2003]) is rather interesting but definitely beyond the scope of this thesis. An overview about "The original design and ultimate destiny of the World Wide Web" by Tim Berners Lee, the "inventor" of the WWW, can be found in [Berners-Lee, 2000b].

The specification of links in HTML introduces the link concept as follows:

> HTML offers many of the conventional publishing idioms for rich text and structured documents, but what separates it from most other markup languages is its features for hypertext and interactive documents. ... the link (or hyperlink, or Web link), the basic hypertext construct. A link is a connection from one Web resource to another. Although a simple concept, the link has been one of the primary forces driving the success of the Web.
>
> A link has two ends – called anchors – and a direction. The link starts at the “source” anchor and points to the “destination” anchor, which may be any Web resource (e.g., an image, a video clip, a sound bite, a program, an HTML document, an element within an HTML document, etc.).

But apart from the amazing success of the Web its technology concerning the modeling of document relationships is rather poor. Ted Nelson notes:

> HTML is precisely what we were trying to **prevent** – ever-breaking links, links going outward only, quotes you can’t follow to their origins, no version management, no rights management.

What he criticizes are the very restricted possibilities to model relationships between documents in HTML . As described in Section 3.1.1 he has developed a system called *Xanadu* that is much more powerful concerning interrelation mechanisms. The main drawbacks of HTML links are listed below (The specification of links in HTML4.01 can be found in [W3C, 2004c]):

- HTML links are **not stable**.
  If the destination document is (re)moved the link points to nirvana.
- HTML links are **unidirectional**:
  Destination document’s do not “know” that a link points at them.
- HTML links are always **1:1 relationships**.
  There is always one source and one destination. It is not possible to model more complex relationships between documents with a HTML link.
- HTML links do not support **version management**.
  As they are hard-coded it is not possible to update the destination to a new version automatically. Further it is not possible to create HTML links that point to a whole bunch of versions of the same document (see previous point).
- HTML links do not provide **rights management**.
  It is not possible to control access to a link. An HTML links is always shown and can be followed by every single user.

- HTML links do not have **types**.
  In other words: There is only one type of HTML link. It is not possible to map different semantics of links to different link types.

- HTML links cannot be annotated by arbitrary **metadata**.
  Apart from the *title* of a link it is not possible to attach metadata. This makes it impossible to have proper descriptions for semantically complex link relationships.

So we can summarize that HTML links are a very poor concept for modeling complex relationships between documents.

3.2.2 XML – XLink, Xpath, XPointer

In its specification ([W3C, 2004a] XML (eXtensible Markup Language) is described as follows:

> The Extensible Markup Language (XML) is a subset of SGML (Standard Generalized Markup Language) that is completely described in this document. Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML.

To establish relations between XML documents the *XML Linking Working Group* has defined XLink (XML Linking Language) which they describe in the following words ([W3C, 2001]):

> XML Linking Language (XLink) . . . allows elements to be inserted into XML documents in order to create and describe links between resources. It uses XML syntax to create structures that can describe the simple unidirectional hyperlinks of today's HTML, as well as more sophisticated links.

The XLink concept is defined by the following features and restrictions:

- It is possible to model linking relationships between more than two resources.
- Links can be annotated with metadata.
- Links can reside in locations separate from the linked resources.
- The hyperlink uses URI (Uniform Resource Identifier) ([Berners-Lee et al., 1998]) as its locator technology.

- The hyperlink is expressed at one of its two ends.
- The hyperlink identifies the destination.
- Links are directed. I.e. traversal is only possible from the end where the hyperlink is expressed to the other end.
- The hyperlink does not define its effect on windows, frames, go-back lists, style sheets in use, and so on. This is determined by user agents. For example, traversal of links normally replaces the current view, perhaps with a user option to open a new window.

Further the W3C (World Wide Web Consortium) has defined the XPointer (XML Pointer Language) Framework ([W3C, 2002]), ...

> ... an extensible system for XML addressing that underlies additional XPointer scheme specifications. The framework is intended to be used as a basis for fragment identifiers for any resource whose Internet media type is one of *text/xml*, *application/xml*, *text/xml-external-parsed-entity*, or *application/xml-external-parsed-entity*. Other XML-based media types are also encouraged to use this framework in defining their own fragment identifier languages.

For defining portions of XML documents the XPointer Schema was defined. It is based on XPath (XML Path Language) and adds the ability to address strings, points, and ranges in accordance with definitions provided in DOM 2: Range (Document Object Model Level 2 Traversal and Range) ([W3C, 2000]). XPath is a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer.

With these concepts XML provides a link mechanism that is much more powerful than HTML links.

3.2.3 Hyper-G

Hyper-G (the commercial version of which is *Hyperwave*) was the first second-generation hypermedia information system. Compared with the WWW (see Section 3.2.1) *Hyper-G* introduced a lot of new concepts. One of the most important ones, especially regarding its link features, is the *Hyper-G Data Model*. In [Maurer, 1996] Hermann Maurer describes it as follows:

> Hyper-G Data Model must contain structuring elements beyond the primitive node-link model. While this can in principle be realized by specialized

link types, we chose to extend the node-link data model by introducing a particular object class, the collection.

A collection is a composite (container) object. It contains documents or other collections. It may be helpful to think of collections as directories in a file system, which may contain files or other directories. This – recursive – definition leads to the concept of a collection hierarchy (see Figure 3.1). Two restrictions apply:

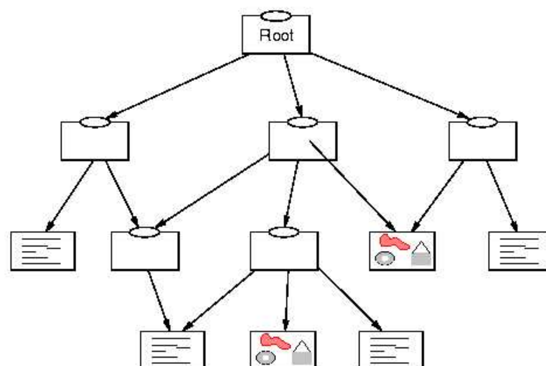

Figure 3.1: The collection hierarchy – A directed acyclic graph

1. Every document or collection must be a member of (at least) one collection (its so-called parent collection), with the exception of the server's root collection.

2. While a document or collection may be a member of more than one collection (see Figure 3.1), the collection hierarchy must be cycle-free. Technically speaking, the hierarchy is a DAG (Directed Acyclic Graph), a generalization of tree structures.

Apart from the data model *Hyper-G* provides a remarkable concept to interrelate documents. Unlike WWW it stores its links separated from the documents in a link database. This has various advantages compared to storing the links within the documents (like WWW does). Maurer notes:

- Links are bidirectional. This means it is possible to find the source from the destination and so you can navigate backwards from destination to source. What is perhaps even more interesting is that you are able to generate link maps that show both incoming and outgoing links around a certain document (like Harmony's Local Map).
- The bidirectional links allow you to guarantee link consistency. For example, if a document is removed, links pointing to it are removed as well (in Hyper-G, they are really flagged as 'open', which makes them invisible for the ordinary user, but they can still be maintained by their owners; if the document removed is re-inserted, the links become visible again). A server-server protocol even allows for maintenance integrity of the link database across server boundaries.
- Links are realized as Hyper-G objects, which contain attributes (most importantly, the anchor's Position and the LinkType). They may also be assigned keywords and are searchable ('give me all links with keyword A attached' which can be modified 'remove all links created by author B after date C') using the operations of the Hyper-G server.
- Links may be assigned individual access permissions, meaning that certain links in a document may only be visible (accessible) to certain users or user groups.
- It is possible to link all types of documents with each other, because the anchor positions are stored separately from the document itself. For example, Hyper-G can deal with links in MPEG video streams, without a need to modify or extend the MPEG standard. When showing the movie, the movie player receives both the MPEG stream and the associated anchor objects, and shows both.
- Creating and manipulating links does not require modification of the documents the link is attached to. This simplifies interactive link creation, but it also means that creating a link to or from a document does not require write permission for the document. This is useful for

annotations (which are realized as a link type in Hyper-G), where the person annotating is not necessarily the owner of the object. It is also necessary in order to attach links to read-only documents (for example data on CD-ROMs or other servers to which the Hyper-G server cannot write). A further advantage is that it simplifies automatic link generation.

As we can see, *Hyper-G*'s link concept is much more sophisticated than the HTML approach. But some features that can be rather useful when modeling document relationships are missing. For example *Hyper-G* supports *1:1 links* only which makes it rather difficult to model more complex document dependencies.

3.2.4 The History of Dinopolis

The *Dinopolis* project started in 1997 as an open source project at the IICM at *Graz University of Technology*. The first approach was to implement a massively distributable componentware system (see [Dallermassl et al., 2000b], [Dallermassl et al., 2000a] and [Objects, 97]). After an extensive design phase a JAVA prototype implementation of the system – meanwhile called DINO (Distributed Interactive Network Objects) – was presented at *CeBit* 1999. This was Dino version 2.0 which was the core of the Medical Telematics Platform prototype which was developed in cooperation with DLR. Due to the great success of this presentation and the ensuing strong interest among medical institutions the development of *Dinopolis* started based on Dino 2.0.

The core features of the early *Dinopolis* design are described in [Dallermassl et al., 2000b] as follows:

> From the very beginning four main design principles formed the credo of Dinopolis development:
>
> - Dinopolis is a very modular, extendable and platform-independent system rather than a huge proprietary monolith. The base system providing the information model is very slim.
> - Dinopolis allows to integrate and combine arbitrary systems, no matter if they are only dealing with passive content or also with active content as "real" distributed object systems do.
> - Dinopolis is an application framework that supports major industry standards. It provides access to the underlying distributed information space in a fully transparent manner for users as well as for applications utilizing it.

- Dinopolis systems can be arbitrarily distributed across the network. Nevertheless all handles used to access objects within the framework are globally unique and stable. Therefore once a handle is obtained, the object that it refers to can be moved to arbitrary locations, even across the network but the handle will still be valid.

The early versions of *Dinopolis* were implemented in *Java* since it was the best choice for platform independent development at that time (in Chapter 4 we can see that times are changing and so are the programming languages of choice). Since the kernel of the *Dinopolis* system was rather slim it was necessary to provide a mechanism to extend the system's functionality easily.

The core feature of *Dinopolis* was certainly the separated address and relation spaces which were general enough to integrate all systems that allow any kind of object addressing. In [Dallermassl et al., 2000b] this concepts are outlined as follows:

> The most important design point in the Dinopolis system is to completely decouple addressing from relations and content. Addresses in Dinopolis are wrapped by handles and are used to gain access to objects. It is not possible to navigate through the address space. That's what relations are good for: relations can be of arbitrary type, they can exist between arbitrary objects and they allow navigation in the resource space.

[Blümlinger, 2000] describes the link management concept of *Dinopolis version 3.0.* It is designed as a DLS (Distributed Link Service) (see Figure 3.2 and [Carr et al., 1995]).

The most important key features are:

- **Relations are consistent**
  Existing relations have to stay valid regardless of any operation concerning the location of the objects it connects. I.e. moving an object does not influence a relation at all. When an object is deleted all relations belonging to it are removed too.

- **Relations are bidirectional**
  Bidirectional links make it easier to achieve robust links. Further they allow a straightforward integration of embedded systems that support bidirectional links themselves. And since *Dinopolis* implements all navigation issues through relations, forward and backward navigation are supported perfectly by bidirectional relations.

