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

Aspects of component composition in distributed frameworks

submitted by:

Edmund Haselwanter

Oktober 2003

Institute for Information Processing
and Computer Supported New Media

Supervisor: Univ.Doz. DI. Dr. techn. Klaus Schmaranz

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

Aspekte der Komponenten Komposition in verteilten Systemen

vorgelegt von:

Edmund Haselwanter

Oktober 2003

Institut für Informationsverarbeitung
und Computergestützte Neue Medien (IICM)

Begutachter: Univ.Doz. DI. Dr. techn. Klaus Schmaranz

Abstract

The present thesis discusses aspects of component composition in distributed frameworks. Distributed components are the central part of *Dinopolis*, a distributed component middleware framework.

The main concepts of componentware, distributed object systems, and middleware as well as the main concepts of *Dinopolis* are introduced. The *Dinopolis Object* definition is presented and the requirements to the *Dinopolis Objects* are outlined. The life-cycle of *Dinopolis Objects* and the processes and algorithms concerning *Dinopolis Objects* are introduced. Finally the *Dynamic Type* mechanism, which makes it possible to add and remove functionality to/from *Dinopolis Objects* at runtime, is discussed.

Kurzfassung

Die vorgelegte Diplomarbeit beschreibt Aspekte der Komponenten Komposition in verteilten Systemen. Verteilte Komponenten sind ein zentraler Punkt in *Dinopolis* einem verteilten Komponenten Middleware Framework.

Die Konzepte von Konponenten Systemen, verteilten Objekt Systemen und Middleware sowie die Konzepte von *Dinopolis* werden vorgestellt. Der *Dinopolis Objekt* Begriff wird definiert und die Anforderungen an *Dinopolis Objekte* werden präsentiert. Der Lebenszyklus eines *Dinopolis Objektes* und die ein *Dinopolis Objekt* betreffenden Prozesse und Algorithmen werden vorgestellt. Schlussendlich wird der *Dynamic Type Mechanismus* erläutert, welcher es erlaubt, zur Laufzeit Funktionalität zu einem *Dinopolis Objekt* hinzuzufügen bzw. wieder zu entfernen.

Acknowledgments

First of all I want to thank Klaus Schmaranz. He offered me the possibility to work in the IICM *Dinopolis* project and always had time for inspiring discussions. I want to thank the whole *Dinopolis* team for the warm welcome and support whenever I had demands or questions.

I would like to express my gratitude to the following people for their support and assistance: Again Klaus Schmaranz for lecturing my thesis. Heimo Haub and Philip Zambelli for the great team work and the inspiring discussions. Thomas Oberhuber and Evelin Fisslthaler who had always an ear for me and for proofreading my thesis.

Last but not least thanks to my family, especially to my parents who made all this possible and supported me throughout my whole studies.

Table of Contents

List of Figures

Chapter 1

Introduction

This master thesis is focused on *Software Components*, their runtime composition and execution environment and arose within the development of the *Dinopolis Middleware System*. Development of the *Dinopolis Middleware Framework* is subdivided into smaller teams of about two to four people. I started my work at the beginning of the detailed design phase of the *Object Management* module. My team members Heimo Haub and Philip Zambelli work on their Phd thesis in the area of distributed object frameworks, component software development and middleware architectures.

The *Dinopolis Middleware System* is designed as a distributed component middleware framework [Sch02b, Sch02c]. *Dinopolis* has its origins in DINO 2.0 (Distributed Interactive Network Objects) which was developed by a team of researchers at the IICM[1], Graz University of Technology, and was presented at the CeBit'99 in Hannover as the core of the MTP (Medical Telematics Platform) prototype. The MTP project has been developed together with several researchers of the DLR[2].

The *Dinopolis Middleware System* is a middleware layer on top of existing systems. Some key features of *Dinopolis* are:

[1]Institute for Information Processing and Computer Supported new Media

[2]Deutsches Zentrum für Luft- und Raumfahrt/German Aerospace Center

- A completely platform and technology independent design. Implementations in any mainstream object oriented programming language are possible. Prototype implementations of parts of the system are written in Java. The current implementation language is C++.
- A highly modular and extensible architecture with a very slim kernel. Logging, security, transaction, versioning etc. are implemented as modules that can be loaded on demand when needed.
- A flexible embedder concept allowing to integrate and combine already existing external systems such as file-systems, databases, web-servers and active systems like collaboration or communication systems etc.
- A highly sophisticated addressing mechanism providing globally unique handles for components or parts of them. These addresses are robust against object or component movement within the system (see [Sch02a]).
- A runtime configurable, distributed and robust component model allowing objects or components to reside anywhere on the network.
- A highly flexible object and component interrelation mechanism which provides freely typed $n_1 : n_2 : ... : n_n$ relations between arbitrary objects and components (see [Sch02d]).
- A highly sophisticated, adaptable role based security concept.

In the sense of a middleware framework it is possible to embed arbitrarily many already existing external systems [Ber96]. Schmaranz states in his Habilitation thesis[Sch02b]:

> “In order not to develop 'just another system' which overcomes identified problems but causes others and therefore leads to religious discussions, Dinopolis follows a different path: *Dinopolis is a middleware layer on top of existing systems that utilizes and combines their features by means of a highly sophisticated embedding mechanism.*”

These features are provided through *Distributed Objects* which can be requested by applications. These distributed objects are called *Dinopolis Objects*. *Dinopolis Objects* represent addressable units in the *Dinopolis Middleware System*. A *Dinopolis Object* is known to the *Dinopolis Middleware System* by a stable, globally unique identifier which never changes throughout the whole life-cycle of this *Dinopolis Object*. This identifier is robust against move-operations within the *Dinopolis Middleware System*.

Another feature of *Dinopolis Objects* are relational dependencies between *Dinopolis Objects*. Relational dependencies between *Dinopolis Objects* can be modeled with so called *Interrelations*. This enables robust navigation through e.g. *Dinopolis Objects* representing documents in arbitrary many different languages.

One key feature of *Dinopolis Objects* is that they are reconfigurable at runtime. This means that functionality of the *Dinopolis Middleware System*, which naturally includes embedded external systems, can be added to a *Dinopolis Object* on demand, or removed from a *Dinopolis Object* when it is not needed anymore. To achieve this, all functionality is provided through small program units called *Definitions* which represent components in the sense of *Software Components* as discussed in chapter 3. *Dinopolis Objects* are realized as composites or containers of such components. This can be seen as a kind of class inheritance at runtime. Newly attached *Definitions* may override already attached implementations or add completely new functionality to *Dinopolis Objects*. Mainstream object oriented programming languages don't provide the needed technologies at language level. However, object composition, class or interface inheritance at compile-time is not enough to realize this kind of functionality. Several mechanisms have been introduced to solve the arising problems. An extended life-cycle management compared to objects known from object oriented programming languages has been developed as well as a mechanism to be able to change the target of method invocation to runtime-added implementations.

Since *Dinopolis* is designed as a middleware layer, network transparency has a very big influence on the design of the *Dinopolis Objects* and the overall

Dinopolis System Architecture. *Dinopolis* is a distributed object framework. *Dinopolis Objects* are addressable units in the *Dinopolis Middleware System* and themselves network transparent which means it makes no difference if a local or remote instance of a *Dinopolis Object* is used.

Dinopolis supports a highly sophisticated, role based security concept. This has a high impact on the design of the *Dinopolis Objects* since it has to be possible to intercept any call and response on/from a *Dinopolis Object*.

A brief overview about the modules of the *Dinopolis System Architecture* is given in chapter 4, for an in depth presentation of the *Dinopolis Middleware System* please have a look at [Sch02b].

1.1 Chapter Overview

Chapter 2, Motivation:
Software developers have to deal with a very heterogenous environment. Applications should run on different operating systems and hardware architectures and interoperate with other applications running on other systems. In this chapter a real world scenario is outlined to illustrate the requirements which have to be met by a system architecture dealing with the arising problems. A distributed component middleware framework is identified as the system architecture of choice. Based on the introduced features and requirements, the context of this master thesis within the *Dinopolis Middleware Framework* is established.

Chapter 3, Technologies:
The software developing process evolved over time. The used tools and technologies follow certain paradigms and have their benefits and drawbacks. *Dinopolis* is designed as a distributed component middleware framework. The design relies on several underlaying technologies. This technologies are presented in this chapter. Starting with the historical evolution from structured procedural programming to component based software development,

a closer look at *Software Components*, *Distributed Objects*, and *Middleware Frameworks* is presented.

Chapter 4, The Dinopolis Middleware System:
The *Dinopolis* middleware framework is the distributed component middleware framework of choice. The goal of *Dinopolis* is to provide services and operations in a heterogenous environment in a uniform fashion. For this aim the technologies presented in chapter 3 are bundled together to provide the services and operations through so called *Dinopolis Objects*. The internals of the *Dinopolis Object* make up the main part of this master thesis. *Dinopolis Objects* heavily rely on their supporting environment, the *Object Management* module. In this chapter the context between *Dinopolis Objects*, the *Object Management* module and the *Dinopolis System Architecture* is established. First the *Dinopolis Object Definition*, describing the parts of a *Dinopolis Object* is presented followed by a brief overview of the *Dinopolis System Architecture* presenting the environment of the *Object Management* module.

Chapter 5, Use Cases and Requirements:
The *Object Management* module as well as *Dinopolis Objects* have to meet certain requirements. These requirements are a result of several use-cases during the analysis phase of *Dinopolis* and its predecessors. In this chapter use-cases and requirements concerning the *Object Management* module and the *Dinopolis Objects* are presented. The chapter starts with uses-cases illustrating the application programmers' view, followed by the general requirements for the *Object Management* module, closing with requirements for the internal structure of *Dinopolis Objects*, namely operations, services, content, interrelations and meta-data.

Chapter 6, The *Object Management* module:
The *Dinopolis Middleware Framework* has a highly modular architecture. Functionality concerning *Dinopolis Objects* is provided by the so-called *Ob-*

ject Management module. *Dinopolis Objects* and the *Object Management* module heavily rely on the functionality provided from the other modules. The first part of this chapter is focused on the relations and dependencies between the *Object Management* module and the other modules of *Dinopolis*. In the second part of this chapter the sub-modules of the *Object Management* module are introduced.

Chapter 7, The *Dinopolis Object*:
Application programmers using the *Dinopolis Middleware Framework* will use its provided functionality through *Dinopolis Objects*. According to the *Dinopolis Object* definition presented in section 4.1 *Dinopolis Objects* consist of several parts. Their sub-modules are presented at the beginning of this chapter. Since *Dinopolis Objects* are so-called long-living objects, their lifecycle is introduced next. The main part of this chapter is dedicated to the processes and algorithms for creating, deleting, loading and storing *Dinopolis Objects*.

Chapter 8, The *Dynamic Type* mechanism:
Dinopolis Objects heavily depend on a mechanism, which allows adding, and removing functionality at runtime. Mainstream object-oriented programming languages like C++ or Java don't provide an appropriate mechanism at the language level. This chapter starts with the clarification of needed terms followed by a presentation of how the mechanism works in detail.

Chapter 9, Summary And Outlook:
The *Dinopolis Middleware Framework* is in its final developing phase. Most of the modules are at the very end of the detailed design stage. As a proof of concept two prototype variants of the *Dynamic Type* mechanism were implemented showing that this mechanism works as expected. In this chapter, the results are summarized and an outlook on further development of the *Dinopolis Middleware Framework* is presented.

Chapter 2

Motivation

We live in a networked world. Traditional computers, notebooks, PDAs, mobiles and several other electronic devices are in some way interconnected. Therefore, software developers have to deal with a very heterogenous environment. It is impossible to foresee all future technologies and standards, so the design of a software product or architecture has to be very flexible and configurable. It must be possible to easily reuse or integrate already existing solutions of a productive environment. This has a high impact on the design of a software product or architecture. Re-inventing the wheel is very expensive in both time and money.

This is not only true for the several system architectures themselves, but also for the information, which is processed with programs on top of them. It should be easy to access the stored information in different ways and formats depending on the requesting client or usage. Consider a movie in mpeg2[1] format. A copy operation which normally doesn't need to interpret the requested data can work with any binary representation of the movie whereas a special client like a movie player may request the movie over a network connection as a downsampled, mpeg4 encoded video stream to save network bandwidth or, if used locally, request the native mpeg2 format.

[1]Moving Picture Experts Group `http://www.chiariglione.org/mpeg/`

2.1 A real world scenario

Let's discuss a sketched real world scenario namely a distributed medical information system to work out the requirements for a software architecture dealing with the problem areas outlined above:

Consider an emergency situation like a car-accident. An emergency ambulance with an emergency doctor onboard already has arrived on the scene. The emergency ambulance is equipped with a mobile device providing access to a distributed medical information system. This medical information system deals with electronic patient records and provides a security checked request and look-up service.

As an assumption let us say the identities of the concerned people like their name or any other appropriate information can be found out immediately. With this starting point the emergency doctor could request a individual electronic patient record for each concerned person from the distributed medical information system. Some essential information like blood type or special medication are immediately available. The emergency doctor could add his initial diagnosis to the corresponding electronic patient record on the spot.

Based on this initial diagnosis further needed inpatient treatment could be prepared by the emergency admission at the hospital which themselves can request the updated electronic patient records. While the transportation of the patients is still going on, one of the x-ray cabins at the hospital could be prepared for e.g. an x-ray of the left leg. At the hospital the patient's left leg is then x-rayed and the resulting image is added to his/her electronic patient record. The doctor on duty has all the relevant data from birth to present at his mouse-click.

2.2 Key features of the overall system

With the scenario sketched above some of the features needed by such a system architecture can be outlined. Please keep in mind that it is not subject of this thesis to give an in depth summary of all requirements to the *Dinopolis Middleware System*, but to present what is expected from a proper component model and distributed object environment. Nevertheless, to understand the requirements to the component model and its environment a discussion about the features of the overall system architecture is essential.