- **Relations have Metadata**
  Arbitrary metadata can be assigned to relations. This is necessary to model the

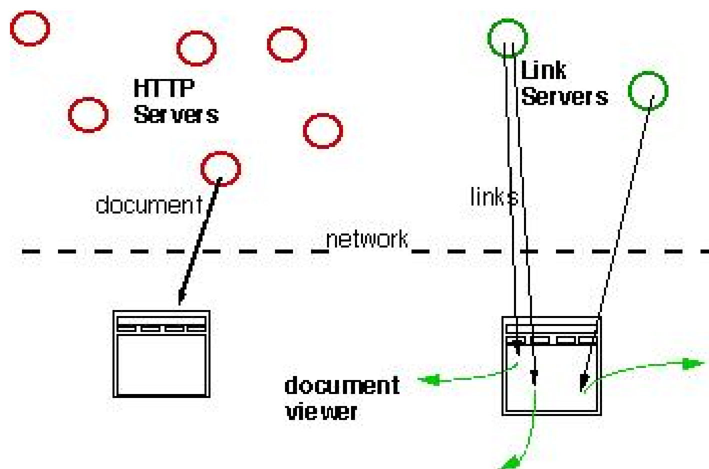

Figure 3.2: A document viewer first requests a document and then some links

semantics of a relation. For example the type of a relation is encoded into its metadata. Further metadata can contain information an application needs to interpret a relation. Display hints may also be part of a relation's metadata.

- **External link bases are supported**
  It is possible to include link bases of various systems that are embedded within a *Dinopolis* system instance. The link bases have to follow an open standard.

- **Multidimensional relations are supported**
  Dinopolis doesn't only support one-to-one relations but also one-to-many, many-to-one, and many-to-many relations. This gives a very powerful tool for authors of hypermedia documents.

- **Relations can connect relations**
  Relations do not only model relationships between objects. It is also possible to connect relations to objects or other relations. This makes it possible to introduce another level of indirection.

We can summarize that this previous version of *Dinopolis* already supported a rather mature link management. Nevertheless, some important features were missing. For example modeling a relation's semantics by simple meta information is not very powerful compared to the concepts described in Chapter 4.

Chapter 4

Interrelations in Dinopolis

The major goal of *Dinopolis* as a middleware framework is to integrate various "external" systems transparently. I.e. it enables them to share data and even communicate with each other without the necessity to know each other. Since *Dinopolis* is a distributed framework, the embedded systems can be distributed over the network too. This enables application programmers to implement software based on several different systems without caring about their integration.

Further *Dinopolis* offers additional functionality. Each piece of information in the *Dinopolis* system is uniquely addressable. This is accomplished by a highly sophisticated mechanism based on globally unique handles (GUHs). This addressing mechanism is explained in Section 4.2. Another key feature of *Dinopolis* – and as you already know this is the main topic of this thesis – are *Interrelations*. They allow to model arbitrary relationships between *everything* known to the *Dinopolis* system.

A basic technology that is necessary to implement the features described above is encapsulating every functionality within components. According to Grady Booch (see [Booch, 1987]) components can be defined as follows.

> "A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction"

Dinopolis models this concept with so called *Dinopolis Objects*. A detailed description of *Dinopolis Objects* and how they are managed can be found in Section 4.1.

To fulfill all these tasks effectively, *Dinopolis* is designed to be modular and extensible. According to the *Dinopolis* layer model all functional modules reside in the kernel of *Dinopolis*. Access to them is granted by the *Kernel Access Module*, where all security

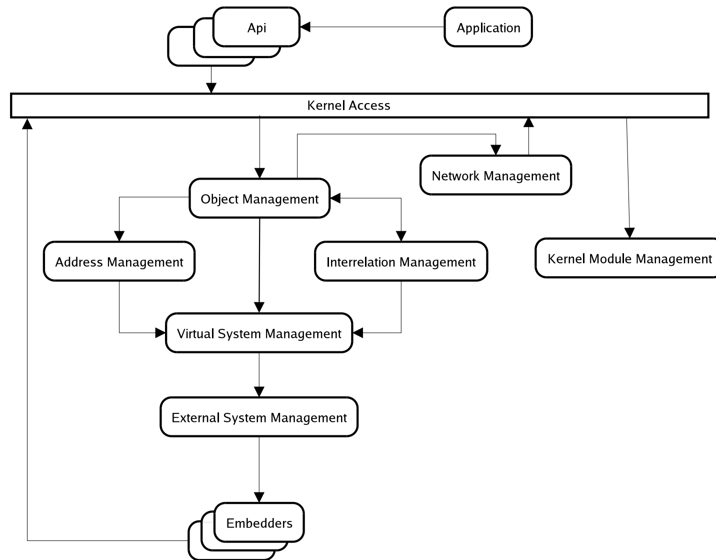

Figure 4.1: Overall Dinopolis System Architecture

checks are performed. The *Object Management* module handles everything concerning the life cycle of *Dinopolis Objects*. Addressing *Dinopolis Objects* is handled by the *Address Management Module*. The same applies to all other base functionality of the *Dinopolis* system. And, of course, the *Interrelation Management Module* is contacted for managing *Interrelations* between *Dinopolis Objects*. More kernel modules may come into play if features like transactions or versioning are used. An overview about the *Dinopolis* architecture is given in Figure 4.1. For details about the responsibilities of each module please see [Schmaranz, 2002a].

Since the functionality provided by the *Object Management* module and the *Address Management Module* are fundamental for the *Interrelation* mechanism, these modules are outlined in the following sections.

4.1 The Object Management Module

The *Object Management* module is certainly the most important module of all *Dinopolis* kernel modules. Schmaranz describes this module in his habilitation thesis [Schmaranz, 2002a] as follows:

The *Object Management* module is the central part of the *Dinopolis* System. It is involved in all operations that have to deal with the life cycle of *Dinopolis Objects*. This includes creating and deleting as well as loading, storing and rearranging objects. The *Object Management* module is always *directly* or via callbacks *indirectly* responsible for the internal structure of a *Dinopolis Object*. The term *internal structure* refers to content, meta-data, attached *Interrelations*, *Operations*, and *Services*.

4.1.1 Dinopolis Objects

Dinopolis Objects are the central units of functionality in the *Dinopolis* middleware system. Their features can be outlined by the following items:

- *Dinopolis Objects* are uniquely addressable.
- *Dinopolis Objects* are components.
- *Dinopolis Objects* encapsulate arbitrary passive or "active" content.
- *Dinopolis Objects* can be annotated by arbitrary *Metadata*. The metadata is organized by hierarchically structured keys.
- *Dinopolis Objects* provide arbitrary functionality to the user space. Functionality is either provided through *operations* or through *services*.
- *Operations* provide functionality in a way that is very similar to methods known from object oriented programming languages.
- *Services* are some kind of *active* functionality provided by *Dinopolis Objects*. This means that a *Dinopolis Object* offers a (graphical) user interface directly to applications. This allows application programmers to use complex functionality, provided by some external systems, without knowing their internal structure.
- *Operations* and *Services* can be added and removed at runtime. This is realized by the *Dynamic Type Mechanism* (see Section 4.1.2).
- Access to *Dinopolis Objects* is protected by a very sophisticated security mechanism (see Figure 4.2).
- *Dinopolis Objects* provide at least a uniform mechanism to access *Content*, *Metadata*, *Interrelations*, *Operations*, and *Services*.

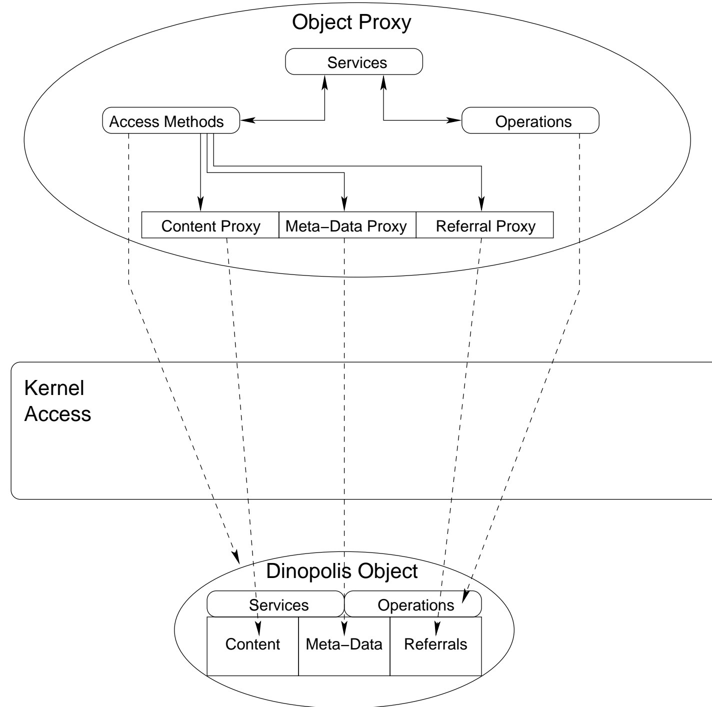

Figure 4.2: Dinopolis Object Proxy

The *Internal Structure* of a *Dinopolis Object* consisting of *Content*, *Metadata*, *Interrelations*, *Operations*, and *Services* can be seen in figure 4.3.

4.1.2 The Dynamic Type Mechanism

The functionality provided by *Dinopolis Objects* heavily depends on the *Dynamic Type* mechanism. *Dynamic Types* are used to alter the functionality provided by a certain *Dinopolis Object* at run time. Haselwanter describes this feature in [Haselwanter, 2003]:

> The functionality of *Dinopolis Objects* heavily depends on a mechanism which allows modifying the method dispatching at runtime. Since it has to be possible to add and remove functionality to/from *Dinopolis Objects* during runtime, the main task of the *Dynamic Type* mechanism is to provide

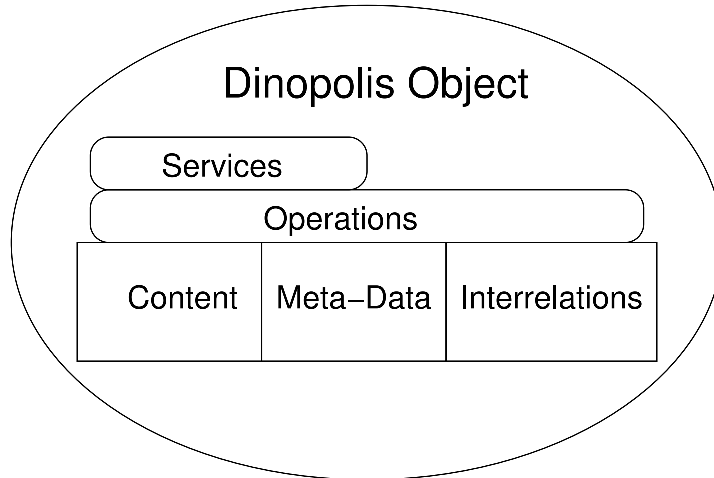

Figure 4.3: The Dinopolis Object's Internal Structure

> functionality at runtime which is normally covered by a compiler at compile time. This encloses building a virtual table for method dispatching, including scope and super calls, name-mangling and ambiguity-checking of method signatures on the *Dinopolis Object* level.

Further he notes:

> . . . the type of a *Dinopolis Object* will be split into a static and several dynamic parts. The static part includes everything to provide standardized access to the internal structure and functionality and the guaranteed base functionality of *Dinopolis Objects*. This part is called *Static Type* of the *Dinopolis Object*. On the other hand, the set of dynamic parts, which can be attached or detached to/from a *Dinopolis Object* at runtime is called the *Dynamic Type* of a *Dinopolis Object*.

Figure Figure 4.4 illustrates this behavior.