The following key features of the overall system and their implications can be identified:

- First of all the patient record has to be located. The physical location of this record can be anywhere in the system. For this to work, each electronic patient record needs a stable and unique address within the system and the system must provide a distributed lookup service for locating an electronic patient record.
- The system architecture must support mechanisms to integrate arbitrary external systems. It must be possible to pass through their full functionality. With this an application can be implemented, which supports attaching of e.g. an x-ray image of an embedded x-ray machine.
- An electronic patient record usually consists of several chunks of data: The doctors reports, x-ray data and so on. Due to national law this data cannot be archived anywhere but in the appropriate institutions. This leads to the requirement of interrelations between distributed addressable units in the system.
- Since storing personal data is a very sensitive area, the system architecture must provide security mechanisms to ensure that only users with proper access-rights can carry out certain tasks.
- With proper access-rights it has to be possible to navigate to the re-

lated data-fragments. Due to this fact the interrelations must support functionality for navigating through the system.

Within the development process of the *Dinopolis Middleware System* at the IICM and the DLR several additional key features of the overall system were identified:

- In order not to be bound to a special platform or the availability of a certain technology, the design of the system architecture has to be platform and technology independent.
- The core system architecture has to be as slim as possible allowing to use special features via loadable modules.
- It must be possible to combine embedded external systems to form more powerful so-called *virtual systems*.
- Due to several reasons it must be possible to move an electronic patient record from one physical location to another. It must be guaranteed that after this move operation of an electronic patient record it keeps its initial unique address and exists only once in the system.

To achieve this, we need a flexible, adaptable, runtime re-configurable and network-transparent software-architecture. The presented key features easily lead to a very complex design, so one of the main goals is to reduce the complexity of the software design and build a very modularized system.

Like the object oriented programming paradigm helps reducing the complexity by hiding the implementation details - one doesn't need to know how a object carries out a certain task, but uses its public interfaces and therefore its well known methods - it would be preferable to just instantiate an object for a specific task and work with its provided services and operations, instead of directly communicating via a very specialized API of a certain resource.

What we need is another higher level of abstraction. Hence, the fact that we want to design a massively distributed system, dictates a middleware approach which provides a uniform network-transparent access to the underlying resources. With this we are able to embed arbitrarily many platform dependent resources (e.g. file-systems, databases, web servers) in a platform independent fashion.

Additional features like a highly sophisticated security system or transactions can be provided by this middleware too.

Another problem, which is not addressed by the middleware-paradigm is runtime re-configuration or adaption of the used/provided distributed objects. This and all the problems concerning these objects which arise out of the distribution-context must be solved with a proper component-model within this middleware architecture.

Chapter 3

Technologies

When talking about a distributed component middleware framework it is worth having a closer look at the underlying technologies. They all have their implications on the design of *Dinopolis Objects* and thus are presented in this chapter. We will have a closer look on components and component-based software development, distributed objects and middleware frameworks. These technologies appeared and evolved over time and offer useful abstractions to keep things understandable and re-usability easy. We will start with a short history of software development and the programming paradigms of each point in time.

3.1 History

M. D. McIlroy announced a *software crises* at the NATO conference on software engineering in Garmisch, Germany, 1968. In his talk *Mass Produced Software Components* he was the first to introduce components to overcome this crisis [McI69]. He stated, that compared to the hardware industry, the software industry is less industrialized, e.g. there are no catalogs of standard parts or even standard parts at all. It is not possible to compose products from readily available components which is what developers are used to in the field of electronics or mechanics.

The use of components would automatically lead to a massive reuse of existing software parts. Reuse primarily has economic benefits like lower cost and effort which leads to less time to market. But reuse can help raising the quality of software as well.

Reuse of code fragments, or even whole parts of software, is a broad field of research [Mey96]. Many attempts have been made to develop powerful tools and techniques to solve certain problems.

Another important aspect of the software development process is to reduce the complexity of software systems to make them easier to develop, understand and maintain. This aspect is addressed by components as well, since they are most likely much smaller units of code and they are furthermore loosely coupled to form a software product. Since McIlroy's talk several improvements in software developing methods and programming language design took place.

3.1.1 A first attempt: Structured procedural programming

In the late 1960's and early to mid 1970's structured programming became popular. Structured programming evolved from the procedural programming paradigm. The idea is to break up a big system into small parts and use additional, hierarchical program flow structures on that smaller units (e.g. for, while, if-then-else) (see [Dij70], section "on understanding programs" and [Wir71]). Most of the time this is combined with a top-down design process where the whole system is split up into smaller parts. These parts are themselves split up into more smaller parts at the size of functions or procedures which are implemented, tested and then assembled together to build up the program.

Some of the advantages compared to former programming methods (e.g spaghetti code) are as follows:

- The program need not be understood as a whole by all developers. Programming can be done in teams where each team member is responsible for his/her assigned task.

- Changes remain local to the blocks, functions or bigger parts like files. This enhances the maintainability of the produced code immensely.
- Several parts of programs can be grouped together to form libraries of routines and thus are subject to reuse. Reusing already developed code reduces cost and time to market dramatically.

But structured programming is focused on functions and algorithms. The processed data is considered as something external. This again led to copy and paste of code, because relationships between data types were not considered important. Functions and procedures can be packed together forming libraries. Function-oriented libraries are mostly just usable in the original development context and difficult to reuse otherwise. There are a lot of success stories for libraries. E.g. in the field of mathematics it is natural to think in functions and there are many good mathematical libraries.

More or less parallel to the upcoming programming languages, supporting the structured procedural programming paradigm another, big step further in reliable software development took place. Programming languages became aware of user-defined data types. This enhances readability and writability of code by a great amount and allows for type- checking in a more problem-domain specific fashion. In combination with static or strong typing (see [Seb93]) a lot of errors can be determined at compile time.

3.1.2 The next revolution: The object-oriented paradigm

Object-oriented programming has the focus on - *nomen est omen* - objects. The real world gets modeled with objects. The data is kept close to the relevant code. Object oriented programming became dominant in the mid 1980's largely due to the influence of C++, an extension of the C programming language, and the rising popularity of graphical user interfaces where thinking in objects like windows, toolbars, menus etc. is more natural than thinking in functions and procedures.

An object-oriented programming-method or language supports at least the following four key concepts.

Abstraction: This are the essential defining characteristics of an object that make it distinguishable from all other objects (depending on the viewer). This expresses to the *outside* what can be done with an object without revealing how this is done internally.

Encapsulation: Also known as *information hiding*. Prevents other objects manipulating its internals directly in unexpected ways. Encapsulation serves separating the contractual interface from its implementation.

Inheritance: Relationship between classes. A subclass can inherit behavior or structure from one (single inheritance) or more (multiple inheritance) superclasses. Further subclassing forms a *Is-A* hierarchy among classes. Subclasses may augment or redefine existing structure and behavior of their superclass or superclasses.

Polymorphism: The ability to appear in several forms. Whenever an object of a specific type is expected every object which type is a subtype can take its place.

A programming language not supporting inheritance is called object-based. With these powerful concepts more complex software systems can be build [Boo94]. Abstraction and Encapsulation help to separate the declaration from the implementation where the declaration forms a sort of *contract* between the client of a class and the class itself.

3.2 From Objects to Components

Why use components? To answer this question we have to look at the benefits and drawbacks of components by means of software development. This section covers the advantages and disadvantages of *Component Based Software Engineering* (CBSE, see [KB98]) or *Component Based Development*

(CBD, see [MM99]) respectively over other techniques like functional decomposition in structured programming or object-oriented analysis, design and programming.

3.2.1 About Software Components

It is very difficult to give a generally accepted definition for the term component in the field of software development. A definition heavily depends on the context and the abstraction level in which the software component is used. In the literature several definitions depending on the point of view are given. A relatively narrow definition is given by Clemens Szyperski, who states in his book *Component Software: Beyond Object-Oriented Programming* [Szy02]:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties"

A more general definition is given by Grady Booch in his book *Software Components with Ada: Structures, Tools and Subsystems* [Boo87]

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

Bertrand Meyer also offers us a wide definition for software components [Mey99]:

> "A software component is a program element with the following properties:
>
> - The element may be used by other program elements (*clients*)

- The clients and their authors do not need to be known to the element's authors"

A chronological overview about several definitions can be found in [Szy02].

Looking at the definitions presented above a software component is a "thing" which can be deployed in some form and then plugged together with other software components to form ever-new, more complex structures e.g. applications (see 3.2.2). Assembling together already existing software components makes them a subject to extensive reuse which naturally has a high impact on their design [Mey96].

Software components can be identified in different levels of abstraction and granularity. Modern operating systems provide an environment for applications which can be seen as components at this abstraction level. Relational database engines can be named components too.

Smaller programming elements are used by several plug-in architectures. Plug-in architectures became very popular with multimedia content on the Internet and are used by all well-known web browsers nowadays. Some functionality is *outsourced* to specialized components which can handle either whole document classes like the Acroread plug-in or embedded multimedia content like movies or animations (Apple Quicktime, Macromedia Flash). Plug-ins allow the extension of applications with ever new functionality without having to modify the application itself.

Another widespread approach of software components is the *Visual Basic Extension* technology (short VBX) from Microsoft released in 1991. This technology originally designed for the Visual Basic language environment enables software developers to quickly assemble applications. VBX evolved over OCX (from OLE Control Extension where OLE stands for Object Linking and Embedding), the second generation control architecture for the Microsoft-OS environment to the ActiveX technologies. ActiveX is, strictly speaking, a brand name for a changing bundle of technologies like COM/-COM+ (Component Object Model), DCOM (Distributed COM) and the

.net environment. Famous ActiveX containers are Microsoft Word, Excel and Internet Explorer. All these technologies heavily depend on the Microsoft Windows operating system

Another family of technologies comes from Sun Microsystems who provide several component models like JavaBeans, Servlets, Enterprise Java Beans (short EJB) and J2EE Application Components which all are build on top of the Java programming language. The main advantage of the Java approach over the Microsoft strategy is that Java components are not bound to a specific operating system but interpreted as bytecode by a virtual machine. Virtual machines are available for all major operating systems and offer another benefit compared to ActiveX, namely sandboxing, where the system environment is guarded against the executed program elements.

There are several other fields in the area of software development where software components are used nowadays and many other approaches to achieve CBSD. Whenever it is convenient, a closer look at one of the approaches is presented.

CBSD is a very young discipline. A lot of problems just surfaced and are not solved yet. Summarizing the identified characteristic properties of software components are:

(i) A component is a unit of (third party) composition.

(ii) A component is a unit of deployment.

(iii) Components can be identified in different levels of abstraction.

(iv) Components need an environment where they can *live* in.

(v) Components may be *local* or *distributed*. Local means that their execution is confined to just one machine whereas distributed components allow distributed execution across multiple machines [Emm02].

3.2.2 Components are for composition

Components are designed to be usable for composition. Prefabricated units can be composed and thus reused to ever-new composites. This is simple and straightforward. But it turns out that producing such *prefabricated units* is not that easy. Several restrictions come into play depending on the level of abstraction and the nature of a software component.

Who will compose the components? Following the definitions of Szyperski and Meyer, components are subject to third party composition where the author of the component doesn't need to be known by the developer dealing with the component.

A software component can be offered to third parties in several ways. According to Meyer in [Mey99].

- Binary only, just the interface description available to hide the internals. Most commercial components are provided binary only.
- Source only with nearly no description and little information hiding. To use such a component the developers must read the source code. This is the case for most of the free software available on the Internet.
- Information hiding, with reuse through a well defined interface. Source available for introspection, discussion and correction

Whitebox versus blackbox abstrations and reuse

Reusing a component through the described interfaces only refers to blackbox abstraction. This is a very loose coupling of the client using the component and the component itself. There is only a contract on the interface level. If not changing the interface, evolutions of the used component will not have any impact on the client. In contrast, reusing a component and relying on some internals of a component, e.g. some information gained from inspecting the source code, is a tighter coupling of the client to the component. It is more likely that changes or evolutions of the referred component will brake the client implementation.

Context dependencies

On the other hand a component may require some services from the client to function properly. A component for manipulating a filesystem may specify that it needs a filesystem interface. This must be provided from the execution environment.

3.3 Distributed Objects

Distributed Objects have their origin in distributed computing. Distributed computing refers to operations that run on separated computers which are connected via a network. A common example for distributed computing is ATM (automated teller machine). The computer in the ATM, the computer in the ATM's home bank and the computer in the user's home bank are connected via a network and work together to deliver the user's cash within seconds. Many solutions in the area of distributed computing are proprietary. With the upcoming of ever increasing powerful home computers more complex software products arise with the need for a common standardized technology to provide distributed computing. There are several attempts to solve this aim. The major players in the field of distributed objects nowadays are presented in table 3.1 and discussed in the following sections.

3.3.1 Common steps in distributed computing

There are some parts in distributed computing environments which more or less will be the same in every specific incarnation. One side can be identified

Player	**Architecture**	**Network Object Model Technology**
Object Management Group (OMG)	OMA (Object Management Architecture)	CORBA/IIOP (Common Object Request Broker Architecture/Internet Inter-ORB Protocol)
Microsoft	Windows DNA (Distributed interNet Applications Architecture)	COM/DCOM (Distributed Common Object Model), ActiveX
Sun Microsystems	Enterprise JavaBean Architecture	Java RMI (Remote Method Invocation), JavaBeans

Table 3.1: Major Players, Architectures, and Technologies

as a client where the opposite side then happens to be the server. These roles may not be that static and the roles can change over time, depending on the concrete architecture. Figure 3.1[Sch98] shows an abstraction of a distributed object call.

The steps which have to be carried out by the participates are as follows [Sch98]:

1. The client invokes a method call on a client proxy.
2. The client proxy organizes the input object parameters into a platform-independent byte stream suitable for transport.
3. The Object broker on the client side surveys the network to find the remote object broker that has the particular remote object in its registry. Then the client object broker passes the object ID, the method to be called, and the transport bytes to the remote object broker.
4. The remote object broker connects to the desired remote object via

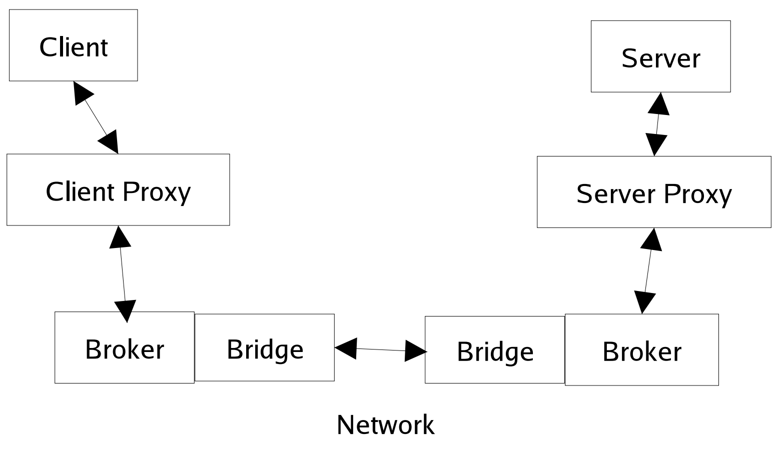

Figure 3.1: Distributed Object Call

a server proxy. Then it passes the transport bytes on to the server proxy.

5. The server proxy transforms the transport bytes back into object parameters and then makes the call to the requested method.

6. Once the method is executed, the results can be passed back to the client application in a similar way.

3.3.2 Remote Procedure Call (RPC)

The first full-scale implementations of the Remote Procedure Call (RPC) concept appeared in the late 1970s and early 1980s. The concept follows a client/server architecture and increases the interoperability, portability, and flexibility of an application using this concept. The application is distributed across several heterogeneous platforms. From the application programmers' view it looks like using normal procedure calls, so called "stub" routines. The stub then actually performs the call across the network to another computer.

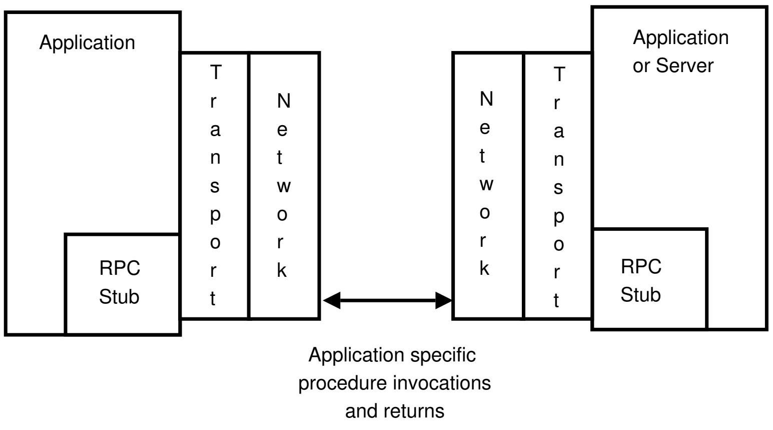

Figure 3.2: Schematic Remote Procedure Call Environment

This reduces the complexity of developing applications that span multiple operating systems. RPC makes network access to resources transparent to the application developers by insulating them from the details of the various operating system and network interfaces.

When the client application calls an RPC procedure, the underlying RPC stub creates and sends a message to another (server-) application via Transport/Network. The application on the server then interprets the message, services the request, and places the return values, if any, into another message which is sent back to the client application. A typical RPC communication is synchronous, which means the client application waits for a reply from the server application before proceeding.

3.3.3 Java RMI

Java provides the Java Remote Method Invocation (RMI) system to enable application programmers to create distributed Java technology-based applications. RMI is basically an object-oriented RPC mechanism and allows for invoking methods on remote Java objects from other Java virtual machines which can be on different hosts.

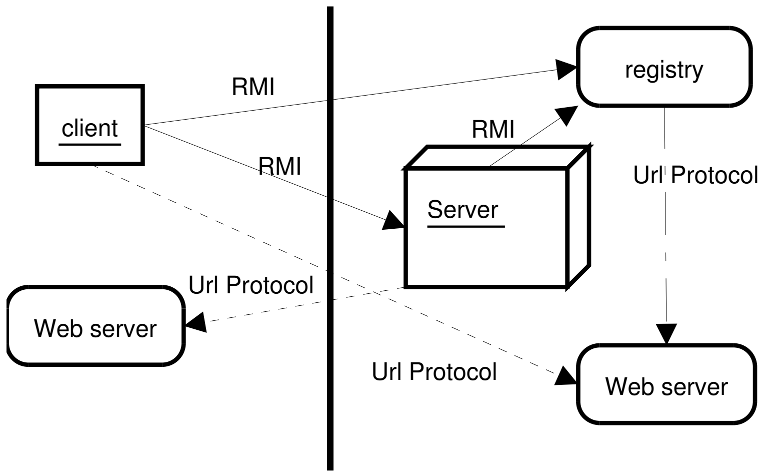

Figure 3.3: RMI distributed application

Figure 3.3 depicts an RMI distributed application. There are three processes that participate in supporting remote method invocation:

1. The *Client* is the process invoking a method on a remote object
2. The *Server* is the process which actually owns the remote object.
3. The *Registry* provides a mapping from names to objects. Objects are registered with the registry. Once an object has been registered with the registry, one can gain access to the object through its name.

All used classes must be known by both, the client and the server. If, for instance, one parameter of a method is itself an object the server needs the class file for this object. If this class file is not already present on the server the RMI class loader allows to load it from e.g. a web server.

JRMP the RMI protocol

RMI usually uses direct socket connections to hosts on the Internet. The standard low-level protocol is called Java Remote Method Protocol (JRMP) and is based on TCP. Some environments, especially Intranets, are protected through firewalls and thus do not allow direct connections. The RMI transport layer therefore wraps this protocol into an HTTP request and sends it via a proxy to the remote location. RMI provides a third protocol, namely Internet Inter-Orb Protocol (IIOP)[1], which enables Java RMI enabled applications to work with objects written in other languages than Java.

Stubs and Skeletons

The RMI functionality is provided through so called stubs and skeletons. Stubs and skeletons are generated by the *rmic* compiler. The client's local representation (proxy) for a remote object is called a stub. The application programmer invokes the method call on the stub which is responsible for carrying out the method call on the remote object.

The following tasks must be carried out by the stub:

1. The stub initiates a connection with the remote JVM containing the remote object.
2. The parameters are marshalled (written and transmitted) to the remote JVM.
3. The stub then waits for the result of the method invocation.
4. The return value gets unmarshalled or an exception occurs.
5. Finally the return value is returned to the caller.

Similar to the RPC mechanism presented above the details of the RMI

[1] `http://java.sun.com/products/rmi-iiop/`

invocation mechanism are hidden from the caller. The network-level communication and the serialization of parameters is done behind the scenes.

The counterpart of the stubs on the client side are the so called skeletons on the server side. The task of the skeleton is to dispatch the incoming method invocation to the actual remote object implementation.

The following steps have to be carried out by the skeleton :

1. The skeleton unmarshals the parameters for the remote method.
2. Then the skeleton invokes the method on the actual remote object implementation.
3. The last step is to marshall the result (return value or exception) to the caller.

3.3.4 Common Request Broker Architecture (CORBA)

The *Common Object Request Broker Architecture* (CORBA) is the heart of the Object Management Architecture(OMA) from the Object Management Group short OMG. The OMG was found in 1989 and operates as a non-profit organization. The OMG is by far the largest consortium in the software industry with approximately 800 members in the early 2002. The mission [2] of the OMG reads as follows:

“The OMG was formed to create a component-based software marketplace by accelerating the introduction of standardized object software. The organization’s charter includes the establishment of industry guidelines and detailed object management specifications to provide a common framework for application development. Conformance to these specifications will make it possible to develop a heterogeneous computing environment across all major hardware platforms and operating systems. Implementations of OMG specifications can be found on many operating systems across the world today.”

[2] `http://www.omg.org/news/about/index.htm`

Architecture

The main idea of CORBA is to enable open interconnections of a wide variety of languages, implementations, and platforms. A client application written in Java running on a Windows 2000 platform can use functionality provided from remote objects written in a different programming language e.g. C++ running on a different server platform e.g. Linux. Therefore several abstraction layers between the client-application and the server-objects provide the needed indirections.

CORBA essentially has three parts:

- A set of invocation interfaces
- The object request broker(ORB)
- A set of object adapters

A simplified structure of an ORB-based system is shown in figure 3.4 [Szy02].

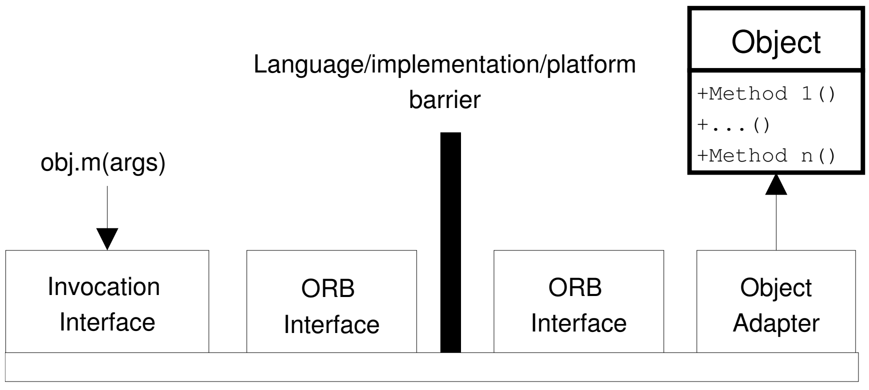

Figure 3.4: Simplified structure of an ORB-based system

The invocation of object operations, so called method invocation require late binding of the implementation. Based on the receiving object's reference the object implementation which actually implements the called method is selected. The invocation interfaces marshals the invocation's arguments such that the ORB core can locate the receiver object and the invoked method and transport the arguments. At the opposite end the object adapter unmarshals the arguments and invokes the requested method on the receiver object [Szy02].

One of the goals of CORBA is to be language independent. This has a strong impact on how to specify the language-depending interfaces of the implementations. The invocation interfaces on the one hand and the object adapters on the other can, and most of the time will be implemented in different programming languages. There must be a common basis of how to describe the interfaces in a language independent fashion, a so called common language. For this aim the OMG has specified the OMG interface definition language (OMG IDL). An example for a OMG IDL specification is presented in the following listing:

```
module Example
{
  struct Time
  {
    unsigned short hour;
    unsigned short minute;
  }
  interface HelloWorld
  {
    string sayHello(in string myName);
  }
}
```

Another requirement to be met is a binding from the common language to a specific language. Today bindings to OMG IDL are available for several languages, including C, C++, Smalltalk, COBOL, and Java. The common language and the bindings to concrete languages allow for constructing generic marshalling and unmarshalling mechanisms for the arguments and relating calls from or to a concrete language to the common language.

3.4 Middleware Frameworks

In an average organization a wide variety of heterogenous hardware can be found. Personal computers, minicomputers, mainframes, and workstations running different operating systems connected by networks. Integration of services is difficult, uneven, and OS dependent. Sometimes applications available on one local area server are not available on other servers because other departments use servers on which the application cannot run. Or the application is available on different servers but some features like printing are not available because there is no support for the printer protocol provided from the remote server.

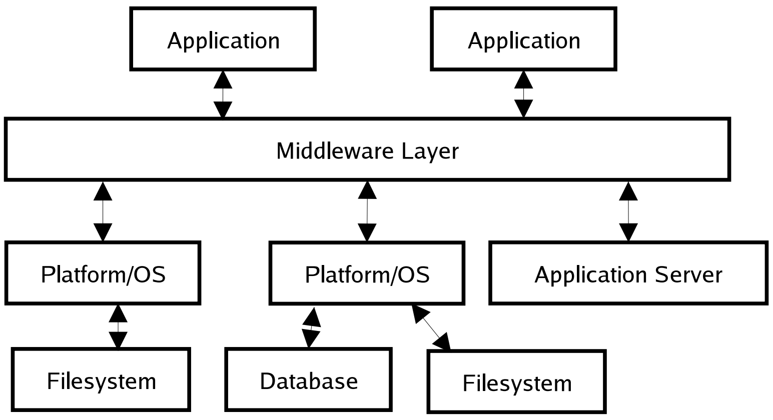

Figure 3.5: Middleware Layer

In the past, supporting of a standard programming language like C or Cobol was enough to solve this problems. The application was ported to the new operating system without much effort. Today's applications are much more complex and rely on graphics libraries, printing libraries, database access etc. which are not part of the language definition. To have the same programming freedom, standard programming interfaces have to be supported on the target platform. Another possibility is to provide standard protocols and the application programmers may implement their applications against services accessible through that standard protocols.

An indirection between the application and the system providing access to certain recourses can be the solution to the described problems. This indirection can be seen as another layer between the application and the system providing the necessary services. But this alone is not enough. To be able to call a middleware service, it must be available for more than one operating system, otherwise it is a platform service[Ber96].

Chapter 4

The Dinopolis Middleware System

The goal of the *Dinopolis System Architecture* is to summarize the technologies presented in the previous chapter, i.e. to make platform and operating system dependent services available transparently through distributed components, the so called *Dinopolis Objects*. An in-depth description of the *Dinopolis System Architecture* can be found in [Sch02b].

Nevertheless, to establish the context between *Dinopolis Objects* and the *Dinopolis System Architecture* a brief overview of the overall system is given in this chapter. At first we clarify what a *Dinopolis Object* is, followed by a short introduction of the several modules of the *Dinopolis System Architecture*.

4.1 The Dinopolis Object Definition

The terms *object* and *component* are very often used in the area of software development and got several meanings in different contexts. To have a consistent definition throughout this thesis the following applies to a *Dinopolis Object*.