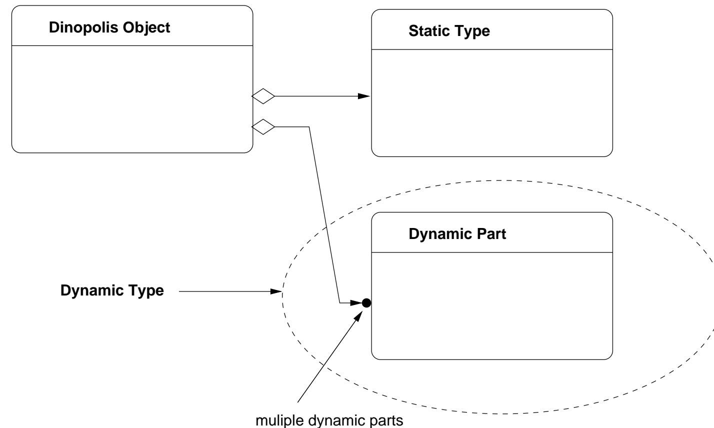

Figure 4.4: Static and Dynamic Types.

4.2 The Address Management Module

As already described in Section 4.1.1 *Dinopolis Objects* are uniquely addressable. This doesn't apply to a local instance of a *Dinopolis* system only. The uniqueness must be guaranteed globally. Therefore the "address" of a *Dinopolis Object* is called GUH.

The *Address Management Module* is responsible for providing mechanisms to generate and resolve GUHs. Generating "keys" that are globally unique is rather simple if you split the key into two parts. The first part represents some system unit – in our case this is a *Dinopolis* instance – whereas the second part is locally unique. The combination of these two parts results in a key that is guaranteed to be globally unique. Resolving these handles requires some LOOKUP SERVICE. The *Address Management Module* provides an appropriate mechanism based on an algorithm called DOLSA (Distributed Object Lookup Service Algorithm) (see [Schmaranz, 2002b] for details).

The *Address Management Module* has to fulfill a second task concerning addressing. The content of *Dinopolis Objects* is stored in so called *Virtual Systems*. Therefore the place where the data representing a *Dinopolis Object*'s content is stored to has to be addressable. The *Address Management Module* – in cooperation with the *Virtual System Management Module* is responsible for managing these addresses.

4.3 Requirements to the Interrelation Mechanism

As already described all the functionality and data a *Dinopolis* system provides is encapsulated in so called *Dinopolis Objects*. Even if these objects contain all the information that is needed to model the content of an elaborate information system they do not make much sense without some mechanism to interrelate the information. In other words: We need a possibility to model semantics.

During the last decade several link mechanisms were developed for various information systems (see Chapter 3 for an overview). *Dinopolis*' link mechanism aims to go a step beyond all existing systems – especially concerning the possibilities to model hyperdimensional relations between a number of objects. This chapter presents the general characteristics of this *Interrelation* mechanism and defines the requirements it has to fulfill. Since there are two kinds of "users", that have to deal with *Interrelations*, the requirements are separated into two sections. The *User Requirements* (see Section 4.3.1) describe the features that are required by end users of applications that are based on *Dinopolis*. The *Software Requirements* (see Section 4.3.2) define the application programmers' requirements to the *Interrelation* mechanism.

4.3.1 User Requirements

Users of *Dinopolis* based applications want to organize their data. This includes modeling component and object spaces. Therefore they need a possibility to establish relationships between documents. Of course releasing those relationships has to be possible too. Further it must be possible to interconnect various pieces of information. Realizing *transclusions* must be possible too, since this is an often needed but seldom provided feature.

Since a *Dinopolis* system, or parts of it, may be shut down from time to time, it must be possible to make those relationships persistent. Of course it also has to be possible to reload them. Further the *Dinopolis* architecture has to offer a possibility to change the location of storage.

Users want to model *Interrelations* that differ semantically. Thinkable examples could be a hyperlink *Interrelation*, a bookmark *Interrelation*, or different *Interrelations* dealing with document composition. Therefore *Dinopolis*' relation mechanism has to offer distinguishable types of relationships. Further the functionality of those *Interrelation* types has to be extensible. This means reconfiguring the type of a relation at runtime must be possible, since semantics of relationships between documents can change from time to time.

Relations may connect objects that are distributed across the network. Since these

relationships have to be multi directional, it has to be possible to identify *Interrelations* uniquely.

End users won't be pleased with a mechanism that just connects different documents. They want to add some additional information to the relationships they model. For example they will have a need to simply describe a relationship. They want to provide information about the creation of the *Interrelation*: Who has established this relationship. When did he/she do it. And for which purpose. Therefore *Interrelations* must be able to hold meta data.

Applications that are based on *Dinopolis* may be arbitrarily complex. The same applies to the document structure they organize. Therefore relationships in *Dinopolis* must provide a possibility to group connections to various objects within an *Interrelation*. So complex semantics of relationships can be modeled. A group of connections within an *Interrelation* could be called *semantic set* . As semantics of an *Interrelation* may change during its life time, end users need a possibility to add and remove *semantic sets* to/from *Interrelations*. Further an *Interrelation* can be asked for the *semantic sets* it supports. Requesting all connections that belong to a certain *semantic set* allows users to select a group of similar documents at once. Of course it has to be possible to assign a connection to several *semantic sets*. Removing a connection from a *semantic set* has to be possible too.

Interrelations would be rather useless if one could not resolve them. Therefore they have to provide some possibility to retrieve one or more objects they interconnect. The simplest resolve operation each *Interrelation* must offer is requesting all objects a relation connects. But different types of *Interrelations* may provide different possibilities to traverse their connections in various ways. Further it has to be possible to request all relations an object is involved in.

Dinopolis provides a mechanism to integrate one or several embedded *External Systems* like filesystems, databases, or even complex information systems. This is done by so called *Virtual Systems* which allow to integrate a complex combination of *External Systems*. The *Virtual System Management Module* is responsible for the administration of all *Virtual Systems* in a *Dinopolis* instance. Of course the data that is integrated by *Virtual Systems* can be interrelated. So *Dinopolis* must provide a mechanism to import these relationships between external documents too. Importing external relationships has to be done in a completely transparent way. To end users there is no difference between *Interrelations* that are created explicitly and *Interrelations* that are imported implicitly.

4.3.2 Software Requirements

Based on the end users' requirements to applications that are based on *Dinopolis*, application programmers require appropriate features from *Dinopolis*' *Interrelation* mechanism.

Applications may work on rather large data sets. Therefore they need a possibility to connect *Dinopolis Objects* of arbitrary type by creating new *Interrelation* objects of different semantic types. Interconnecting parts of *Dinopolis Objects* with an *Interrelation* has to be possible too. Further, applications need some mechanism to delete *Interrelations*. When deleting an *Interrelation* the *Interrelation Management Module* has to ensure that all *Dinopolis Objects* attached to the *Interrelation* are notified. If deletion of the *Interrelation* would lead to an inconsistent state of one or more *Dinopolis Objects* it has to be prohibited or delayed.

Applications usually provide a possibility to store the data they work on. This applies to *Interrelations* too. So the *Interrelation Management Module* must provide for making *Interrelations* persistent. Of course reloading them has to be possible too. All the data of *Dinopolis* systems, i.e. all *Dinopolis Objects*, are stored to so called *Virtual Systems*. Again this applies to *Interrelations* too. Further it must be possible to move an *Interrelation* from one *Virtual System* to another one or even to another instance of a *Dinopolis* system.

Applications deal with *Dinopolis Objects* of different types. These types provide for modeling different semantics. *Interrelations* may have various semantics too. This has to result in different distinguishable types of *Interrelations*. Those *Interrelation* types have to be uniquely identifiable. The *Interrelation Management Module* has to provide for a possibility to introduce new types of *Interrelations* at runtime. As one cannot know all *Interrelation* types that will be necessary during the lifetime of a *Dinopolis* system, it has to be possible to register new *Interrelation* types with the *Interrelation Management Module*. Further it has to be possible to reconfigure the type of an *Interrelation* at runtime. Therefore the types of *Interrelations* must not be hard-coded. This allows for extensible functionality so that future evolution in the sense of exchanging and extending is possible and simple without the need for basic redesign.

Applications have to display relationships between objects to users. For this purpose they may need some meta information about *Interrelations*. Consequently *Interrelations* must provide a possibility to hold meta data of various types.

Interrelations deal with *Dinopolis Object* that may be distributed across the network. Therefore *Dinopolis Objects* have to be uniquely addressable. Applications may also have a need to model relationships between *Interrelations*. Therefore the requirement of addressability applies to *Interrelations* too.

Dinopolis Objects provide a lot of these features, that are required from *Interrelations*

by applications. The dynamic type mechanism (see Section 4.1.2) of *Dinopolis Objects*, the fact that *Dinopolis Objects* are addressable, and the possibility of *Dinopolis Objects* to hold *Metadata* result in the requirement that interrelations have to be *Dinopolis Objects*.

Within an *Interrelation* connections to *Dinopolis Objects* must be identifiable uniquely. These identifiable connections are represented by so called *Endpoints*. It must be possible to attach a *Dinopolis Object* to an *Interrelation* by creating a new *Endpoint*. Additionally *Endpoints* have to be able to hold meta data. Detaching an *Endpoint* from an *Interrelation* has to be possible too.

To give semantics to an *Interrelation* it has to be possible to group *Endpoints* together. One could say we can define properties for each *Endpoint*. Of course each *Endpoint* may have several different properties (e.g. thinking of a multi dimensional Hyperlink an attached *Endpoint* may have the properties "source-document" and "german-language"). *Endpoints* are grouped within so called *EndPoint Classification Sets*. An *EndPoint Classification Set* is a set of *Endpoints* representing connections that share some property. An *EndPoint Classification Set* is the technical equivalent to a *semantic set*, introduced in Section 4.3.1. These *EndPoint Classification Sets* must be uniquely identifiable by unique IDs. Attaching and removing *Endpoints* to/from *EndPoint Classification Sets* must be possible. Of course an *Interrelation* can be asked for the *EndPoint Classification Sets* it supports. Introducing new *EndPoint Classification Sets* to an *Interrelation* as well as removing *EndPoint Classification Sets* has to be possible. If removing of an *EndPoint Classification Set* is not possible for any reason (e.g. the *EndPoint Classification Set* is essential for a resolve operation), removal has to be inhibited or delayed.

It must be possible to resolve an *Interrelation* in various ways. An *Interrelation* can provide various resolve operations depending on its type. Requesting all *Dinopolis Objects*, an *Interrelation* connects, can be seen as the minimum resolve operation which is provided by all *Interrelation* types. Of course the *Interrelation* doesn't need to return the *Dinopolis Objects* directly. Some mechanism of indirection has to be provided instead. More sophisticated resolve operations based on *EndPoint Classification Sets* can be provided by more specialized *Interrelation* types.

Importing data from different *External Systems* allows applications to work on a combination of different data sets. Of course these *External Systems* can contain some relationships between their data. These relationships have to be imported into *Dinopolis* too and are called *Implicit Interrelations*. *Implicit Interrelations* are mapped to the *Dinopolis* system transparently. this means that to some extent applications can handle them as if they were created explicitly. But since these relationships are implicit they must not lead to *Interrelation* objects.

The *Interrelation* mechanism also has some impact on *Dinopolis Objects*. So *Dinopolis Objects* have to provide an *Interrelation* handler to assure that they do not need to know too much about the mechanisms dealing with *Interrelations*. When a *Dinopolis Object*

is deleted this has to result in detaching it from all *Interrelations* it is involved with. If deletion of the *Dinopolis Object* would lead to an inconsistent state of one or more *Interrelations*, deletion has to be prohibited or delayed. The way a *Dinopolis Object* deals with the problems that may occur during deletion may differ from object type to object type.

Dinopolis Objects have to hold information about each connection to an *Interrelation* within so called *Endpoints*. It must be possible to retrieve all *Endpoints* from a *Dinopolis Object*. Further it must be possible to ask a *Dinopolis Object* for all *Dinopolis Objects* it is interconnected with via a certain *Interrelation*. The *Interrelation Management Module* can be asked for all *Dinopolis Objects* that are interconnected to a given *Dinopolis Object* via a certain *Interrelation*. Such a request must not result in a distributed search. The *Dinopolis Objects* must also not be returned directly. Therefore some way of indirection has to be introduced.