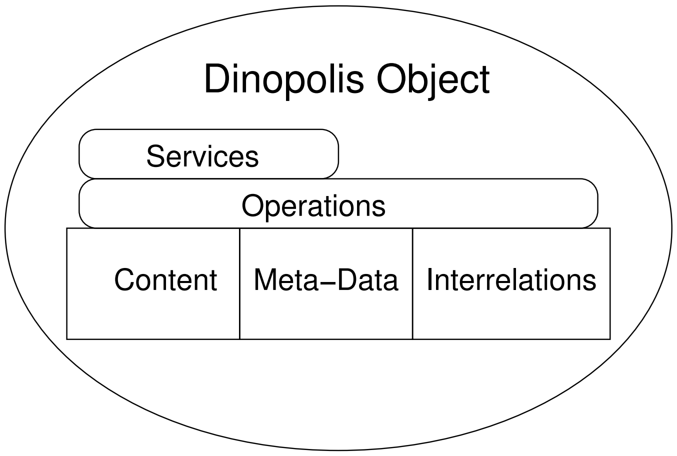

Figure 4.1: The *Dinopolis Object's Internal Structure*

- A *Dinopolis Object* is an addressable entity in the *Dinopolis Middleware System*.
- Addressing a *Dinopolis Object* is done via stable and globally unique handles (GUH). Stable means that it is guaranteed that a request of an *Dinopolis Object* with a given GUH can always be handled.
- A *Dinopolis Object* is a component in the sense of componentware rather than a low-level data-encapsulating entity.
- Hence, the fact that addressing and navigation are separated, navigation is done through *Interrelations*.
- A *Dinopolis Object* encapsulates content. Content can be any type of passive data like a document or streaming data. In a broader sense everything that can be considered persistent state information can be content of a *Dinopolis Object*.
- A *Dinopolis Object* provides a container for meta information like author, creation-date (descriptive information) or access-rights (adminis-

trative information). This meta-data container is tree-structured with hierarchical keys. The meta-data is stored in the form of values to these keys. The values can be of arbitrary type e.g they can be *Dinopolis Objects* themselves.

- Arbitrary many $n_1 : n_2 : ... : n_n$ *Interrelations* can be attached to a *Dinopolis Object*.
- What really makes a *Dinopolis Object* a component in the sense of componentware is the possibility to provide arbitrary functionality to the outside world (user space). Two kinds of functionality may be distinguished in the *Dinopolis Middleware System*: *Operations* and *Services.*
- One kind of functionality a *Dinopolis Object* provides are *Operations. Operations* are comparable to methods known from object-oriented programming languages. They can be invoked with a predefined set of parameters, carry out a task and return a result.
- The other kind of functionality is called *Service. Services* are comparable to operations with the difference that they provide a user-interface that an application can request. With this, application programmers can easily provide functionality of embedded systems without in-depth knowledge about the internal structure of the *Dinopolis Middleware System*.
- *Operations* and *Services* can be subject to change during runtime. This means that it is possible to add and remove *Operations* and *Services* during runtime.
- Access-control is much more sophisticated than in object-oriented programming languages and not limited to `public`, `protected` and `private`. When incorporating the whole security mechanism of the *Dinopolis Middleware System*, rules based on users, groups or roles are possible too.
- Every *Dinopolis Object* provides at least standard uniform access to content, meta-data, interrelations, operations and services.

An outcome of this enumeration is the so called *Internal Structure* of the *Dinopolis Object* consisting of content, meta-data, interrelations, operations and services (see figure 4.1)

4.2 The Dinopolis System Architecture

Due to the aspects of an overall slim system, the concept of loadable kernel modules was choosen. This results in an highly modularized kernel. Figure 4.2 shows the overall *Dinopolis System Architecture*. The responsibilities of the modules are as follows:

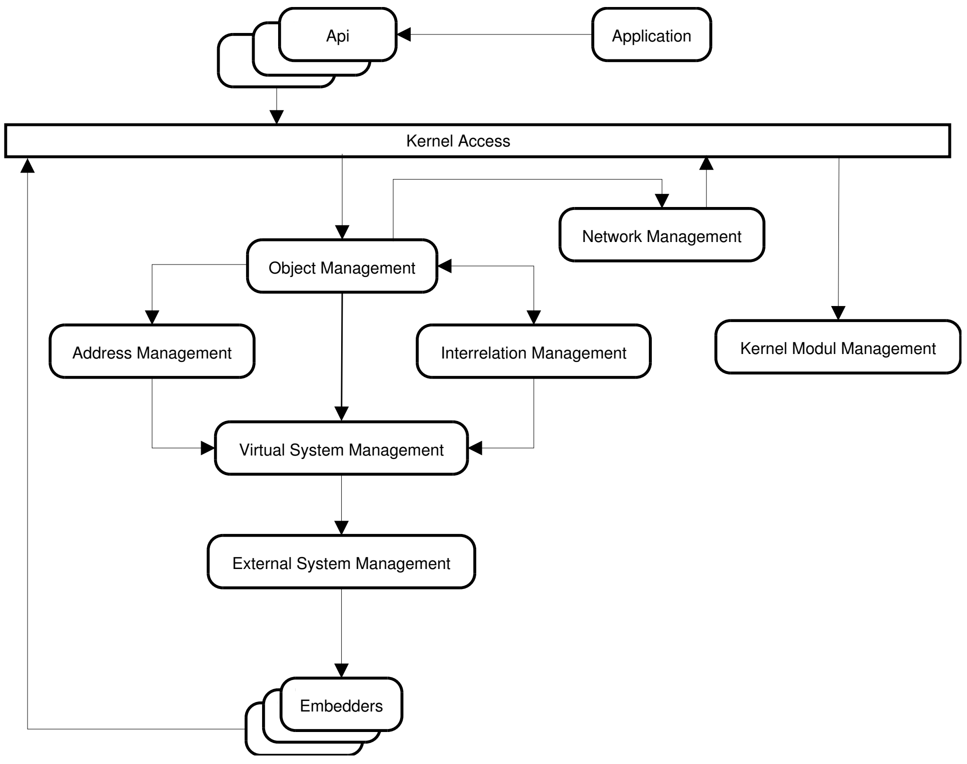

Figure 4.2: Overall *Dinopolis System Architecture*

Kernel Access Module:

To ensure that all calls that have there origin in the user space are security checked, they have to go through the *Kernel Access.* This cleanly separates the user space from the kernel space. This indirection can be achieved by a proxy mechanism where every object returned from the kernel access module is just a placeholder for the original object inside the kernel space (see figure 4.3)

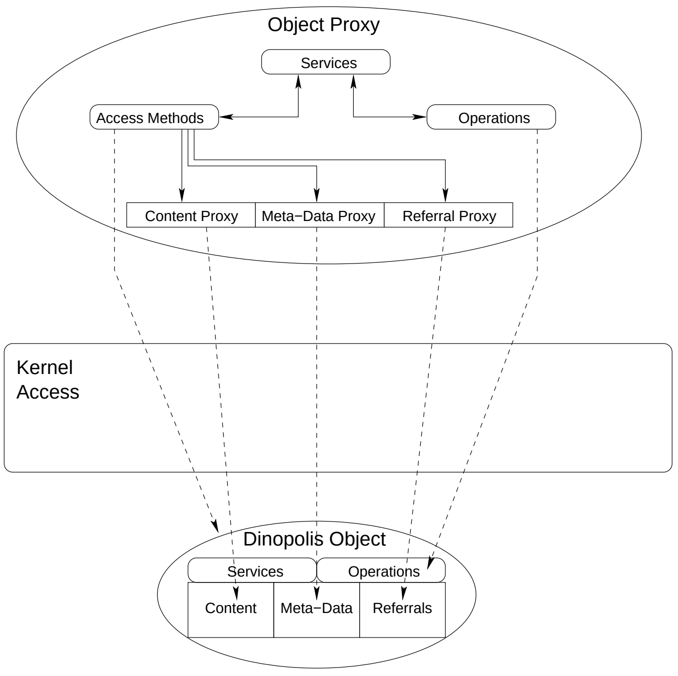

Figure 4.3: Dinopolis Object Proxy

This is the place where the *Security Management* module can grant or deny requests, respectively filter the results based on the security mechanisms implemented by the *Security Management* module.

Object Management Module:

The focus of this thesis lies on this module and all of its submodules. The *Object Management* module is responsible for all operations dealing with *Dinopolis Objects*. First of all it provides operations to control the life-cycle of *Dinopolis Objects* namely create, delete, load and store and operations dealing with the internal structure of a *Dinopolis Object*.

Let us for example consider an application for viewing electronic patient records built on top of the *Dinopolis Middleware System* requests the *Dinopolis Object* representing an electronic patient record from the local *Object Management* module. This is not done directly but through it (kernel access). The *Object Management* module carries out all tasks needed for building up the requested *Dinopolis Object*.

Address Management Module:

The *Address Management* module is responsible for everything concerning globally unique object handles or short GUHs. It is divided into two parts:

1. The *Local Address Management* module which carries out all tasks needed for local physical address handling.
2. The *Global Handle Management*, a universal lookup service which represents an implementation of the *DOLSA* algorithm (please have a look at [Sch02a] for details).

As stated before, the parts of an electronic patient record are most likely to be spread over several *Dinopolis* instances. Putting these Parts together to the whole electronic patient record requires a global unique identifier for each part of this *Dinopolis Object* which affects the user-space and the internal system and a local physical address for each part which is needed to define a local placement.

From the application programmers' point of view the request for an electronic patient record is the request for a *Dinopolis Objects* representing an electronic patient record passed on to the *Object Management* module with the GUH as a parameter which in turn uses the *Address Management* module to find out from which *Dinopolis* instance to request the *Dinopolis Object*. All further steps to make this work properly are carried out by the *Object Management* module and are discussed lateron in this thesis.

Interrelation Management Module:

The *Interrelation Management* module provides functionality needed by interrelations which are themselves *Dinopolis Objects*. The double arrow line between the *Interrelation Management* module and the *Object Management* module reflects the fact that the persistence of interrelations is handled by the *Object Management* module.

The main goal behind the introduction of interrelations is to separate navigation from addressing. The interrelation-concept is type-able and highly sophisticated. With this it is possible to navigate through the *Dinopolis Object Space.* This enables multi-directional linking from e.g a study about a certain topic to all referred electronic patient record or to further information about the author of this study.

Virtual System Management Module:

The Task of this module is managing arbitrarily many *Virtual Systems*. A *Virtual System* is a combination of embedded external systems. This abstraction can be used to construct powerful combinations. This combinations are then available as one *Virtual System*.

Besides other functionality, *Virtual Systems* are used to save and load the persistent data of *Dinopolis Objects*. With the described *Virtual System*, the meta-data of the *Dinopolis Object* could be stored in a database and the content in a file-system offering the advantage of full-text queries on the meta-data. As this is provided through the *Virtual System* concept, it is fully transparent to the *Dinopolis Object*.

External System Management Module:

The embedder implementations for external systems are registered with this module. The *External System Management* module is responsible for on demand instantiation and setup of concrete embedders for external systems.

Embedders:

External systems are accessed by *Embedders*. One can see *Embedders* as special drivers for external systems. An embedder for an external system provides at least the base functionality for an external system, e.g. read access for a file system. With the operation-, services-, hook- and observer mechanisms almost every functionality of an external system can be passed on to the *Dinopolis* system.

For example the functionality of an x-ray machine can be made available through a special embedder for this kind of low-level device. This can be seen as a plug-in mechanism for arbitrary types of devices.

Network Management Module:

The *Network Management* module is responsible for the network transparency of *Dinopolis Objects*, which means that remote *Dinopolis Objects* behave like local *Dinopolis Objects*.

The request for a *Dinopolis Object* always goes to the local *Object Management* module. The local *Object Management* module checks whether the requested *Dinopolis Object* is in its area of influence or not. If the request cannot be handled by the local *Object Management* module it is forwarded to the correct remote *Object Management* module.

Chapter 5

Use Cases and Requirements

It is beyond the scope of this thesis to cover all aspects of the *Dinopolis Middleware System* concerning the *Object Management* module and the *Dinopolis Objects* in full detail. In the following sections an overview of use-cases and the allocated requirements concerning either the *Dinopolis Objects* or the *Object Management* module is presented.

5.1 Use Cases

The following use cases have been outlined. Please note that the correct signatures of methods were not known at this early stage of the design.

Please note that in the presented use-cases strings are used to identify various things (e.g. *Static Types* etc.). In the final implementation *Qualifiers* are used instead of strings to specify the desired types.

- *Use Case 1:* Applications request an object by specifying its *Static Type* and perform a typecast on it in order to be able to work in a type-save manner:

```
Image my_image = (Image) OMM.createObject("Image");
int width = my_image.getWidth();
```

- *Use Case 2:* If an application wants to add, remove or override methods it typecasts the object to a special interface (here called *TypeAble*). By adding and removing *Definitions* the type and the behavior of the object can be changed at runtime.

  ```
  my_typed_object = (TypedAble)my_image;
  my_typed_object.attachDefinition("fully_qualified_class_name");
  ...
  my_typed_object.
    removeDefinition("another_fully_qualified_class_name");
  ```

  Please remember that *Declarations* are added implicitly. The *Definitions* are asked for all *Declarations* they implement. These *Declarations* are added to the list of *Declarations* (see also use case 3).

- *Use Case 3:* An object can be asked for a list of *Declarations*. For each *Declaration* a so-called *Dynamic Type System Cast* can be applied which returns an object that can be type-casted to the *Declaration*. The requestor can then work in a type-save manner with the casted object. Additionally the object can be upcasted (i.e. to its *Static Type*) and cross-casted (to another *Declaration* whenever desired.

  ```
  String[] decl_names = my_typed_object.getDeclarations();
  ...
  AnotherDeclaration decl = (AnotherDeclaration)my_typed_object.
    dynamicTypeSystemCast("AnotherDeclaration");

  int j = decl.getXXX();
  ...
  Image upcasted_object = (Image)decl.dynamicTypeSystemCast("Image");
  ```

- *Use Case 4:* The whole mechanism is observable. Applications as well as other parts of the system may subscribe as observers in order to get notifications if the dynamic type changes or if the binding (to *Definitions*) is changed.

  ```
  my_typed_object.
    attachDynamicTypeChangeObserver(BCObserver bc_observer);
  my_typed_object.
    detachDynamicTypeChangeObserver(BCObserver bc_observer);
  ```

```
my_typed_object.
  attachBindingChangeObserver(BCObserver bc_observer);
my_typed_object.
  detachBindingChangeObserver(BCObserver bc_observer);
```

- *Use Case 5:* There is a possibility to invoke methods on specific *Definitions* by specifying the "Scope" of a method or by using a "Super" operator:

```
Object return_value = my_typed_object.
  invokeCurrent(String method_name, Object[] params);
Object return_value = my_typed_object.
  invokeSuper(String method_name, Object[] params);
Object return_value = my_typed_object.
  invokeScoped(String method_name,Object[] params,
               String target_class_name);
```

5.2 General Requirements

- **Sub Modules have to be designed as components.**
  Dinopolis Objects can be seen as submodules of the *Object Management* module. Since it must be possible to provide arbitrary functionality through the *Dinopolis Objects* they have to be designed as components in the sense of componentware.