Chapter 5

Interrelations in Detail

This chapter gives an overview of the *Interrelation* Management Module in general and of the internal structure of *Interrelations*. Details about the idea of *EndPoint Classification Sets* can be found in Chapter 6.

5.1 Interrelation Management Module

As described in Chapter 4 the *Dinopolis* system is designed in a very modular way. Therefore all functionality of a *Dinopolis* system is provided by so-called *Functional Modules*. The *Functional Module* that is responsible for everything concerning *Interrelations* in *Dinopolis* is called *Interrelation Management Module*.

The *Interrelation Manager* is responsible for all operations concerning the life-cycle of *Interrelations*. This includes creating, deleting, storing, and loading *Interrelations* as well as changing the types of *Interrelations*. Since *Interrelations* are *Dinopolis Objects*, most of these tasks can be delegated to the object management module (see Section 4.1).

5.2 Internal Structure of Interrelations

Interrelations are uniquely addressable connections, relations, or dependencies between arbitrarily many *Dinopolis Objects* of arbitrary type or parts of them. *Interrelations* between *Interrelations* are possible too. An exception regarding addressability are so-called implicit *Interrelations*. These are special *Interrelations*, which are "created"

implicitly to represent logical relations between objects that are not addressable yet. Since implicit *Interrelations* are "generated" by the "unit" (e.g. a *Virtual System*), that holds these temporary objects, they are no real objects. *Interrelations* are rather a virtual construct. They allow transparent access to the internal structure of e.g. an external system like a filesystem. When speaking about *Interrelations*, **explicit** *Interrelations* are meant, since the main aspect of this thesis is concentrated on them.

Every single *Interrelation* has some *Static Type*. To achieve appropriate functionality for all thinkable scenarios *Interrelations* can have arbitrary *Dynamic Types*. *Dynamic Types* are described in Section 4.1.2.

For each connection an *Interrelation* holds a so called *Endpoint*. To be more precise, an *Endpoint represents* anything (e.g. *Dinopolis Object*, a part of a *Dinopolis Object*, *Interrelation*) that is attached to an *Interrelation*. Multiple attachment of one *Dinopolis Object* to an *Interrelation* results in several different *Endpoints*.

To give semantics to an *Interrelation*, it is possible to group *Endpoints*. One could say we can define properties for each *Endpoint*. Of course each *Endpoint* may have several different properties (e.g. thinking of a Hyperlink, an attached *Endpoint* may have the properties "source-document" and "german-language"). *Endpoints* that share a property are grouped within so called *EndPoint Classification Sets*. For details on *EndPoint Classification Sets* see Chapter 6.

Interrelations can hold Metadata about themselves as well as for each attached *Endpoint*. An *Endpoint*'s metadata do not need to be the same as those for the corresponding *Dinopolis* object. Further an *Endpoint*'s metadata may depend on the properties assigned to the *Endpoint*. Additionally an *Interrelation* may hold Metadata about the *EndPoint Classification Sets* it supports.

5.3 Design Overview

In the *Interrelation Management Module* several parts can be identified. We must distinguish between sub-modules which are dealing with the management of *Interrelations*, sub-modules which represent the model of an *Interrelation*, and sub-modules which hold information about the interconnected objects. Additionally the *Interrelation* relevant parts of the design of *Dinopolis Objects* are also in the scope of this section.

The *Interrelation Manager* (see Section 5.4) is responsible for dealing with the lifecycle of *Interrelations*. An *Interrelation* (see Section 5.5) is a *Dinopolis Object* which represents the model of an interconnection. Information about connected *Dinopolis Objects* is stored in *Endpoints* (see Section 5.6.1). These *Endpoints* are organized by the

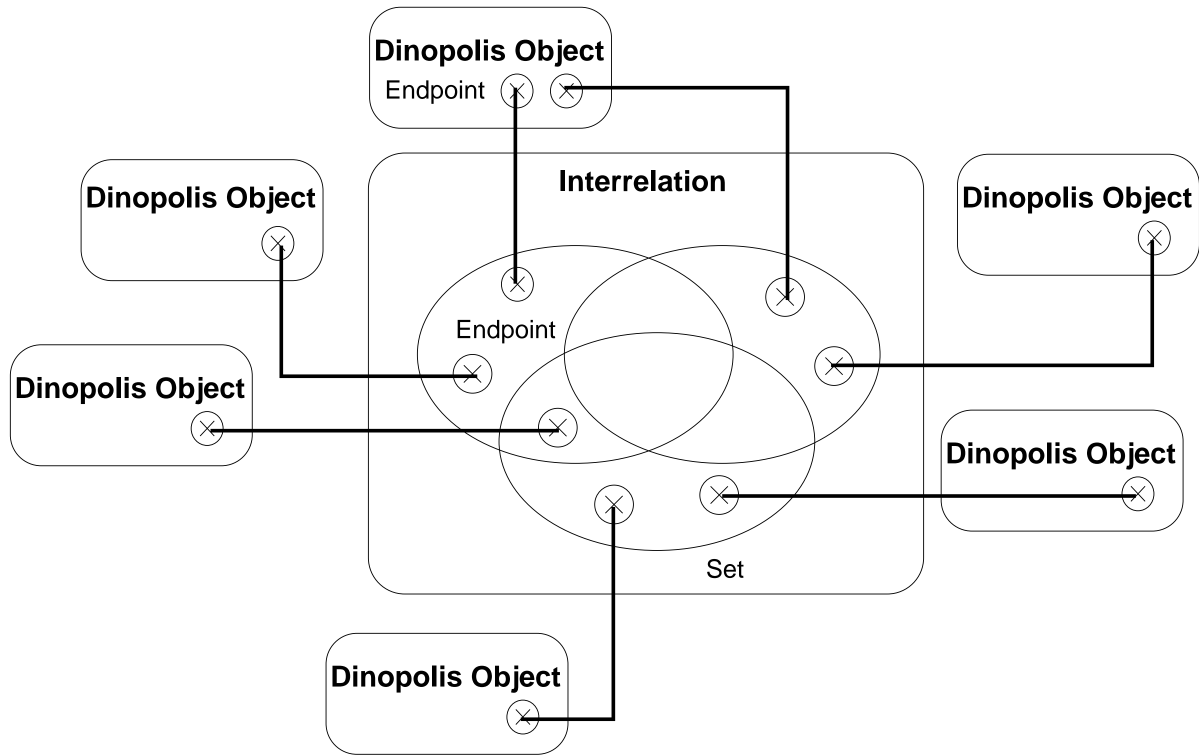

Figure 5.1: An Interrelation's logical-view

so called *Endpoint Container* (see Section 5.5.3).

Furthermore various properties can be assigned to *Endpoints*. With these properties it is possible to model the semantics of an *Interrelation*. *Endpoints* that share a property are grouped in *EndPoint Classification Sets* (see Chapter 6). *EndPoint Classification Sets* are organized within a *Set Container* (see Section 6.4). An example for the logical view of an *Interrelation* organizing its *Endpoints* in several *EndPoint Classification Sets* is shown in Figure 5.1.

5.4 Interrelation Manager

The *Interrelation Manager* is responsible for managing the life cycle of *Interrelations* in the *Dinopolis* system. It provides the public interface of the *Interrelation Management*

Module and is registered with the *Module Management Module* as a functional module.

5.4.1 Lifecycle of an Interrelation

The *Interrelation Manager* is responsible for all operations concerning the life cycle of an *Interrelation*. The following life cycle operations are supported:

- Create an *Interrelation*
  See Section 7.1 for a detailed description of this process.
- Delete an *Interrelation*
- Store an *Interrelation*
- Load an *Interrelation*
- Move an *Interrelation*
- Change the type of an *Interrelation*
  See Section 5.4.2.

Since *Interrelations* are *Dinopolis Objects* all these operations can be delegated to the *object management module* (see Section 4.1).

5.4.2 Dynamic Type Mechanism concerning Interrelations

The *Interrelation Manager* also provides operations concerning *Dynamic Types* of *Interrelations*. This functionality is handled by the *Interrelation Manager*, which internally passes the corresponding calls to the *Object Manager*(see Section 4.1).

5.5 Interrelations

Interrelations interconnect *Dinopolis Objects* of various types. An *Interrelation* is a *Dinopolis Object* and therefore it is addressable over the network, can hold meta-data and provides extensible functionality by means of the *Dynamic Type* mechanism. Therefore the highest possible integration within the *Dinopolis* system is guaranteed. Figure 5.2 shows the internal view of an *Interrelation*.

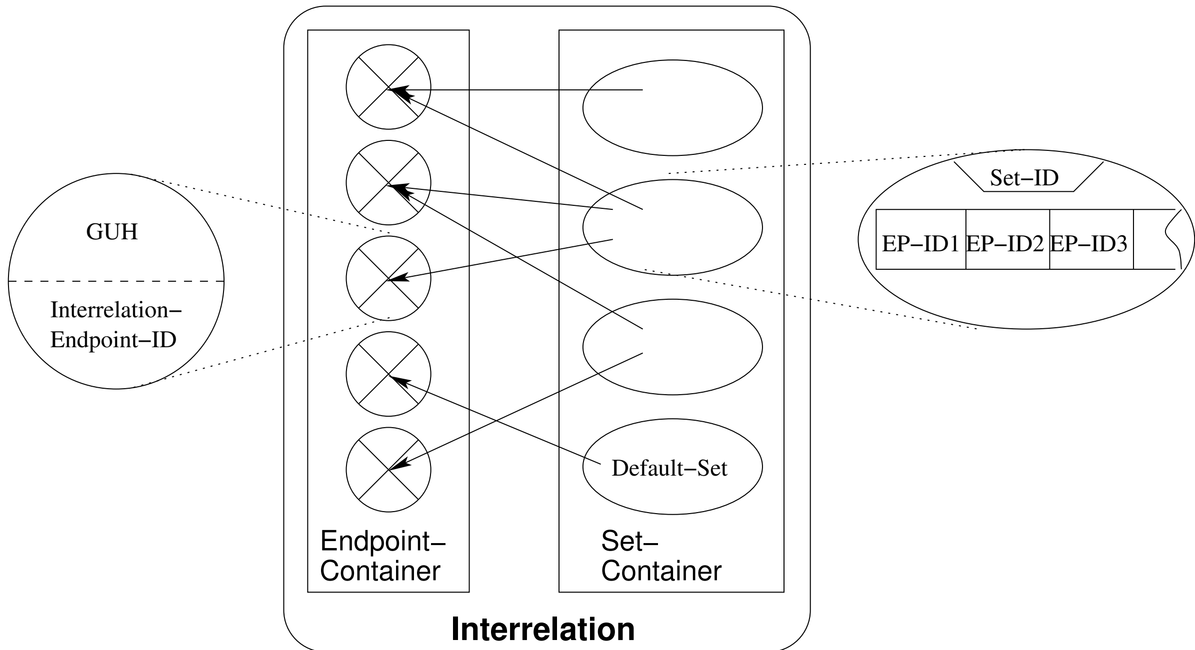

Figure 5.2: An Interrelation's Internal Structure

From a Logical point of view an *Interrelation* is a container of connections. These connections are represented as so called *Endpoints* (see Section 5.5.2). Of course it is possible to attach or detach a *Dinopolis Object* from an *Interrelation* at runtime. Therefore the following operations exist:

- Attach a *Dinopolis Object* to an *Interrelation*.
  An *Endpoint* representing the connection to the given *Dinopolis Object* is added.
- Detach a *Dinopolis Object* from an *Interrelation*.
  The *Endpoint* representing the given *Dinopolis Object* is removed.

The semantics of an *Interrelation* is managed by assigning properties to *Endpoints*. This is done by grouping the *Endpoints* within sets of same properties. Therefore the following operations are provided by all *Interrelations*:

- Create a new *EndPoint Classification Set* within an *Interrelation*.
- Get all *EndPoint Classification Sets* an *Endpoint* is assigned to.

- Remove an *EndPoint Classification Set* from an *Interrelation*
  If removing of an *EndPoint Classification Set* is not possible for any reason (e.g. the *EndPoint Classification Set* is essential for a resolve operation) removal has to be inhibited or delayed.
- Assign an *Endpoint* to an *EndPoint Classification Set*.
- Remove an *Endpoint* from an *EndPoint Classification Set*.

For details on *EndPoint Classification Sets* refer to Chapter 6.

Additional functionality is provided by several operations in the sense of the *Dinopolis* architecture (see Section 4.1). One of these functionalities is to resolve an *Interrelation*. Various resolve operations are thinkable depending on the *Interrelation*'s type. The following resolve operations are provided by all interrelations:

- Get all attached *Endpoints*.
  This method returns all *Endpoints* attached to this *Interrelation*.
- Get all attached *Endpoints* with a certain property.
  This method returns all *Endpoints* that are assigned to a certain *EndPoint Classification Set*.
- Get a certain *Endpoint* that is attached to the *Interrelation*.

Dynamic Types can define additional resolve operations. For example retrieving a set of desired *Endpoints* could be processed by passing an appropriate *visitor* to the *Interrelation*. The *Interrelation*'s resolve operation allows the *visitor* to retrieve some *Endpoints*, depending on the *visitor*'s type. The information about the concerned *Endpoints* that the *visitor* requests, depends on its type too. The *visitor* must be some kind of container as it holds all the *Endpoints* it retrieves. Therefore *iterators* of different types can be requested from it. These *iterators* allow for convenient navigation over the *Endpoints*. The design patterns *iterator* and *visitor* are described in detail in [Gamma et al., 1995]. For a detailed description of this resolve process see Section 7.4.1.

Further it may be useful to request the number of attached *Endpoints*. Which special queries are provided by an *Interrelation* depends on its type. The following requests are provided by all interrelations:

- Get the number of attached *Endpoints*.
- Get the number of attached *Endpoints* with certain property.

Dynamic Types can define additional queries.

5.5.1 Types of Interrelations

Since *Interrelations* are *Dinopolis Objects* they can have different types. Each *Interrelation* have a *Static Type* which is inherited from the base-type "*Interrelation*". This base-type provides all basic functionality needed by all *Interrelations*.

For appropriate functionality the basic static type of *Interrelations* can be extended by definitions of arbitrary *Dynamic Types*. The dynamic type mechanism of *Dinopolis Objects* is described in detail in [Haselwanter, 2003]. Basic functionality of an *Interrelation* can be defined using the dynamic type mechanism. It is also possible to define additional functionality, such as more powerful resolve methods.

Thus several *Interrelation* types can be modeled. It is also possible to extend an existing *Interrelation* by adding definitions of two or more different *Dynamic Types*. This mechanism allows for modeling relations like "multilingual hyperlinks". In the case of extending an *Interrelation* with different incompatible *Dynamic Types* (e.g. undirected vs. directed *Interrelation*) the dynamic type mechanism of the *Dinopolis Object* provides a mechanism to resolve such a conflict.

An *Interrelation* can provide various *EndPoint Classification Sets* to model relations between connected *Dinopolis Objects*. These sets are also extensible and definable by using the dynamic type mechanism. For a detailed description of this feature see Chapter 6.

5.5.2 Endpoints (of Interrelations)

As *Interrelations* have to know about the *Dinopolis Objects* attached to them, they have to provide a mechanism to hold this information. The amount of information that is held by an *Endpoint* depends on the *Interrelation*'s type.

The following information about *each connection* is provided by *Endpoints* of all *Interrelations*:

- GUH of the *Dinopolis Object*
- IREPID (*Interrelation Endpoint* ID)
  As a *Dinopolis Object* may be attached several times to the same *Interrelation* for different reasons (and with different properties), each connection needs a unique identifier within the *Interrelation*. Therefore this ID is generated by the *Interrelation* itself.

Endpoints may provide additional information about the connections they represent. The following additional data may be useful:

- Type of the *Dinopolis Object*.
  If the type of an interconnected *Dinopolis Object* is held within the *Interrelation*, this information has to be synchronized, since the dynamic type of a *Dinopolis Object* may change during its lifetime. To realize synchronization, it is necessary to observe the *Dinopolis Object*.

- Metadata about the connection.
  Endpoints may hold arbitrary metadata. For example metadata about the corresponding *Dinopolis Object* may be useful. Caching this information may prevent from unnecessary accesses to the *Dinopolis Object*. According to the general way metadata is organized in *Dinopolis Objects* an *Endpoint*'s metadata is structured hierarchically.

So *Endpoints* are containers that hold all the information about connections to *Dinopolis Objects*.

5.5.3 Endpoint Container

The *Endpoint Container* organizes all *Endpoints* of an *Interrelation*. It is also responsible for generating unique *Interrelation Endpoint* IDs. Further it provides a mapping from those IDs to the corresponding *Endpoints*. A handler on the *Endpoint Container* provides all functionality needed to work on it internally.

Methods that the *Endpoint Container* respectively the *Endpoint Container Handler* provides:

- Add an *Endpoint*.
  This implies the generation of an *Interrelation Endpoint* ID and establishing an appropriate mapping within the *Endpoint Container*.

- Remove an *Endpoint*.
  This releases the ID to *Endpoint* mapping.

- Get a certain *Endpoint*.
  This operation accepts an *Endpoint* ID as its argument and returns the corresponding *Endpoint*.

- Get all *Endpoints*.

- Get the number of *Endpoints*.

5.5.4 EndPoint Classification Sets

Since *EndPoint Classification Sets* are an outstanding feature of *Dinopolis*' *Interrelation* mechanism this thesis dedicates an own chapter to this topic. So please refer to Chapter 6 for details on *EndPoint Classification Sets*.

5.5.5 Metadata

Interrelations can hold arbitrary metadata. Besides the general metadata about the *Interrelation* itself it may be useful to hold metadata about each of its *Endpoints*. Whether an *Interrelation* can hold such additional *Metadata* depends on the *Interrelation*'s type.

Dinopolis Objects provide a standardized way to organize metadata. And since *Interrelations* are *Dinopolis Objects* they use this mechanism to organize their metadata.

5.5.6 Data held by an Interrelation

An *Interrelation* holds information about arbitrary connections between *Dinopolis Objects*. This information are *Endpoints* which are managed by the *Endpoint Container* and *EndPoint Classification Sets* which are managed by the *Set Container*.

Since there are a lot of different circumstances that influence the organization of an *Interrelation*'s persistent data, the way this data is managed depends on the *Interrelation*'s type. The recommended default mechanism to hold this information within an *Interrelation* object is as follows:

Since an *Interrelation* is a *Dinopolis Object*, data is kept according to the concept of the internal structure of a *Dinopolis Object*. In this structure data is encapsulated within the content part. Since the data held by an *Interrelation* can be seen as its content the straightforward approach is to use the content part of *Dinopolis Objects* to hold the Interrelation-specific data. The advantage of this approach is that the clear design of *Dinopolis Objects* is not broken. But it is not possible to hold additional "content" such as an image (e.g. for visualization purposes) within an *Interrelation*. For details on this topic see [Thalauer, 2004].

5.6 The Interrelation Part of Dinopolis Objects

One of the most outstanding features of the *Dinopolis* system is that *Interrelations* are bidirectional in the sense that a *Dinopolis Object* knows its *Interrelations* and vice versa. Therefore it would be insufficient that the *Interrelations* know about all *Dinopolis Objects* that are attached to them. It is also necessary that a *Dinopolis Object* knows all *Interrelations* it is involved with (see Figure 5.3).

Of course this is only true for explicit *Interrelations*. If a *Dinopolis Object* is "attached" to implicit *Interrelations* it just has to provide a mechanism to create all corresponding *Endpoints* on the fly. For details on *implicit Interrelations* see [Thalauer, 2004].

For the sake of modularity *Dinopolis Objects* do not need to provide functionality that deals with the organization of *Interrelations* directly. This functionality is added by attaching an appropriate *Interrelation* definition (also called *Interrelation* part of a *Dinopolis Object*). This is a special *Dynamic Type* which has to be attached to all *Dinopolis Objects*. The public interface of the *Interrelation* definition is provided by a so called *Interrelation* handler (see Section 5.6.3).

As each *Dinopolis Object* has to know all explicit *Interrelations*, it is attached to, they have to be organized in a way the *Interrelation* definition can access them. Several possibilities to manage an object's *Interrelations* are described in Section 5.6.2.

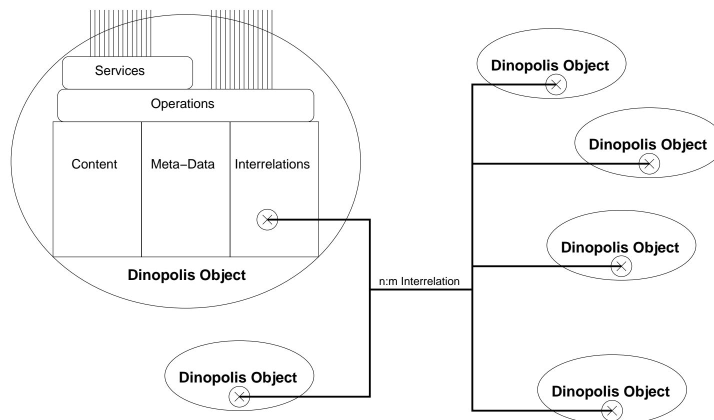

Figure 5.3: The *Dinopolis Object's Internal Structure*

5.6.1 Dinopolis Object side Endpoint

As *Dinopolis Objects* have to know about the *Interrelations* they are attached to, they have to provide a mechanism to hold this information to avoid a distributed search. This is comparable to the *Interrelation* side *Endpoints* described in Section 5.5.2. Therefore we will call this data structure *Endpoint* too. In most cases it should be clear, which *Endpoint* is meant, otherwise it has to be specified more precisely (*Interrelation*-side vs. *Dinopolis Object*-side *Endpoint*).

Endpoints representing *Explicit Interrelations* are attached explicitly whereas *Endpoints* for *Implicit Interrelations* are generated by the *Dynamic Storage Part*. This is a special *Dynamic Type* provided by the object management module. For details on this topic see [Thalauer, 2004].

The amount of information held by *Endpoints* within a *Dinopolis Object* depends on the *Dinopolis Object*'s *Interrelation* definition. The following information about *each connection* to an *Interrelation* is provided by all *Dinopolis Object*-side *Endpoints*.

- Identifier of the *Interrelation*
  For an *Explicit Interrelation* the *Endpoint* holds the *Interrelation*'s GUH, whereas it has to know the ID of the *Virtual System* plus a local ID (LUH) for *Implicit Interrelations*.

All *Endpoints* representing explicit *Interrelations* additionally provide the following information:

- *Dinopolis Object*-side IREPID (*Interrelation Endpoint* ID)
  As a *Dinopolis Object* may be attached several times to one *Interrelation* for different purposes (and with different properties), each connection needs a unique identifier within the *Dinopolis Object*. So this ID consists of two parts. The GUH of the *Interrelation* and the IREPID , generated by the according *Interrelation*.

Dinopolis Object-side *Endpoints* may provide additional information. The following additional data may be useful in many cases:

- Key for object subset
  As *Interrelations* may point to some part of the *Dinopolis* Object, an internal key for object-subsets is necessary. As the exact realization of this ID is heavily dependent on the type of the *Dinopolis Object*'s content, the content handler has to provide for generating and dealing with it. Since *Endpoints* can also point to certain metadata of the *Dinopolis Object*, the subset-key can be requested from the metadata handler too. (Content and metadata handler are both provided by the object management module.)