- ***Dinopolis Objects* have to be as slim a possible. Special functionality has to be requested explicitly.**
  Dinopolis Objects are heavily used in the *Dinopolis Middleware System*. As they can be used to encapsulate small pieces of data their base interface has be as slim as possible but can be extended for additional functionality as needed.

- **It must be possible to create *Dinopolis Objects* of different types.**
  Since *Dinopolis Objects* provide arbitrary functionality using different combinations of embedded systems, this has to be reflected in the type

of the *Dinopolis Object*. With this, application programmers are able to program in a type-safe manner.

- **A simple and intuitive way for configuring *Dinopolis Objects* at runtime has to be found. Arbitrary object types must not be hard-coded.**
  It has to be possible to configure *Dinopolis Objects* at runtime by just defining how different content is combined, where the meta-data comes from and which parts are responsible for the *Dinopolis Objects*' functionality. No programming must be required for creating new types of *Dinopolis Objects*.

- **A *Dinopolis Object* may be asked for its type.**
  This type must be a unique key for the functionality provided by a certain kind of *Dinopolis Object*. A new *Dinopolis Object* can be requested via this unique type too.

- ***Dinopolis Objects* must be able to represent active objects as well.**
  Objects that can change their inner-state themselves are called *Active Objects* (e.g. Reminder, Calendar, Scheduler, Agent, Chat or Whiteboard applications). A notification mechanism is needed to enable such active objects to trigger events and inform other objects.

- ***Dinopolis Objects* must be observable.**
  It has to possible that interested parts of the *Dinopolis Middleware System* can be notified of certain events (e.g call of an operation on a *Dinopolis Object* or change of the functionality provided through the *Dinopolis Object*. For this a notification mechanism has to be provided. (see the Observer/Observable design pattern described in [GHJV95]).

- **It must be possible to lock a *Dinopolis Object*.**
  In special situations it can happen that a *Dinopolis Object* needs to be locked. This is e.g. the case for transactions or critical concurrent access.

- **The following basic requests concerning *Dinopolis Objects* must be handled by the Object Management Module:**
  - **Create a *Dinopolis Object*.**
    Given an unique type identifier create a *Dinopolis Object* of that certain type.
  - **Delete a *Dinopolis Object*.**
    Given an globally unique handle, delete the corresponding *Dinopolis Object*.
  - **Load a *Dinopolis Object*.**
    Given an globally unique handle, load that *Dinopolis Object* into the memory.
  - **Save a *Dinopolis Object*.**
    Save the all information needed to be able to load that *Dinopolis Object* into memory.
  - **Move a *Dinopolis Object* from one *Virtual System* to another.**
    Move every stored data of the identified *Dinopolis Object* to another *Virtual System*.

5.3 Internal Structure

Dinopolis Objects encapsulate content, meta-data and interrelations and provide operations and services. The operations and services must not be implemented in the *Dinopolis Objects* themselves as they can come from other modules.

5.3.1 Operations

Operations are fully comparable to methods known from object-oriented languages. They wrap functionality that can be performed on *Dinopolis Objects*. Two kinds of operations can be distinguished:

1. **Standard operations**
   Standard operations provide basic *Dinopolis* functionality and must be provided by all *Dinopolis Objects*. They define a minimum interface common to all *Dinopolis Objects*.

2. **Extended operations**
   Extended operations differ from *Dinopolis Object* to *Dinopolis Object*. Those operations are determined by the type of the *Dinopolis Object* and heavily depend on the *Virtual System* where a *Dinopolis Object* resides at the moment.

- The following Standard Operations must be provided by *Dinopolis Objects*:

  - **Content related operations**
    Every *Dinopolis Object* must provide basic operations to work with the encapsulated content e.g. `getContent()`, `saveContent()`, etc.
  - **Operations concerning interrelations**
    Operations for working with the attached interrelations like `get-Endpoints()`.
  - **Operations dealing with meta-data**
    The meta-data of a *Dinopolis Object* has to be accessible.
  - **Operations concerning the extended operations**
    It must be possible to ask a *Dinopolis Object* for its extended operations.
  - **Operations dealing with services**
    It must be possible to ask a *Dinopolis Object* for its services.
  - **Operations which provide for working with the *Dynamic Type System*.**
    E.g. requesting the *Type* of the object or type-casting the object to different types.

- **Extended Operations: Pass through the full functionality of *Virtual Systems*.**

It has to be possible to pass through the full functionality provided by the *Virtual Systems*.

- **Operations have to be implemented via Hooks.**
  Operations must not be implemented hard-coded in *Dinopolis Objects* themselves, but can be requested from other modules in the *Dinopolis Middleware System*. To achieve this, operations have to be implemented via the *Hook* design pattern (see [GHJV95]).

5.3.2 Services

As stated before, services provide a user-interface. This user-interface can be requested by the application and used as is in the application. With this concept developers can remain on their level of abstraction and don't need to know to much about the internals of the system like knowing how to use the correct operations with the correct parameters.

Another advantage of this concept is the reuseability of these services, moving from the application to the *Dinopolis Middleware System* where developers of *Virtual Systems* or *Embedders* can provide several services to all developers using *Dinopolis Objects*.

- **It must be possible to ask a *Dinopolis Object* for its services.**
  Since not all services may be available at every time it must be possible to ask a *Dinopolis Object* which services are currently available. A list of available services in a standardized format will be returned.
- **Services must only use operations for accessing or modifying *Dinopolis Objects* for security reasons.**
  Services are provided to the user-space. Thus to rely on the *Dinopolis* security mechanism services must only use operations since they are security checked.
- **Services must be explicitly requested depending on the *Dinopolis Objects*' type.**

Services are used to manipulate data encapsulated in the corresponding *Dinopolis Object*. This implies explicitly requesting the appropriate service from the *Dinopolis Object*.

- **Services are bound to *Dinopolis Object* types and must not be bound to instances.**
  Since services work with *Dinopolis Objects* of appropriate type using their operations, services are bound to the type of a *Dinopolis Object* rather than to an instance of that type.

- **It must be possible to provide services for different GUI toolkits.**
  Since the *Dinopolis Middleware System* is highly modular and configurable this must be true for parts of it too. Services are just relying on the operations they use, which allows implementing services for different GUI toolkits.

5.3.3 Content

Following the *Dinopolis Object* definition (section 4.1), *content* can be any type of passive data like a document or streaming data. In a broader sense, everything that can be considered persistent state information can be content of a *Dinopolis Object*.

Since it can not be foreseen which concrete types of content will be used in *Dinopolis Objects* we must treat this as a blackbox. Nonetheless, some common requirements can be stated and are as follows:

- **Access to the content is always done through a Content Data Handler.**
  This approach allows to define a minimal set of methods common to every specialized content data handler. New types of content can easily be added by providing an appropriate content data handler for that type.

- **Provide Data Handlers for different Content Types.**
  For every type of content in the *Dinopolis Middleware System* there has to be an appropriate content data handler.
  - **Data Handlers must conform to a general interface.**
    Every content data handler is either derived from a base data handler or implements a base data handler interface.
  - **It must be possible to add new Data Handlers at runtime.**
    It must be possible to register new content data handlers at runtime with the *Dinopolis Middleware System*.
  - **It must be possible to request the same content in different formats (if available).**
    For example,a text document written in LaTeX may be requested as is, plain-text, dvi or pdf (if available).
  - **Request a list of supported formats.**
    It has to possible to request a list of supported formats.
- **Provide different access methods to the same content type.**
  Consider a movie. The data representing that movie has to be accessible as a whole or as a video stream.
  - **Request a list of possible access methods.**
    It has to be possible to request a list of all possible access methods to the content.
  - **Ask an object for its native access method.**
    For several reasons, e.g performance, it is a good idea to access the content through its native access method. Every content data type has a native access method.

5.3.4 Interrelations

As already stated before, navigation through the *Dinopolis Objects* is done with interrelations. Arbitrary many $n_1 : n_2 : ... : n_n$ interrelations can be attached to a *Dinopolis Object*. Interrelations are managed by the *Interrelation Management* module and are therefore beyond the scope of this document. Nevertheless it has to be possible to ask a *Dinopolis Object* for its Interrelations.

- **A *Dinopolis Object* may be asked for an interrelation handler which is able to manage interrelations.**
  The *Interrelation Management* module provides functionality to work with interrelations. This functionality is provided through an interrelation handler which can be requested from a *Dinopolis Object*.

5.3.5 Meta Data

Dinopolis Objects provide a container for meta information. As meta information can be of arbitrary nature, the container must support structural functionality.

- **It has to be possible to assign arbitrary meta-data information to any *Dinopolis Object*.**
  The nature of meta data can be reflected in its type. The meta data container can hold meta data of arbitrary type. It is organized in a tree structure.
  - **Keys are hierarchically organized.**
    The keys can be hierarchically organized for structural reasons. As an example consider a meta-data key "document.author.name" and another called "document.author.address".
  - **Values may be of arbitrary type.**
    The values can be objects of arbitrary type reflecting their nature.
- **There has to be a standardized way to request Meta Data information of *Dinopolis Objects*.**
  As meta-data is an integral part of *Dinopolis Objects*, operations deal-

ing with meta-data of *Dinopolis Objects* are in the standard interface of *Dinopolis Objects*

- **Meta Data might have different origins which must be handled accordingly.**
  Several chunks of meta-data for *Dinopolis Objects* might have different origins. They must be handled accordingly depending on their origin. To illustrate this fact consider the following examples:

  - Coming from the physical system the *Dinopolis Object* is stored in (e.g. file system: modification date, owner, etc.).
  - Stored in the content of the *Dinopolis Object*. (e.g. HTML file's header information: keywords, authors, etc.).
  - Stored in another physical system.

Chapter 6

The Object Management Module

According to the module architecture of the *Dinopolis Middleware System* the *Object Management* module is responsible for *Dinopolis Objects*. The relation between the *Object Management* module and the other modules of *Dinopolis* are presented in section 6.1. *Dinopolis Objects* heavily rely on the *Object Management* module and vice versa. Every request concerning *Dinopolis Objects* is handled by the *Object Management* module. This includes all operations dealing with the life-cycle of *Dinopolis Objects* like creating and deleting, loading, storing and rearranging *Dinopolis Objects*. As an outcome of this strong relationship between the *Dinopolis Objects* and the *Object Management* module, they were designed together by one team of the *Dinopolis* project.

The *Object Management* module consists of several sub-modules with different responsibilities. We can distinguish between sub-modules dealing with whole *Dinopolis Objects* which are presented in section 6.2 of this chapter, sub-modules concerning the *Dinopolis Object* itself (see chapter 7 for details), and sub-modules of the *Dynamic Type* mechanism (see chapter 8 for details).

It would be beyond the scope of this thesis to cover all aspects of the *Object Management* module and the *Dinopolis Objects* in full detail. The main focus of this thesis lies on the *Dynamic Type* mechanism and its internals. Nevertheless, to understand the *Dynamic Type* mechanism it is necessary to present the other sub-modules as well.

6.1 Environment

In this section the relations between the *Object Management* module and the modules in the *Dinopolis Middleware System* are presented.

6.1.1 Module Management Module

All modules in the *Dinopolis Middleware System* are registered with the *Module Management* module. The modules provide their functionality through *Functional* modules which can also be requested from the *Module Management* module. This concept makes it very easy to change the module configuration at runtime. Modules may be registered or unregistered during runtime. Another advantage of this modularized architecture is that the core of the *Dinopolis Middleware System* is very slim.

6.1.2 Kernel Access Module

All modules of the *Dinopolis Middleware System* reside in the so called kernel space of *Dinopolis*. If application programmer want to gain access to a module which resides in the kernel space they have to request it through the *Kernel Access* module. This ensures that all calls from the outside can be intercepted by the implemented security mechanism which itself is a module of the *Dinopolis Middleware System*.

Since *Dinopolis Objects* are requested from the *Functional* module of the *Object Management* module this is true for calling operations or services on

Dinopolis Objects too.

6.1.3 Address Management Module

The *Object Management* module is entry point for everything concerning *Dinopolis Objects*. Requesting a *Dinopolis Object* with the GUH as a Parameter is done through the *Object Management* module too. The *Object Management* module contacts the *Address Management* module to resolve the current location of the *Dinopolis Object* identified by the given GUH.

Either the *Dinopolis Object* resides in the local *Dinopolis* instance or in a remote *Dinopolis* instance. If it turns out that the *Dinopolis Object* resides in a remote *Dinopolis* instance, the request will be delegated through the *Network Management* module to the remote *Object Management* module.

In both cases the next step is to retrieve the location of the persistent data of the *Dinopolis Object* from the *Address Management* module. The persistent data is stored in *Virtual Systems* under a local placement. The *Address Management* module answers the request with a *Virtual System* identifier and a local placement. This local placement is interpreted by the according *Virtual System* to get the stored data.

Retrieving the location of the *Dinopolis* instance, the *Virtual System* identifier and the local placement may be done in one step for performance reasons.

While creating a new *Dinopolis Object* the *Address Management* module is asked for a GUH with a *Virtual System* identifier for the newly created *Dinopolis Object*.

A *Dinopolis Object* can be asked for its GUH too. It can answer this request itself or by delegating it to the *Address Management* module. For this to work the *Address Management* module must be capable of a reverse lookup mechanism.