- Metadata
  An *Endpoint* can hold metadata about the corresponding *Interrelation*. This is an optional feature, but may prevent from unnecessary accesses to the *Interrelation*. Of course arbitrary additional Metadata can be held by an *Endpoint*.

- Type of the *Interrelation*
  If the *Dinopolis Object*-side *Endpoint* holds this information, it has to be synchronized on type changes. To realize synchronization it is necessary to observe the *Interrelation* object.

In contrast to *Interrelation*-side *Endpoints*, *Dinopolis Object*-side *Endpoints* do not have individual properties. Additional information about an *Endpoint* has to be modeled by adding appropriate metadata.

5.6.2 The Dinopolis Object side Endpoint Container

As described in Section 5.5.3 *Interrelation*-side *Endpoints* are organized by an *Endpoint* container. The same applies for *Dinopolis Object*-side *Endpoints*. A handler on the *Endpoint Container* provides all functionality needed to work on it internally.

The main task of the *Endpoint Container* is to manage the organization of all *Endpoints* of a *Dinopolis Object* and provide an interface to communicate with the *Interrelation Handler*. The most important requirement that this container has to fulfill is that a request for a certain *Endpoint* has to be processed in "appropriate" time. I.e. a request for an *Endpoint* must not lead to a systemwide search but has to be answered directly. This doesn't mean that the *Endpoint Container* has to be held within the *Dinopolis Object* itself under all circumstances. Several situations are thinkable, where *Dinopolis Objects* cannot or should not hold their *Endpoints* themselves. The most obvious problem occurs when a *Dinopolis Object*'s persistent data is located in a read-only system. If the *Endpoint Container* would also be stored read-only, it wouldn't be possible to attach this object to any *Interrelation* any more.

But even if the *Dinopolis Object* is residing in a writable system, it could be undesired that attaching this *Dinopolis Object* results in a write-access on the local system of the *Dinopolis Object*. So another argument against storing the *Endpoints* within the *Dinopolis Object* is security. Attaching the *Dinopolis Object* to an *Interrelation* always leads to a write-access on the *Dinopolis* object. No matter who has triggered the attachment (there are no general restrictions for attaching *Dinopolis Objects* to *Interrelations*).

A further reason why this mechanism isn't the best choice under all circumstances, is performance. Consider a rather interesting document that is located on a small system. If too many people would like to link to this document the system could be swamped by extensive write accesses.

For a detailed discussion of the *Interrelation* related data of *Dinopolis Objects* please refer to [Thalauer, 2004].

5.6.3 The Interrelation Handler

On the one hand the *Interrelation Handler* provides all the *Interrelation*-related functionality that other modules and applications need from *Dinopolis Objects*. On the other hand it handles the *Dinopolis Object*-internal, *Interrelation*-related management. It also communicates with *Interrelations*. Especially when the *Dinopolis Object* is attached to an *Interrelation*.

Applications, modules and other *Dinopolis Objects* (including *Interrelations*) can request an *Interrelation Handler* from a *Dinopolis Object*. The *Interrelation Handler* then provides an appropriate proxy.

The following operations are provided by a minimal *Interrelation* handler:

- Request all connections the *Dinopolis Object* is involved in.
- Request number of connections the *Dinopolis Object* is involved in.
- Request a certain connection.

A more sophisticated mechanism to retrieve *Endpoints* from a *Dinopolis Object*'s *Interrelation* handler may work the following way:

- Request all *Endpoints* that fulfill a certain criterion.

  This is done by passing an appropriate container (that has to conform to the *visitor* design pattern) to the *Interrelation*. The container is filled according to its type. A more detailed description of this mechanism can be found in Section 5.5.

  The *visitor* either selects all *Endpoints* or only those it is interested in. For example it can restrict the result to *Endpoints* representing *Interrelations* of a certain type, *Interrelations* containing special metadata or any other filter criterion. A *visitor* can also be designed to process explicit (respectively implicit) *Interrelations* only.

When a *Dinopolis Object* is attached to an *Interrelation*, the *Interrelation* contacts the *Dinopolis Object* to add a corresponding *Endpoint*. On the other side an *Interrelation* needs a possibility to remove an *Endpoint* when a *Dinopolis Object* is detached from it. Therefore each *Interrelation* handler has to provide the following operations to *Interrelations*.

- Add an *Endpoint*.
- Remove an *Endpoint*.

Deleting a *Dinopolis Object* has to result in releasing all connections it is involved in. Therefore it has to be detached from all *Interrelations* in a transaction safe way. Consequently the *Interrelation* handler of each *Dinopolis Object* provides the following operation:

- Detach *Dinopolis Object* from all *Interrelations*.

Chapter 6

Endpoint Classification Sets

6.1 Concept

The main idea behind *EndPoint Classification Sets* is the realization of hyper-dimensional *Interrelations*. hyper-dimensional relations allow to model arbitrarily complex **n:m:l:k:...** relationships between arbitrarily many objects. Therefore *EndPoint Classification Sets* introduce a very powerful tool to the *Dinopolis* system.

For example hyperlinks are not restricted to connect a set of source documents with a set of destination documents. Document structures can e.g. be enriched by introducing various language versions of each document. Further it is possible to introduce versioning. Of course *Interrelations* can also provide sophisticated resolve operations based on these *EndPoint Classification Sets*. So requesting all destination documents of the same version that are written in the same language as a certain source document can be easily implemented by this mechanism.

As we know, a set, in its mathematical sense, is a collection of distinguishable objects that all have a certain property. A set's objects are called elements. Regarding *EndPoint Classification Sets* these elements are *Endpoints* that share a certain property. In some cases it might be desirable that sets are well-ordered. Therefore a "well ordering relation" can be defined for an *EndPoint Classification Set* optionally.

In mathematics various operations are defined on sets:

$A \cap B$	A *intersection* B is the set of all elements that are in both sets A and B
$A \cup B$	A *union* B is the set of all elements that are either in A or in B or in both
$A \setminus B$	A *minus* B are all elements from A that are not in B

These set-operations are exactly what we need to implement arbitrary resolve strategies on *Interrelations*. But even if most of these operations are rather powerful, their software-implementation generally suffers from a lack of efficiency. The efficiency-problem of most of these operations can be solved by caching. Therefore the implementation of *EndPoint Classification Sets* has to be balanced between performance and memory efficiency.

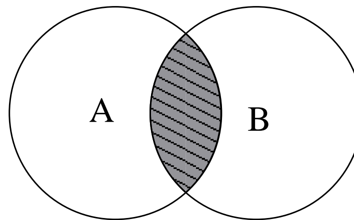

Figure 6.1: Intersection of two sets

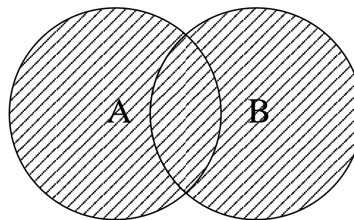

Figure 6.2: Union of two sets

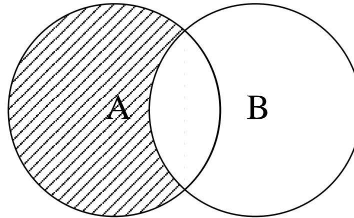

Figure 6.3: Difference of two sets

6.2 An Example

Let's look at the example mentioned above in more detail. We want to define a

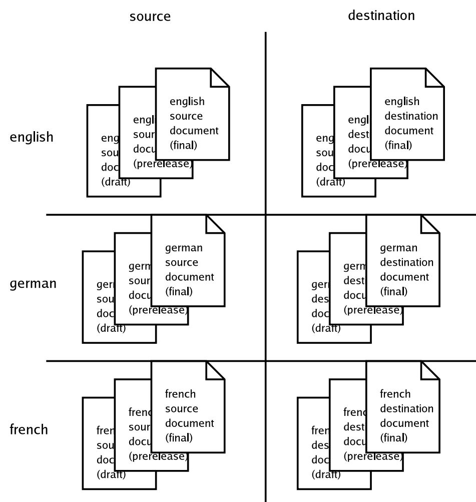

Figure 6.4: Some documents that coud be connected via an Interrelation

hyperlink that supports several source objects and several destination objects. Further it supports various language versions for each object (our example hyperlink supports English, German, and French document versions). And since each document is changed from time to time, our hyperlink supports versioning. Figure 6.4 shows some documents that could be connected via such an *Interrelation*.

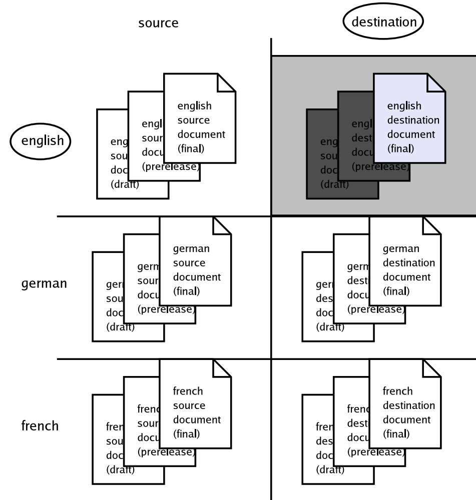

Figure 6.5: select English destination document of final version

So our hyperlink *Interrelation* will support the following *EndPoint Classification*

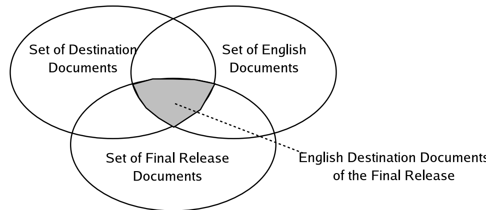

Figure 6.6: Intersection of three sets

Sets:

- set of source documents
- set of destination documents
- set of English documents
- set of German documents

- set of French documents
- set of initial draft documents
- set of pre release documents
- set of final release documents

If we want to retrieve the English destination document(s) of the final release (as shown in Figure 6.5) we would have to perform an intersection as shown in Figure 6.6.

Another thinkable scenario could be a request for all "old" documents (i.e. all doc-

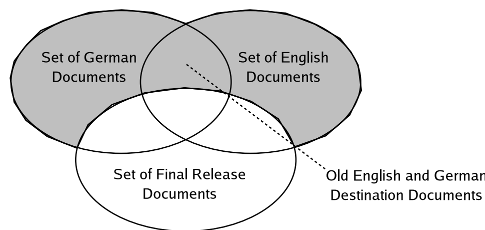

Figure 6.7: Combination of union and intersection.

uments that are not part of the final release) that are either written in German or English. The necessary set operations to fulfill this request are illustrated in Figure 6.7. Of course all combinations of set operations are possible to model complex requests based on *EndPoint Classification Sets*.

6.3 Design of EndPoint Classification Sets

As described in Section 5.5.2 various properties can be assigned to each *Endpoint* of an *Interrelation*. Therefore a rather straightforward implementation of such properties is to accumulate all *Endpoints* that share a certain property within a set. These sets are called *EndPoint Classification Sets*.

As described in Section 6.1 there are three base operations dealing with sets. Of course these operations can be combined arbitrarily to implement more sophisticated operations. The amount of set operations an *Interrelation* supports, depends on its type. Each *Interrelation* supports at least the following base operations:

- Intersection of two sets (see Figure 6.1)
- Union of two sets (see Figure 6.2)
- Difference of two sets (see Figure 6.3)

Of course an *Interrelation* can also provide shortcuts for more sophisticated set operations.

There are three possible reasons for the existence of an *EndPoint Classification Set*:

Explicit EndPoint Classification Sets

As already mentioned, *EndPoint Classification Sets* are *Dinopolis*' concept to model complex semantics for hyper-dimensional relationships between *Dinopolis Objects*. Generally there are two ways to declare an *EndPoint Classification Set* explicitly. On the one hand a *Static* or *Dynamic Type* of an *Interrelation* can define specific *EndPoint Classification Sets* to model its semantics. For example a hyperlink will always define a *source* and a *destination set* to define its direction. On the other hand users may extend an *Interrelation* by manually adding *EndPoint Classification Sets* at runtime. For example one could add language support to an existing hyperlink *Interrelation* by defining several *language sets*.