It must be possible that the persistent data of a *Dinopolis Object* is moved from one *Virtual System* to another. After performing the move the *Address Management* module must be told the new location of the persistent data.

If a *Dinopolis Object* and its persistent data are removed from the *Dinopolis* instance the *Address Management* module must be told to mark the GUH invalid. This GUH is not allowed to be used any more.

6.1.4 Virtual System Management Module

The persistent data of *Dinopolis Objects* is saved in so called *Virtual Systems*. The functionality of the *Virtual Systems* is attached to the *Dinopolis Objects* with the *Dynamic Type* mechanism (see chapter 8). This special *Dynamic Type* which deals with the persistence of a *Dinopolis Object* is called the *Dynamic Storage Type*.

The *Dynamic Storage Type* is a *Definition* which implements one ore more *Declarations*. *Declarations* are used to declare the interface of *Dinopolis Objects* and *Definitions* provide the implementation which actually carry out the tasks. While attaching a *Definition* to a *Dinopolis Object* all *Declarations* implemented by this *Definition* are added to the interface of the *Dinopolis Object*.

The *Dynamic Storage Type* heavily depends on the *Virtual System* and thus the *Definition* is provided from the *Virtual System* where the data is stored.

Actually definitions are requested from a dynamic definition factory. The full calling sequence to get a *Definition* is to ask the origin of the *Definition* for the *Definition*. The origin will ask the *Object Management* module and finally the *Object Management* module will ask the dynamic definition factory.

Since every *Dinopolis Object* stores its persistent data in a *Virtual System*, the *Dynamic Storage Type* is attached while creating the *Dinopolis*

Object. Therefore a *Virtual System* which can deal with the persistent data of the given *Dinopolis Object* must be found and the appropriate *Dynamic Storage Type* must be requested from this *Virtual System*.

Additionally the *Virtual System* can be asked to add system immanent stuff e.g. system immanent interrelations to the *Dinopolis Object*.

To find a proper *Virtual System* for a certain type of persistent data, the *Virtual System Management* module can be asked for a *Virtual System* which can hold the data. This is needed when e.g creating or moving a *Dinopolis Object*.

It is possible to create a *Dinopolis Object* which represents already existing data stored in a *Virtual System*. In this case the *Virtual System* is asked for the Type-ID of the stored data in order to be able to instantiate the correct *Static Type* for the *Dinopolis Object* to be created.

6.1.5 Interrelation Management Module

For several reasons navigation and addressing are strictly separated in the *Dinopolis Middleware System*. Navigation and logic relationship between *Dinopolis Objects* is modelled with *Interrelations*. *Interrelations* are strongly typed and arbitrary types (e.g. hyperlinks, bookmarks) are available. *Dinopolis Objects* must provide access to their attached interrelations.

An iterator over the list of attached *Interrelations* can be requested. *Interrelations* can be attached to or detached from *Dinopolis Objects*. Further functionality of *Interrelations* is provided by the *Interrelation Management* module. Figure 6.1 shows an *Interrelation* between several *Dinopolis Objects*.

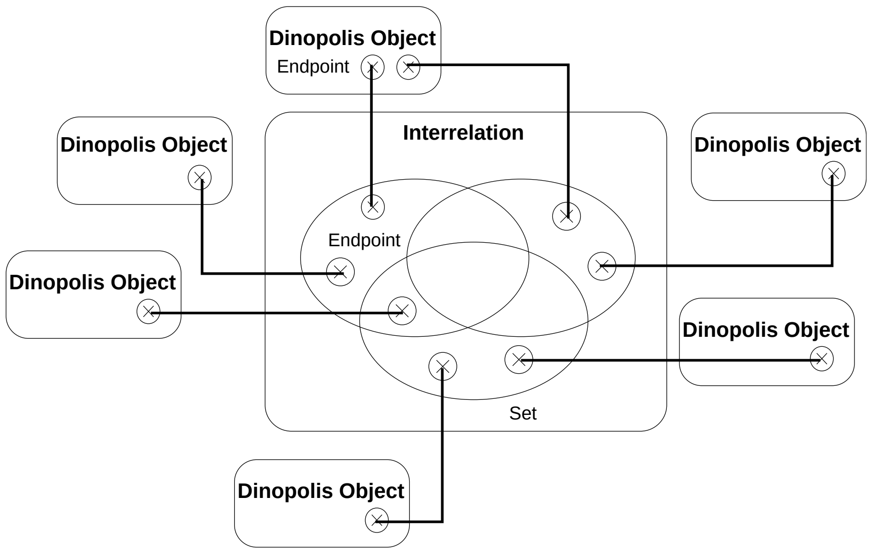

Figure 6.1: A Interrelation

6.1.6 Kernel Modules

As already stated, functionality from modules can be attached to *Dinopolis Objects* with the *Dynamic Type* mechanism. This mechanism requires, that the functionality is provided as *Definitions* implementing one or more *Declarations*. Every kernel module can provide definitions for *Dinopolis Objects*. Consider kernel modules which provide functionality for e.g versioning, transactions or locking. If for instance versioning is required for a *Dinopolis Object*, the *Object Management* module requests a definition for versioning from the *Versioning* module and attaches it to the appropriate *Dinopolis Object*.

6.1.7 Network Management Module

As *Dinopolis Objects* must be transparently accessible from the local *Dinopolis Instance*, requests for *Dinopolis Objects* which are not in the area of influence of the local *Object Management* module must be delegated to the remote *Object Management* module.

To achieve this, the local *Object Management* module retrieves the current location of the requested *Dinopolis Object* from the *Address Management* module and delegates the request to the remote *Object Management* module via the *Network Management* module.

The remote *Object Management* module provides a placeholder object, a so-called remote proxy, for further requests to this *Dinopolis Object*.

6.1.8 Configuration Management Module

The *Configuration Management* module provides information needed by the *Object Management* module. The classpath for the factories of the *Object Management* module or which types of *Dinopolis Objects* are a available in this *Dinopolis Instance* can be requested from the *Configuration Management* module. Since reconfiguration can happen during runtime it can be forced to reread the configuration.

6.2 Sub-Modules concerning Object Management

The *Object Management* module can be decomposed into several parts, each responsible for a certain task. The following parts and their responsibilities can be identified:

Object Manager: The *Object Manager* provides operations like *createObject()* or *loadObject()* dealing with whole *Dinopolis Objects*. These operations are provided by a functional module registered with the *Module Management* module.

Object Management Bootstrapper: The *Module Management* module requires functionality for setting up and tearing down the whole module. This functionality is provided from the *Object Management Bootstrapper*.

Static Type to Type-ID Mapping: Given a Type-ID, this module returns a list of *Static Types* which can deal with the referring kind of data (this information is similar to MIME-Types).

Dinopolis Instance Object-Table: These "internal tables" of the *Object Management* module are used for needed status information of *Dinopolis Objects* (e.g. references counting, object loaded to memory, etc.)

Dinopolis Observer Manager: Provides functionality for observing several aspects in the *Object Management* module like creation of a certain *Static Type*, etc. This functionality can be accessed through a functional module by interested parties.

Chapter 7

The Dinopolis Object

The *Dinopolis Object* definition is already given in section 4.1. In the first part of this chapter the sub-modules of the *Dinopolis Object* are introduced. This is followed by the presentation of the life-cycle of *Dinopolis Object*. Last, the processes and algorithms concerning *Dinopolis Objects* are presented.

7.1 Sub-Modules of the *Dinopolis Object*

Data Handler Factory: The different kinds of data encapsulated in *Dinopolis Objects* are accessed through appropriate *Data Handlers*. The *Data Handler Factory* provides instances of certain *Data Handlers* for the various kinds of data types.

Content Handler: Is a data handler for content data types. A *Content Handler* provides functionality like *storeContent()* or *getListOfSupportedFormats()*.

Meta Data Handler Factory: Provides instances of certain *Meta Data Handlers* for the various kinds of meta-data sets.

Meta Data Handler: Provides functionality like *storeMetaData()* or *get-*

ListOfKeys(). Meta-data is stored in key-value pairs, where the values are again accessed through *Data Handlers* according to their data-type.

Interrelation Handler: The *Interrelation Handler* actually is part of the *Interrelation Management* module and is listed here just for completeness. It is used to handle *Interrelations* attached to the *Dinopolis Object*.

Operation Handler: The *Operation Handler* allows access to all operations available through a *Dinopolis Object*.

Dinopolis Object Operation Hooks: An operation encapsulates a method call which is invokable on a *Dinopolis Object*. All operations are implemented as *Hooks* (see [Sch02b] for details)

Service: A *Service* is a graphical user-interface built on top of the operations. This can be used by applications.

Service Factory: Services are provided for different graphical toolkits. The *Service Factory* provides concrete instances for the different graphical user-interfaces.

Service Handler: The different services provided for a *Dinopolis Object* are requested through the *Service Handler* which can be requested from the *Dinopolis Object*.

Dinopolis Object Observer: Provides functionality for observing certain aspects of *Dinopolis Objects*.

7.2 The Life Cycle of a Dinopolis Object

Because of the fact that we need to make the encapsulated data of *Dinopolis Objects* persistent, we have to deal with so called long living objects.

This brings us to the extended life-cycle of a *Dinopolis Object* compared to conventional objects everybody is used to from object oriented programming languages.

The whole life-cycle of a *Dinopolis Object* starts with its creation followed by construction, destruction and deletion. For an in-detail description of all processes please have a look at section 7.3.

Creation: The operation of creating a *Dinopolis Object* is done by the *Creator*. The *Creator* is only called once in the whole life-cycle of a *Dinopolis Object*. First of all a *Dinopolis Object* of the desired type is requested from the *Object Management* module. Since a *Dinopolis Object* is an addressable unit in the *Dinopolis Middleware System*, a new unique GUH must be requested from the *Address Management* module during creation. Furthermore a *Virtual System* which can hold the persistent data of the *Dinopolis Object* has to be determined. Several other initial steps can be triggered which will be discussed in section 7.3.

Construction: This operation is carried out by the *Constructor*. Every time a *Dinopolis Object* is loaded into memory, the *Constructor* is called to load all persistent data from the *Virtual System* to the *Dinopolis Object* so it is in the nature of the *Constructor* to be called arbitrarily often during the life-time of a *Dinopolis Object*.

Destruction: When no more reference points to a certain *Dinopolis Object* in the memory, the *Destructor* is called. It is in the responsibility of the *Destructor* to store all non-transient data to the corresponding persistent location. Several other necessary clean-up operations can be carried out by the *Destructor*. Like the *Constructor* the *Destructor* can be called arbitrarily often during the life-cycle of a *Dinopolis Object*.

Deletion: At the end of a *Dinopolis Object*'s life-cycle the *Deletor* is called. This just can happen once during the life-cycle of a *Dinopolis Object*. Here another mechanism comes into play: the *Deletor Strategy*. Since *Dinopolis Objects* can be referenced more than once, the deletion of a

Dinopolis Object can be delayed, or if the requestor is not allowed to trigger deletion, even denied.

7.3 Processes and Algorithms

7.3.1 The Creation of a Dinopolis Object

1. The creation of a *Dinopolis Object* is requested via the *Object Manager*. For the creation of a *Dinopolis Object* its *Static Type* and the address referring to the location for its persistent data is needed. The correct *Static Type* is either provided from the requesting application (e.g. the application has knowledge about *Static Types*) or the application may request a list of Type-IDs from the specific system referred to by the location descriptor. The complete address information consists of the *Dinopolis Instance* identifier, the *Virtual System* identifier and the *Location Descriptor* determining the exact location inside the *Virtual System*. The *Dinopolis Instance* identifier can be requested from the *Address Management* module. The *Virtual System* identifier can be requested from *Virtual System Management* module which allows for searching a specific *Virtual System*. The client may have in-depth knowledge about the internal structure of the desired *Virtual System* and provide the *Location Descriptor* directly. Another possibility is to browse the *Virtual Systems* and request the *Location Descriptor* at the desired location.

   If the *Dinopolis Instance* identifier or the *Virtual System* identifier or *Location Descriptor* are not given, the system chooses a default location.

2. The *Object Manager* asks the *Address Management* module for the location of the *Dinopolis Instance* specified by the *Dinopolis Instance* identifier.

3. If it turns out that the specified *Dinopolis Instance* is not the local *Dinopolis Instance*, the creation of the *Dinopolis Object* is delegated

to the remote location by forwarding the request to the remote *Object Manager* through the *Network Management* module.

4. The *Object Manager* requests an instance of the *Static Type* from the *Dinopolis Object Factory.*

5. The *Object Manager* holds a list of *Definitions* for every *Static Type* it supports and first attaches all *Definitions* for the life-cycle management of *Dinopolis Object*. The *Creator* and *Deletor* are requested from the desired *Virtual System.* The *Constructor* and *Destructor* come from *Object Manager* itself.

6. The *Creator* of the dynamic storage part (see section 8.6) is invoked with the given parameters which allocates the required space in the desired *Virtual System.*

7. Next the *Constructor* is called which in turn initiates that all further *Definitions* in the list of *Definitions* are attached to the *Dinopolis Object*.

8. The internal structure is set with the data provided by the requesting client from the *Object Manager*.

9. A GUH for the newly created *Dinopolis Object* is requested from the *Address Management* module.

10. The *Dinopolis Instance Object Table* is updated. The reference counter is incremented and the *Dinopolis Object* is marked as "loaded to memory".

11. In the case of remote object creation the *Creator* is returned to the requesting *Object Manager* which in turn builds a remote proxy object at the remote location and updates its *Dinopolis Instance Object Table.*

12. The *Dinopolis Object* or The remote proxy objct is returned to the requesting client.

If any step fails, an exceptional state is indicated.

7.3.2 Storing the Persistent Data of a Dinopolis Object

1. The *store()* operation which is part of the standard interface of *Dinopolis Objects* is called. The *store()* operation is defined by a *Definition*, which must be provided by the desired *Virtual System* and is part of the *Dynamic Storage Part* of the *Dinopolis Object*.
2. The persistent data of the *Dinopolis Object* gets stored as follows:
   (a) The *storeContent()* method of the appropriate *Content Data Handler*, and
   (b) The *storeMetaData()* of the corresponding *Meta-Data Handler* are called.
   (c) Additionally the *Virtual System* may provide storage of further state information of the *Dinopolis Object*. If provided, this is accessed through *storeStaticTypeMembers()* and *storeDynamicTypeMembers()*.