Implicit EndPoint Classification Sets

EndPoint Classification Sets can also be created implicitly. An example for a useful implicit *EndPoint Classification Set* would be an accumulation of all *Endpoints* of the same type. Caching the results of mathematical set-operations could be another reasonable case. The creation of implicit sets can be triggered by various operations defined by *Static* or *Dynamic Types* of an *Interrelation*.

Default EndPoint Classification Set
Since resolve operations are mainly based on *EndPoint Classification Sets*, every single *Endpoint* has to be assigned to at least one of them. Therefore each *Interrelation* has to know a default set that holds all *Endpoints* that are not assigned to a specific explicit set.

6.3.1 The Internal Structure of an Endpoint Classification Set

EndPoint Classification Sets hold all the information that is necessary to implement the concepts described earlier. Further they have to provide some meta information about their semantics to end users. Therefore an *EndPoint Classification Set* consists of the following data:

- The *EndPoint Classification Set*-ID
  Each *EndPoint Classification Set* has an *Interrelation*-internally unique identifier. This ID is needed to manage the various *EndPoint Classification Sets* within the *Set Container* (see Section 6.4) and of course to identify each set uniquely. This ID is generated by the *Set Container* when a new *EndPoint Classification Set* is created and is immutable during the whole life time of the set.
- A data structure of *Endpoint*-handles
  This data structure organizes the IDs of all *Endpoints* that are assigned to the *EndPoint Classification Set*.
- Metadata
  EndPoint Classification Sets can hold *Metadata* of various types. For the sake of uniformity this is organized in the same way as the metadata of *Dinopolis Objects*. I.e. as tree-structured container of hierarchical keys with values of arbitrary type. The metadata can be altered by using a special metadata handler (see below).

6.3.2 The interface of EndPoint Classification Sets

The following methods are provided by *EndPoint Classification Sets*:

- Get set-ID.
- Add an *Endpoint* by passing the *Endpoint*-ID.
- Remove an *Endpoint* with a given *Endpoint*-ID.

- Get all *Endpoint*-IDs.
  This operation returns a collection of the *Endpoint*-IDs of all *Endpoints* that are attached to the set. To ease further processing of this result it is guaranteed that the collection is sorted by ascending *Endpoint*-IDs.
- Get metadata handler
  The metadata handler provides all operations that are necessary to work on the set's metadata.

6.4 Set Container

The *Set Container* organizes the *EndPoint Classification Sets* of an *Interrelation* and provides interfaces to communicate with other submodules of the *Interrelation*.

The content of the *Set Container* is just a collection of *EndPoint Classification Sets*. A handler on the *Set Container* provides all functionality needed to work on it internally.

The following operations are provided by the *Set Container*:

- Create a new *EndPoint Classification Set*.
  This inserts a new set within the *Set Container*. Further it generates a unique ID for the newly created set.
- Remove a set with a given set ID.
- Get the number of *EndPoint Classification Sets*.
- Retrieve a certain *EndPoint Classification Set*.
  The set that should be returned is defined by its ID.
- Remove an *Endpoint* from all *EndPoint Classification Sets*.
  The *Endpoint* is defined by its *Endpoint*-ID.

Chapter 7

Processes and Algorithms

This chapter outlines some processes that are fundamental for organizing *Interrelations*. Of course it would be far beyond the scope of this thesis to describe all processes in detail. Therefore this chapter concentrates on a collection of the most important processes and algorithms.

7.1 Creating an Interrelation

Notice that this description of the creation process of a new *Interrelation* concentrates on the *Interrelation* related matters only. Since *Interrelations* are *Dinopolis Objects* the actual process is by far more complex than described here.

1. Creation of a new *Interrelation* is requested via the *Interrelation Manager* by passing the *Interrelation*'s static type and a handle which addresses the location of the persistent data of the new *Interrelation*. This handle consists of a *Virtual System Identifier* and a *Local Placement Identifier* which determines the exact location inside the *Virtual System*. Skipping the *Virtual System Identifier* or the *Local Placement Identifier* as a parameter leads to an *Interrelation* that is stored in a predefined default location.

   The chosen *Virtual System* has to be able to hold *Interrelations*. Since the main task of an *Interrelation* is to manage connections by attaching and detaching *Endpoints*, it has to be mutable. Therefore the minimum requirement for *Interrelation* capable *Virtual Systems* is that they are writable.

2. The *Interrelation Manager* now requests a new empty *Interrelation* from the *Object Manager* by passing this handle and the desired static type.

3. Initialization of the basic data of the static *Interrelation* type.
   During the creation of the empty *Interrelation* object the basic *Interrelation* type has to be initialized, i.e. all data containers have to be set up.

   The following steps are necessary:

   a) Create and initialize *Interrelation*-specific data.
      There are several possibilities where this data can reside (see Section 5.5.6). The following steps are necessary to initialize this data:

      i. Create and initialize the *Endpoint Container*.
      ii. Set the Handler for the *Endpoint Container*.
      iii. Create and initialize the *Set Container*
      iv. Set the Handler for the *Set Container*.

   b) Create and initialize the *Content Data* part.
      This includes loading the appropriate *Interrelation* specific *Content Data Handler*.

   c) Create and initialize the *Metadata*-part.
      This includes loading the appropriate *Interrelation*-specific *Metadata Handler*.

   d) Create and initialize the *Interrelation*-part.
      Since each *Interrelation* is a *Dinopolis Object*, it can be attached to other *Interrelations*. Therefore it has to provide some mechanism to manage these *Endpoints*. This functionality is in the scope of responsibility of the *Interrelation* part, which is supported by all *Dinopolis Objects*. This *Interrelation*-part holds an *Endpoint Container* for these connections.

   e) Attach an appropriate *Interrelation Handler*.
      The *Interrelation Handler* provides the operations to manage the connections to other *Interrelations* this *Interrelation* object is attached to.

7.2 Attaching a Dinopolis Object to an Interrelation

A request for attaching a new *Dinopolis Object* is called directly on the *Interrelation*. The only mandatory parameter needed by this operation is the GUH of the *Dinopolis Object*. *Metadata* about this connection can be defined optionally. Additional sematic properties of the connection can be passed to this operation. If the new *Endpoint* shall

represent a reference to some subset of a *Dinopolis Object*, a key to this subset can be defined too.

The algorithm for attaching a new *Endpoint* is the following:

1. Get the *Dinopolis Object*
   The *Dinopolis Object*, defined by the GUH, is requested from the *Object Manager*. If an error occurs an exception is thrown and the algorithm terminates.

2. Get the handler for the *Endpoint Container* of the *Interrelation*.

3. Request the creation of a new *Endpoint* from the *Endpoint Container*.

   a) A new empty *Endpoint* is created.
   b) The *Interrelation Endpoint ID*(IREPID) for the new *Endpoint* is generated.
   c) Assign the *Interrelation Endpoint ID* to the *Endpoint*.
   d) Assign the given GUH to the *Endpoint*.
   e) Add *Metadata* to the *Endpoint*
      If *Metadata* is defined for the connection it is added to the new *Endpoint*.
   f) Register the new *Endpoint* with the *Endpoint Container*.

4. Assign the *Endpoint* to *EndPoint Classification Sets*
   If semantic properties are given for the *Endpoint* the new *Endpoint* is assigned to the corresponding *EndPoint Classification Sets* by calling the appropriate operation on the *Set Container*. If one or more of the *EndPoint Classification Sets* don't exist the *Endpoint* is detached and an appropriate exception is thrown. (Another possibility that could be implemented by a special *Dynamic Type*, is to create the missing *EndPoint Classification Set* implicitly on demand.)

5. Request the *Interrelation Handler* from the *Dinopolis Object*.

6. Register the new connection with the *Dinopolis Object* by calling the appropriate operation on the *Interrelation Handler* by passing the *Interrelation*'s GUH, the *Interrelation Endpoint ID* and optionally *Metadata* and the key to the subset of the object. The following steps concerning the *Dinopolis Object* are necessary:

   - A new empty *Endpoint* is created.
   - The *Interrelation's* GUH is assigned to the *Endpoint*.
   - The *Interrelation Endpoint ID* is assigned to the *Endpoint*.
   - If a subset key is given it is assigned to the *Endpoint*.
   - If metadata is given it is added to the *Endpoint*.
   - Register the new *Endpoint* with the *Endpoint* Container of the *Dinopolis Object*.

7.3 Detaching an Endpoint from an Interrelation

The request for detaching an *Endpoint* is directly called on the *Interrelation* by passing the *Interrelation Endpoint ID*.

Description of the algorithm:

1. Request the handler on the *Endpoint Container*.
2. Send a request for the *Endpoint*, defined by its *Interrelation Endpoint ID*, to the *Endpoint Container*.
3. Request the *Dinopolis Object*, defined by the given GUH, from the *Object Manager*. If an error occurs, the algorithm terminates. (Another possibility that could be implemented by a special *Dynamic Type*, is to delay deletion until the *Dinopolis Object* is available.)
4. Request the *Interrelation Handler* from the *Dinopolis Object*.
5. Remove the connection from the *Dinopolis Object* by calling the appropriate operation on the *Interrelation Handler*.
6. Request the handler on the *Set Container*.
7. Instruct the *Set Container* to remove the *Endpoint* from all *EndPoint Classification Sets*.
8. Remove the *Endpoint* from the *Endpoint Container*.
9. Delete the *Endpoint*.

7.4 Resolve

As already noticed, *Interrelations* may provide various resolve operations depending on their *Static* and *Dynamic Types*. This section describes two thinkable processes.

7.4.1 Request a Selection of Endpoints from an Interrelation

A request for a selection of Endpoints that are attached to an *Interrelation* is called directly on this *Interrelation* by passing an appropriate container. The container has

to conform to the *visitor* design pattern. The type of the container defines which *Endpoints* are requested and which information is included in the container's elements. The container is only accepted as *visitor* if the *Interrelation* is able to handle this type of container or one of its superclasses. If the container finally is accepted as *visitor*, the *Interrelation* provides all the functionality, it needs to be filled in a proper way. If no *Endpoints* are attached to the *Interrelation* the container stays empty.

The basic *Interrelation* type provides functionality to fill containers with *Endpoints* depending on the *EndPoint Classification Sets* they are assigned to. Union, Intersection and Complement are the possible operations on *EndPoint Classification Sets*. The fill-methods can combine them arbitrarily. Further they can provide filters regarding *Endpoint*'s metadata, type, or any other criterion.

To allow for convenient navigation through the *Endpoints* of such a container, different types of *iterators* on the container are provided. Of course these *iterators* can provide some sorting mechanism based on arbitrary criteria. The design patterns *iterator* and *Visitor* are described in detail in [Gamma et al., 1995].

7.4.2 A more specific Resolve Operation

Now we want to have a look on a slightly more specialized resolve operation. Let's recall the example outlined in Section 6.2 for that purpose. From an *Interrelation* providing *EndPoint Classification Sets* for *source* and *destination* documents as well as for versions in *English*, *German*, and *French* language. Further all documents exist as *initial drafts*, *pre releases*, and *final releases.*

What we want is to retrieve the intersection of any of the three sets – one from each classification criterion. For example one could request all English source documents in their final versions.

The following algorithm describes this process:

- Call the operation on the *Interrelation*.
  The *EndPoint Classification Set*-IDs for the three sets, the resulting *Endpoints* should belong, to are passed as parameters.
- Check if the desired combination of sets makes any sense. For example it would be rather contradictory to request all documents that are English **and** French.
- Request the handler on the *Set Container*.
- Request the three sets from the *Set Container*.
- Calculate the intersection of the first two sets.

- Calculate the intersection of the result of the previous step and the third set.
- Request the handler on the *Endpoint Container*.
- Create an empty result-collection of *Endpoints*.
- For each *Endpoint*-ID in the result set of the second intersection do the following steps:
  - Retrieve the according *Endpoint* from the *Endpoint Container*.
  - Add the *Endpoint* to the result-collection.
- return the result-collection.

Chapter 8

Summary and Outlook

The *Dinopolis* middleware framework allows for distributing documents and services of any kind transparently. The *Interrelation* mechanism provides all functionality that is needed to model hyper-dimensional relationships between these distributed objects. Because of the stable object addressing, that *Dinopolis* provides, there is no danger of dangling links. Further, complex semantics of relationships can be modeled easily by assigning various properties to each connection between a *Dinopolis Object* and an *Interrelation*. Of course *Interrelations* allow to implement links to parts of objects. Transclusions are possible too.

These concepts allow application programmers as well as end users to model structured component spaces in distributed systems on their particular level of abstraction. I.e. application programmers will primarily model relations between components and objects whereas end users will mainly design document structures.

By now the core modules of the *Dinopolis* framework and therefore all its basic functionality are implemented. The development of a prototype of a study record management system, based on *Dinopolis*, (called ESRM (Electronic Study Record Manager)) is in full swing at the moment. Without any doubt this will reveal the benefits of the *Dinopolis* system. On the other hand this prototype development will for sure put us in a position to detect some flaws of the overall system.

Some of the features that *Dinopolis*' *Interrelation* mechanism offers are rather new. The next few months or years will show to which extent they will be used by application programmers and end users respectively. But it is already foreseeable that most of the features will become accepted in the long run!

References

[Berners-Lee, 2000a] Berners-Lee, T. (2000a). Weaving the web.

[Berners-Lee, 2000b] Berners-Lee, T. (2000b). *Weaving the Web*. Harper San Francisco.

[Berners-Lee and Connolly, 1995] Berners-Lee, T. and Connolly, D. (1995). RFC 1866: Hypertext Markup Language — 2.0. Status: PROPOSED STANDARD.

[Berners-Lee et al., 1996] Berners-Lee, T., Fielding, R., and Frystyk, H. (1996). RFC 1945: Hypertext Transfer Protocol — HTTP/1.0. Status: INFORMATIONAL.

[Berners-Lee et al., 1998] Berners-Lee, T., Fielding, R., and Masinter, L. (1998). RFC 2396: Uniform Resource Identifiers (URI): Generic syntax. Status: DRAFT STANDARD.

[Bluemlinger et al., 2003] Bluemlinger, K., Dallermassl, C., Haub, H., and Zambelli, P. (2003). Object Life-Cycle Management in a Highly Flexible Middleware System. 0(0). http://www.dinopolis.org/download/files/jmlc2003_object_life_cycle_man% agement.pdf.

[Blümlinger, 2000] Blümlinger, K. (2000). Advanced Link Management in Hypermedia Systems. Master's thesis, IICM, Graz University of Technology. available online http://www.dinopolis.org/documentation/misc/theses/.

[Booch, 1987] Booch, G. (1987). *Software Components with Ada: Structures, Tools, and Subsystems*. Benjaming-Cummings.

[Bush, 1945] Bush, V. (1945). As we may think. *The Atlantic Monthly*, 176(1):101–108.

[Cailliau, 1995] Cailliau, W.-R. (1995). A little history of the world wide web.

[Carr et al., 1995] Carr, L., Roure, D. D., Hall, W., and Hill, G. (1995). The distributed link service: A tool for publishers, authors and readers.

[Dallermassl et al., 2000a] Dallermassl, C., Haub, H., Krottmaier, H., Schmaranz, K., and Zambelli, P. (2000a). Using highly sophisticated middleware for building arbitrarily distributed teaching environments. *Proceedings ICCE/ICCAI 2000: Learning Societies In The New Millennium: Care ativity, Caring & Commitments*, pages 1439–1442.

[Dallermassl et al., 2000b] Dallermassl, C., Haub, H., Maurer, H., Schmaranz, K., and Zambelli, P. (2000b). Dinopolis - a leading edge application framework for the internet and intranets. *Proceedings WebNet 2000*, pages 111–116.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley.

[Gromov, 2003] Gromov, G. R. (2003). Internet history.

[Haselwanter, 2003] Haselwanter, E. (2003). Aspects of component composition in distributed frameworks. Master's thesis, IICM, Graz University of Technology.

[Maurer, 1996] Maurer, H. (1996). *HyperWave - The Next Generation Web Solution*. Addison-Wesley.

[Nelson, 1987] Nelson, T. (1987). *Literary Machines*. The Distributors, Indiana.

[Objects, 97] Objects, D. D. I. N. (97). Freismuth d. and helic d. and meszaros g. and schmaranz k. and zwantschko b. *Proceedings EdMedia 97*.

[Schmaranz, 2002a] Schmaranz, K. (2002a). Dinopolis - A Massively Distributeable Componentware System. Habilitation Thesis.

[Schmaranz, 2002b] Schmaranz, K. (2002b). Dolsa - a robust algorithm for massively distributed, dynamic object-lookup services. submitted to J.UCS.

[Szyperski, 2002] Szyperski, C. (2002). *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 2 edition.

[Thalauer, 2004] Thalauer, S. (2004). Aspects of interrelations in distributed component systems. Master's thesis, IICM, Graz University of Technology.

[W3C, 2000] W3C (2000). Document object model (dom) level 2 traversal and range specification.

[W3C, 2001] W3C (2001). Xml linking language (xlink) version 1.0.

[W3C, 2002] W3C (2002). Xpointer framwork.

[W3C, 2004a] W3C (2004a). Extensible markup language (xml) 1.0 (third edition).

[W3C, 2004b] W3C (2004b). Hypertext markup language (html).

[W3C, 2004c] W3C (2004c). Links in html documents.

Glossary

CORBA: *Common Object Request Broker Architecture* An architecture and specification for creating, distributing, and managing distributed program objects in a network. It allows programs at different locations and developed by different vendors to communicate in a network through an "interface broker." CORBA was developed under the auspices of the OMG (Object Management Group) and has been sanctioned by both ISO and X/Open as the standard architecture for distributed objects (also known as components).

DINO: *Distributed Interactive Network Objects* A Java library which was developed by a team of researchers at the IICM, Graz University of Technology. It was presented at the CeBit'99 in Hannover as the core of the MTP (Medical Telematics Platform).

KISS: The KISS (Keep It Small And Simple) principle is often invoked when discussing design to fend off creeping featurism and control complexity of development. In short this means that classes shall not have fat interfaces and methods shall fulfill exactly one atomic task according to their abstraction level. The usual explanation for this acronym is "Keep it Simple, Stupid", others are "Keep it Simple and Stupid" (often in artificial intelligence) or "keep it small and simple". The acronym was originally used as a piece of advice from lawyers counseling their clients.

Schema: XML Schemas (or Schemes) express shared vocabularies and allow machines to carry out rules made by people. They provide a means for defining the structure, content and semantics of XML documents.

XSLT: eXtensible Stylesheet Language Transformation is a language for transforming XML documents into other XML documents. XSLT is designed for use as part of XSL, which is a stylesheet language for XML. In addition to XSLT, XSL includes an XML vocabulary for specifying formatting. XSL specifies the styling of an

XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary. XSLT is also designed to be used independently of XSL. However, XSLT is not intended as a completely general-purpose XML transformation language. Rather it is designed primarily for the kinds of transformations that are needed when XSLT is used as part of XSL.

List of Acronyms

API .... Application Programming Interface

CCM .... CORBA Component Model

CT .... Computer Tomography

DAG .... Directed Acyclic Graph

DINO .... Distributed Interactive Network Objects

DLR .... Deutsches Zentrum für Luft und Raumfahrt (German Aerospace Center)

DLS .... Distributed Link Service

DOLSA .... Distributed Object Lookup Service Algorithm

DOM 2: RANGE Document Object Model Level 2 Traversal and Range

ESRM .... Electronic Study Record Manager

GUH .... Globally Unique Handle

HTML .... HyperText Markup Language

HTTP .... HyperText Transfer Protocol

IICM .... Institute for Information Systems and Computer Media

IREPID ...... *Interrelation Endpoint* ID

KISS .... Keep It Small And Simple

MRI .... Magnetic Resonance Imaging

MTP.......... Medical Telematics Platform

SGML .... Standard Generalized Markup Language

URI........... Uniform Resource Identifier

W3C.......... World Wide Web Consortium

WWW........ World Wide Web

XLink .... XML Linking Language

XML.......... eXtensible Markup Language

XPath .... XML Path Language

XPointer .... XML Pointer Language

XSLT .... eXtensible Stylesheet Language Transformation

Index

A

access control mechanism 2

accounting 16

address 31

Address Management Module 31

addressing 2, 26, 34

application programmer 34

attach 41

B

Booch, Grady 26

Bush, Vannevar 15

C

CeBit 23

component 8, 26

component model 2

componentware 8

connection 33

D

detach 41, 49, 62

Deutsche Luft und
    Raumfahrtsgesellschaft 1

Dinopolis 1, 10

Dinopolis architecture 27

Dinopolis Object 3, 26, 28, 32, 38

Distributed Object Lookup Service
    Algorithm 31

DLR 1

document structure 33

DOLSA 31

Dynamic Type 2, 3, 29, 38, 40, 43

Dynamic Type Mechanism 29

E

Endpoint 35, 38

Dinopolis Object side 47

Interrelation side 43

Endpoint Classification Set . 35, 38, 39, 51

default 57

explicit 56

implicit 56

Endpoint Classification Set ID 57

Endpoint Container 44

Endpoint handle 57

Endpoint ID 47

Extensible Markup Language 19

External System 3, 26, 33

G

globally unique handle 3, 7, 26, 31

Graz University of Technology 23

group of connections 33

GUH 3, 26, 43, 47

H

Haselwanter, Edmund 29

HTML 14, 18

HTTP 14

hyper link 9

hyper-dimensional relation 51

Hyper-G 20

Hyper-G Data Model 20

hyperlink 16, 51

hypermedia 15

hypertext 15

HyperText Markup Language 14

HyperText Transfer Protocol 14

Hyperwave 1, 20

I

IICM 1, 23

Implicit Interrelation 33, 35

information system 17, 32

Interrelation 9, 26, 37, 40

Interrelation Endpoint ID 43

Interrelation Handler 49

Interrelation Management Module .. 37

Interrelation Manager 37–39

Interrelation mechanism 32

Interrelation part 46

intersection operation 51

IREPID 43, 47

J

Java 24

K

KISS 8

L

life cycle 40

link 14

- 1:1 .... 18, 23
- hyper-dimensional 9
- multi directional 9
- simple 9
- typed 9, 19
- unidirectional 18
- unstable 18

link mechanism 32

LUH 47

M

memex 15

meta data . 8, 19, 28, 33, 34, 44, 45, 48, 57

metadata 2, 38

minus operation 51

MTP 1

multi directional 33

N

Nelson, Ted 15, 18

O

Object Management Module 27

object proxy 28

object relationships 14

object subset 47

open source 23

Operation 8

operation 28

P

persistence 34

properties 41

properties of a connection 35

property 39

R

reconfiguration 32

relation 32

resolve 33, 35, 42

resolve operation 33

rights management 18

robustness 9

S

security 2, 28

semantic set 33, 35

semantics 32, 34

service 8, 28

Set 39, 41

set 38, 39, 51

Set Container 39, 58

SGML 19

software requirements 34

Static Type 3, 29, 38, 43

storing 34

T

Thalauer, Stefan 1

transaction 2

transclusion 16, 32

type 32, 34, 43, 44

U

union operation 51

V

version management 18

Virtual System 31, 33

W

Web 17

World Wide Web 14, 17

WWW 14, 17

X

Xanadu 15, 18

XLink 19

XML 19