If any step fails, an exceptional state is indicated.

7.3.3 Removing an Object from Memory

The removing of a *Dinopolis Object* from memory is triggered by the *Dinopolis Instance Object Table.* This is done if no more references point to the loaded *Dinopolis Object*. The *Dinopolis Instance Object Table* must provide functionality for so called “distributed” reference counting, in order to be able to decide if no more references point to the *Dinopolis Object* under examination. Only the “local” *Object Manager* of the corresponding *Dinopolis Object* can carry out its removing from memory. The following steps are undertaken:

1. The *Destructor* of the *Dynamic Storage Part* is called. The destructor may trigger storing the persistent data of the *Dinopolis Object* if

necessary. This is can be indicated by a so called “dirty flag” in the *Dinopolis Instance Object Table*.

2. The *Dinopolis Object* is marked as “removed from memory” in the *Dinopolis Instance Object Table* and the entry can be deleted.
3. The *Object Manager* can trigger further clean-up operations.

If any step fails, an exceptional state is indicated.

7.3.4 Loading a Dinopolis Object to Memory

The loading of a *Dinopolis Object* into memory is always requested with its GUH as a parameter. The following steps have to be carried out by the *Object Management* module:

1. Loading is requested from the *Object Manager* with a given GUH.
2. The *Object Manager* determines if the requested *Dinopolis Object* is already loaded into memory. Therefore the *Dinopolis Instance Object Table* is asked if the requested *Dinopolis Object* aleready resides in memory. If yes, the instance is returned and the process is finished.
3. If no instance can be returned from the local instance table, the location of the corresponding *Dinopolis Instance* for the given GUH is requested from the *Address Management* module. If this is a remote location, the request is delegated to the remote *Object Manager* via the *Network Management* module.
4. The (remote) *Object Manager* asks the *Address Management* module to resolve the *Virtual System* ID and *Virtual System*-Object ID. With this information the *Virtual System Management* module is asked for the desired *Virtual System*.
5. The *Virtual System* is asked for the Type-ID of the persistent data of the *Dinopolis Object* with the given *Virtual System*-Object ID

6. With this Type-ID the *Object Manager* requests information about the corresponding *Static Type* from the *Static Type* to Type-ID mapping.
7. An instance of this *Static Type* is requested from the *DinopolisObject Factory*
8. The *Object Manager* holds a list of *Definitions* for every *Static Type* it supports, and first attaches all *Definitions* for the life-cycle management of the *Dinopolis Object*. The *Creator* and *Deletor* are requested from the desired *Virtual System*. The *Constructor* and *Destructor* come from the *Object Manager* itself.
9. The *Constructor* of the *Dynamic Storage Part* is called. All remaining *Definitions* are attached to the *Dinopolis Object*. All interrelations are attached accordingly. If the *Virtual System* provides additional storage capabilities, the stored members are loaded to the *Dinopolis Object* (*loadStaticTypeMembers()*, *loadDynamicTypeMembers()*.
10. The *Object Manager* tells the *Dinopolis Instance Object table* to mark the *Dinopolis Object* as "loaded to memory" and to increase the corresponding reference counter.
11. The *Object Manager* returns the object or the remote proxy to the requestor.

If any step fails, an exceptional state is indicated.

7.3.5 Deleting a Dinopolis Object

For several reasons deleting a *Dinopolis Object* is designed as a two-step operation. First the so called *Deletor Strategy* is called, and if the deletion is not refused, the *Deletor* of the *Dinopolis Object* is called which actually deletes the *Dinopolis Object* from the *Dinopolis Instance*.

1. The *Object Manager* triggers the execution of the *Deletor Strategy.*

2. The *Deletor Strategy* can request information about the current state of the *Dinopolis Object* from the *Dinopolis Object Instance Table* (e.g. number of references pointing to *Dinopolis Object*) and decide how to proceed. Different possibilities exist:

   (a) The *Deletor Strategy* can refuse the deletion of the *Dinopolis Object* completely and raises an exception.

   (b) The *Deletor Strategy* delays the deletion of the *Dinopolis Object* until no more references to it exist. Then it calls the *Deletor* of the *Dinopolis Object* which knows how to delete the persistent data.

   (c) The *Deletor Strategy* carries out the object deletion immediately and puts an appropriate "no longer existing" object in its place. (therefore it replaces all *Definitions* with *Definitions* that throw an *ObjectDeletedException* at every method invocation). Then it calls the *Deletor* of the *Dinopolis Object*, which knows how to delete the persistent data.

For system consistency reasons it is guaranteed in either case that no un-resolvable reference to an object is left.

7.3.6 Moving the Persistent Data of a Dinopolis Object

For several reasons it has to be possible to move the persistent data to another location (e.g. the corresponding *Virtual System* must be shut down for maintenance reasons). The following steps have to carried out:

1. The movement is requested from the *Object Manager* by passing the GUH for the source *Dinopolis Object* and the new location address (*Dinopolis Instance* identifier, *Virtual System* identifier and *Location Descriptor*).

2. The *Object Manager* asks the *Address Management* module for the location of the *Dinopolis Instance* specified by the *Dinopolis Instance*

identifier.

3. If it turns out that the specified *Dinopolis Instance* is not the local *Dinopolis Instance*, the moving of the *Dinopolis Object* is delegated to the remote location by forwarding the request to the remote *Object Manager* through the *Network Management* module.
4. The remote system's *Object Manager* requests the source *Dinopolis Object*. With this remote object the rest of the move operation is carried out exactly like a local move.
5. The desired *Virtual System* is asked if it can hold the data of the *Static Type* of the source *Dinopolis Object*. If yes the *Object Manager* starts to build a temporary object.
6. The *Object Manager* requests an instance of the *Static Type* from the *Dinopolis Object Factory.*
7. The *Object Manager* holds a list of *Definitions* for every *Static Type* it supports, and first attaches all *Definitions* for the life-cycle management of *Dinopolis Object*. The *Creator* and *Deletor* are requested from the desired *Virtual System*. The *Constructor* and *Destructor* come from *Object Manager* itself.
8. The *Creator* of the dynamic storage part (see section 8.6) is invoked with the given parameters which allocates the required space in the desired *Virtual System*.
9. Next the *Constructor* is called, which initiates that all further *Definitions* in the list of *Definitions* are attached to the *Dinopolis Object*.
10. The internal structure is initialized with the data provided by the source *Dinopolis Object* from the *Object Manager*.
11. The Object Manager informs the *Interrelation Management* module that the object has been moved to a new location.
12. If the *Dinopolis Object* is moved across *Dinopolis Instance* boundaries the source *Object Manager* replaces all *Definitions* through *Network Definitions*.

13. The *Dinopolis Instance Object Tables* of both *Dinopolis Instance* are updated.

14. The *Object Manager* informs the *Address Management* module which has to carry out all necessary steps to reflect the new location of the persistent data.

15. After all necessary internal tasks of the *Address Management* module have been performed successfully, the source *Object Manager* is informed. It initiates all necessary clean-up operations.

Chapter 8

The Dynamic Types Mechanism

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 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.

Adding and removing functionality to/from *Dinopolis Objects* cause a change of their public interface. Since this fact cannot be reflected with strong static typed object-oriented programming languages another solution to this problem has to be found. Nevertheless all provided operations are inspired by traditional object-oriented programming languages and their well known mechanisms (e.g. adding functionality to a *Dinopolis Object* uses the concept of inheritance).

To rely on *Dinopolis Objects* two additional requirements come into play. The first requirement is, that *Dinopolis Objects* must provide standardized access to their internal structure and base functionality as *Dinopolis Objects*. The second requirement is, that it must be possible to provide guaranteed access to their base functionality depending on their type.

Summarizing the following must be taken under considerations:

1. Adding/removing functionality to/from *Dinopolis Objects*
2. Method dispatching at runtime
3. Standardized access to the base functionality of *Dinopolis Objects*
4. Guaranteed base functionality of a certain *Dinopolis Object* type

In this chapter several listings of program code are provided for illustration. The chosen programming language is Java since the prototype has been implemented in Java too. This has been done to verify, that an implementation in an object-oriented language without multiple inheritance is possible. This implies that several parts of the presented code can be implemented more elegant in object oriented language like C++, where templates and multiple inheritance can be used.

8.1 Static and Dynamic Types

Following the requirements described above, 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 8.1 illustrates this behavior. Every *Dinopolis Object* is a composition of a static part (the *Static Type* of the *Dinopolis Object*), which

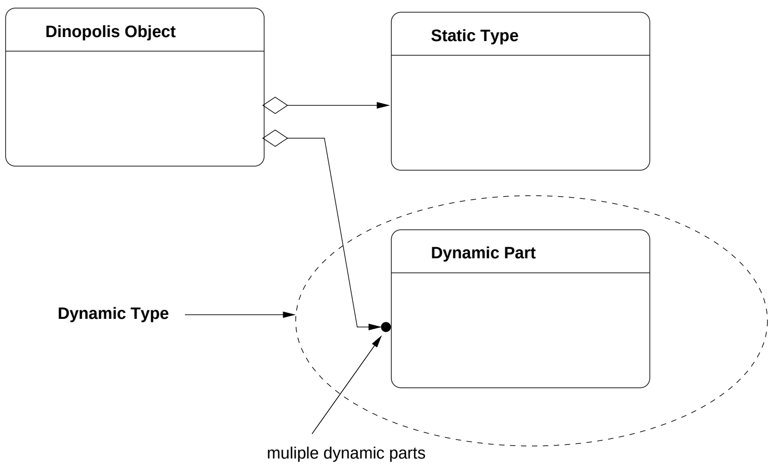

Figure 8.1: Static and Dynamic Types.

never changes during the whole lifetime of this *Dinopolis Object* and zero or more dynamic parts, forming the *Dynamic Type* of this *Dinopolis Object*.

As functionality is provided at method-level known from object-oriented programming languages the following behavior applies:

- Every functionality provided by the static part can be immediately invoked on the *Dinopolis Object*.
- The functionality of the static part is available throughout the whole lifetime of the *Dinopolis Object*.
- Every functionality provided through a dynamic part of a *Dinopolis Object* can be invoked immediately after attaching it to the corresponding *Dinopolis Object*.
- Newly attached functionality may *overlay* already available functionality of a *Dinopolis Object*. This is exactly the same behavior like overriding a method in object-oriented programming.
- Functionality provided through a dynamic part may overlay functionality already provided through the static part.

- The dynamically added functionality may become invalid due several reasons. In this case the overlaid functionality becomes available again.
- Internally it is possible to call the overridden methods, which is known as super- or scope- method-calls.

8.2 Declarations and Definitions

Most todays programming languages differ in the way how methods are declared and how these declarations have to be defined. Sometimes this is not strictly separated, which can lead to a mixup of the meaning of the two terms *Declaration* and *Definition*.

In the *Dinopolis Middleware System* the term *Declaration* stands for meaningfully grouped methods declared with their full signatures. *Declarations* in the sense of *Dinopolis* are used to expose the functionality of *Dinopolis Objects*. Compared to object-oriented programming languages *Declarations* declare the public interface of a *Dinopolis Object*.

Definitions on the other hand are the concrete implementation of such *Declarations*. *Definitions* may implement one ore more *Declarations* and any additional methods they need to work properly.

Since it is important for the *Dynamic Type* mechanism to distinguish at runtime weather a method is part of a *Declaration* or not, *Declarations* have to be marked somehow to expose this special meaning. This is necessary because *Definitions* which implement *Declarations* mostly will have additional methods which are for internal usage only. These methods have no benefit from the *Dynamic Type* mechanism and are therefore not treated specially.

Another important information is weather one ore more *Declarations* are implemented by the *Static Type* or by (dynamic) *Definitions*. This has consequences while removing a *Definition* and is covered in section 8.5.

As an example consider the Java programming language. The marking can be done with an empty interface named `Declaration` and the grouping can be done with an interface which is derived from the `Declaration` interface.

Figure 8.2 shows an example where the `StaticType` implements two declarations namely `Declaration1` and `Declaration2` and serves as a *Definition* for these two *Declarations*. `DynamicDefinition` is another *Definition* implementing `Declaration2` and `Declaration3`. This also demonstrates that (dynamic) *Definitions* may implement *Declarations*, which are implemented by a *Static Type* as well.

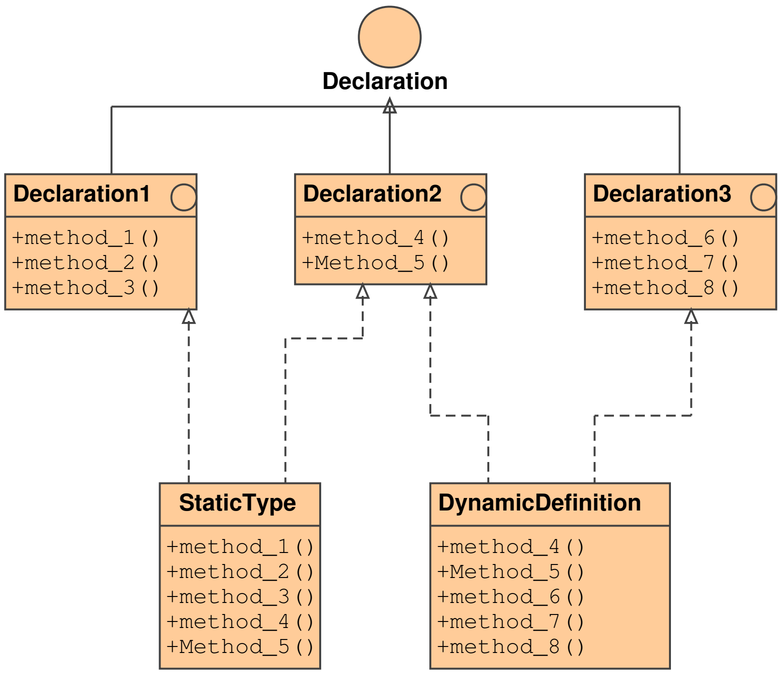

Figure 8.2: Declarations in Static and Dynamic Types

8.3 Adding Functionality

At the programming language level adding new functionality to a class is done by deriving from the class and adding the desired implementation. This must be done at compile time. In the *Dinopolis Middleware System* we have to think in *Declarations* and *Definitions*. Adding functionality to a *Dinopolis Object* has to be done by adding *Declarations* and corresponding *Definitions* to the *Dinopolis Object*.

Functionality can be added to *Dinopolis Objects* at runtime by simply attaching a *Definition* which provides the desired implementation. Since adding functionality to a class changes the class interface, this is true for *Dinopolis Objects* as well. The newly available interfaces or, in terms of *Dinopolis*, *Declarations* are added implicitly to the *Dinopolis Object*. This means that every *Declaration* which is implemented by a certain *Definition* is provided as a new interface, request-able from the corresponding *Dinopolis Object*.

8.4 Overriding Functionality

Overriding functionality at the programming language level means hiding the implementation of the superclass. This is true for the attaching mechanism of *Dinopolis Objects* as well. Whenever a *Definition* gets attached to a *Dinopolis Object*, where already implementations are present for the *Declarations* implemented by this *Definition*, these implementations get hidden behind the newly added ones. This is the same behavior known from virtual methods in C++, or the default behavior of methods in Java, where always the last definition of a method is carried out.

The overridden implementations are still available through a special operation, similar to a super or scope method-call, known from common object-oriented programming languages.

8.5 Removing Functionality

Removing functionality has no equivalent in object-oriented programming languages. It could be simulated by implementing empty method bodies, or methods that immediately raise a *MethodNotSupportedException*. The *Dynamic Type* mechanism provides additional functionality for removing (detach) previously attached *Definitions* at runtime. When detaching a *Definition* from a *Dinopolis Object*, the hidden functionality gets available again. All *Declarations* which were implemented just by the *Definition* to be detached, are not available anymore and thus removed from the list of implemented *Declarations*.

Detaching a *Definition* has to meet one strong requirement. The detach operation is forbidden if it would lead to an "incomplete" state of the corresponding *Dinopolis Object*. A *Dinopolis Object* is said to be "incomplete" if not all methods declared by its *Static Type* are defined through a *Definition*.

8.6 The Dynamic Storage Part

There are cases were the *Static Type* only can express a *Declaration* but cannot provide a *Definition* implementing it: The functionality for storing the persistent data of a *Dinopolis Object* is such a case, which heavily depends on the actual running system configuration. In fact, the *Definitions* carrying out the storage operations must be requested from the corresponding *Virtual System* at runtime.

This is always true if a method has to part of standardized interface of the *Dinopolis Object* but the implementation depends on the actual system configuration.

This kind of *Definitions* are called the *Dynamic Storage Part* of a *Dinopolis Object*.

8.7 Explicitly Scoped Calls

System programmers may be interested in calling a special implementation of a method. The general system behavior is to call the most recently attached implementation the so called *Current Definition* or *Current Binding Target*. The *Explicitly Scoped Call* allows to specify any other previously attached *Definition*, which can carry out the method call. This is very powerful but error-prone and must be handled with care.

8.8 Explicitly Supered Calls

Similar to the explicitly scoped call the *Explicitly Supered Call* allows to call the "super" *Definition* of the *Current Binding Target*. This can be very useful if one is interested in reusing the previously attached *Definition* because otherwise it must be implemented again.

8.9 The *This* Member Variable

The *This Member Variable* provides access to the whole *Dinopolis Object* from within every attached *Definition*. This is needed if a *Definition* wants to use functionality provided by other *Definitions*.

8.10 Implementation Details

8.10.1 The Explicit Virtual Table

The task of the *Explicit Virtual Table* is to provide a dispatching table for method invocation similar to virtual tables known from C++ or Java. *Virtual Tables* are needed if just the binding of a method name to its implementation is known at compile time but not weather this is the last definition of this method in the class hierarchy or not. This is called *dynamic binding* or late binding.

A generic dynamic binding algorithm [VH96] can be defined as follows:

1. Determine the message receiver's class;
2. If the class implements a message with this selector, execute it;
3. Otherwise, recursively check parent classes;
4. If no implementation is found, signal an error.

The implementation of such an algorithm is a broad field of research and optimization since it is in the nature of object-oriented programming to heavily make usage of method invocation.

From this generic algorithm a more specific one for the *Dynamic Type* mechanism can be derived.

1. Determine the last *Definition*, which implements the method called on the *Dinopolis Object*.
2. execute it.
3. if no implementation is found, signal an error.

To determine the last *Definition* a simple table data structure can be used. Every entry describes one method and contains information about the return type, the parameter types, the exceptions and an ordered list of definitions, which provide a implementation for this method.

This table gets initialized while instantiating the corresponding *Dinopolis Object* with the *Static Type* implementation of this *Dinopolis Object* and is a member of this *Dinopolis Object*.

A more efficiently implementation can use a map with the mangled name of the method as a key and the describing data encapsulated in an object as the value.

Furthermore a list of all *Declarations* for which currently a *Definition* is provided can be requested. This *List of Declarations* is initialized with all *Declarations* for which the *Static Type* implementation serves as a *Definition*. This information is always extracted implicitly from the *Definitions*.

A *Dinopolis Object* can always be asked for this list of *Declarations* and type-casted to one of the *Declarations*. This is called a *Dynamic Type System Cast*.

Attaching Definitions

While attaching the appropriate *Definition*, the *List of Declarations* gets updated with all implemented *Declarations* and the *Explicit Virtual Table* gets updated with all methods declared by this *Definition*.

The following policy applies:

- Every *Declaration* which is not currently in the *List of Declarations* is added to this list. *Declarations* which are already present in the *List of Declarations* are ignored.
- For every method in the *Definition* which is declared through a *Declaration* the following applies:
  - If the method is declared through a *Declaration* which was not in the *List of Declarations* a new entry for this method is added to the *Explicit Virtual Table* and initialized with the values for return type exceptions and this *Definition* as the last *Definition*.
  - If there is already a entry for this method in the *Explicit Virtual Table*:

1. The value of the return type of the method to add is compared with the return type in the table-entry for this method. If they differ the attaching operation is aborted and an exceptional state is indicated. The *List of Declarations* and *Explicit Virtual Table* the remain unmodified. This is the expected behavior known from Java and C++ compilers which also abort the compiling process, if the return type of a method changes in an inheriting class.
2. The same behavior is chosen for exceptions. The exceptions of the method to add are compared with the exceptions in the table-entry for this method. If they differ the attaching operation is aborted and an exceptional state is indicated. The *List of Declarations* and *Explicit Virtual Table* remain unmodified.
3. The *List of Definitions* in the table-entry is inspected if it already contains a *Definition* of the same type. More than one *Definition* of the same type would leed to an ambiguity problem and has to be avoided. If there is already a *Definition* with the same type, the attaching operation is aborted and an exceptional state is indicated. The *List of Declarations* and *Explicit Virtual Table* remain unmodified.

Detaching Definitions

While detaching the appropriate *Definition* all *Declarations* which have no implementation anymore are deleted from the *List of Declarations* and the *Explicit Virtual Table* adjusts the *Current Binding Target* accordingly.

The following policy applies:

- First it has to be ensured that the detaching of the *Definition* will not lead into an “incomplete” state of the *Dinopolis Object*. That is, if any method of the *Static Type* gets undefined.

- Next the following applies:
  - The last entry in the *List of Definitions* for the current inspected table-entry is deleted.
  - If the *List of Definitions* for this method is now empty, the table-entry for this method is deleted.
- Every *Declaration* which is not implemented any more is deleted from the *List of Declarations*.

8.10.2 The Virtualizer

It has to be ensured that always the most recently attached implementation for a method call is invoked. To ensure that this behavior applies, an indirection between the method call coming from the outside of a *Definition* and the execution of the implementation of the method is needed. This is true for method calls in the "inside" of a *Definition* as well. The problem is referred to as the *implicit self recursion problem* (see [Szy02] for a detailed discussion).

Consider the following example:

```
public class Definition implements Declaration
{

  public void methodA()
  {
        // implements methodA from Declaration
  }

  public void methodB()
  {
        // implements methodB from Declaration
  }

}


public class AnotherDefinition implements Declaration, AnotherDeclaration
{
  public void methodA()
  {
        // implements methodA from Declaration
  }

  public void methodB()
  {
        // implements methodA from Declaration
  }

  public void methodC()
  {
        // implements methodC from AnotherDeclaration
    methodA();
  }
}
```

First `AnotherDefinition` is attached to the *Dinopolis Object*. Next `Definition` is attached to the *Dinopolis Object* which in turn overrides `methodA()` and `methodB()`.

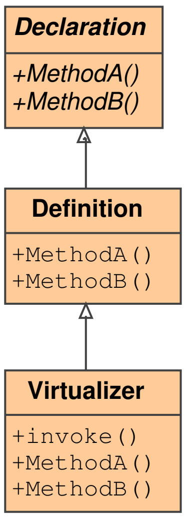

Figure 8.3: Virtualizer Class Diagram

Without additional steps invocation of `methodC()` on the *Dinopolis Object* would brake the desired behavior, that in every case the last *Definition* of a method is called.

We must have the possibility to intervene every method call and check if we already operate on the correct *Definition* or otherwise delegate the method call to the actual *Current Binding Target.* This can be achieved by statically inheriting from every *Definition* and implementing the additional needed steps. This additional class is called the *Virtualizer* of the *Definition.* Figure 8.3 shows the class diagram of the `Virtualizer` for the `Definition` class.

```
public class Virtualizer extends Definition
{
  public void methodA()
  {
    Virtualizer virtualizer;
    virtualizer = (Virtualizer)evt_.getCurrentDefinition("methodA");
    virtualizer.invoke("methodA");
  }

  public methodB()
  {
    Virtualizer virtualizer;
    virtualizer = (Virtualizer)evt_.getCurrentDefinition("methodB");
    virtualizer.invoke("methodB");
  }

  public Object invoke(String method_name)

  {
    if (method_name.equals("methodA"))
    {
      return super.methodA();
    }
    if (mangled_name.equals("methodB"))
    {
      return super.methodB();
    }
  }
}
```

The *Explicit Virtual Table* provides functionality for requesting the *Current Binding Target* of a method call. This *Current Binding Target* is requested by the *Virtualizer* and it issues a special `invoke(method_name)` to it.

The `invoke(method_name)` method then calls the actual implementation of the *Definition* via a super call for the method.

Chapter 9

Summary and Outlook

The concept of the *Dinopolis System Architecture* allows to distribute services over the network in a transparent fashion. The concept of *Dinopolis Objects* can be used to build complex applications relying on that services. Application programmers using *Dinopolis Objects* can focus on the implementation of their client and not on low-level network operations. *Dinopolis Objects* are very flexible and allow runtime re-configuration of their behavior. *Dinopolis Objects* therefore use the *Dynamic Type* mechanism to provide this kind of flexibility. The prototype implementation of the *Dynamic Type* mechanism served as a prove of concept and helped to make some relevant design decisions e.g. to favorite the implicit adding of *Declarations* over the error-prone explicit adding mechanism.

A clear guideline for system programmers regarding the implementation of *Dinopolis Objects* and their *Definitions* has to be made. Further investigations on concrete *Dinopolis Objects* will follow and hopefully show new insights of how easy or how difficult the system is usable and adaptable. Tools like a preprocessor for automatically building the source code of the needed *Virtualizers* have to be implemented.

The core modules of the *Dinopolis System Architecture* are now at the very end of the detailed design phase. The implementation of the overall system will begin in the near future, where C++ will be the implementation language of choice.

References

[Ber96] Philip A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.

[Boo87] Grady Booch. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjaming-Cummings, 1987.

[Boo94] Grady Booch. Coming of age in an object-oriented world. *IEEE Software*, 11(6):33–41, November 1994.

[Dij70] Edsger W. Dijkstra. Notes on Structured Programming. circulated privately, April 1970.

[Emm02] Wolfgang Emmerich. Distributed component technologies and their software engineering implications. In *Proceedings of the 24th international conference on Software engineering*, pages 537–546. ACM Press, 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[KB98] Wojtek Kozaczynski and Grady Booch. Component-based software engineering. *IEEE Software*, 155:34–36, September/October 1998.

[McI69] M. D. McIlroy. “Mass produced” software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages

138–155, Brussels, 1969. Scientific Affairs Division, NATO. Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.

[Mey96] Bertrand Meyer. Object technology: The reusability challenge. *Computer*, 29(2):76–78, February 1996.

[Mey99] Bertrand Meyer. Component and object technology: On to components. *Computer*, 32(1):139–140, January 1999.

[MM99] Bertrand Meyer and Christine Mingins. Component-based development: From buzz to spark. *Computer*, 32(7):35–37, July 1999.

[Sch98] Mike Schup. Standardization efforts in the area of distributed objects. online, 04 1998. `http://www.sis.pitt.edu/~mbsclass/standards/schupp/DistributedObjects.html`.

[Sch02a] Klaus Schmaranz. Dolsa - a robust algorithm for massively distributed, dynamic object-lookup services. submitted to J.UCS., 2002.

[Sch02b] Klaus Schmaranz. A massively distributable componentware system, habilitation thesis, June 2002. Habilitation Thesis.

[Sch02c] Klaus Schmaranz. On second generation distributed component systems. *J.UCS*, 8(1):97–116, 2002.

[Sch02d] Klaus Schmaranz. Robust interrelation management in massively distributed component systems. submitted to J.UCS., 2002.

[Seb93] R.W. Sebesta. *Concepts of Programming Languages*. Benjamin / Cummings, 2nd edition, 1993.

[Szy02] Clemes Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 2 edition, 2002.

[VH96] Jan Vitek and R. Nigel Horspool. Compact dispatch tables for dynamically typed object oriented languages. In *Computational Complexity*, pages 309–325, 1996.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.