TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Building distributed Smalltalk/Java applications using COBRA

van Meer, P.G.A.

Award date:
1998

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

ICS/EB 677

Technische Universiteit Eindhoven

Faculty of Electrical Engineering
Section of Information and Communication Systems

Master's Thesis:

Building distributed Smalltalk/Java applications using CORBA

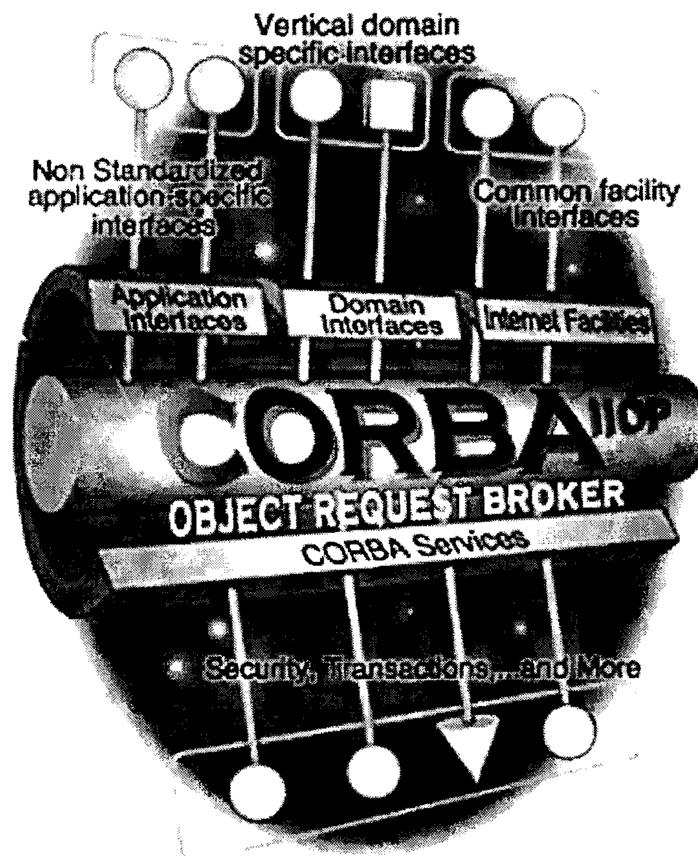
P.G.A. van Meer

Coach:	ir. S. van de Kuilen (ELC Object Technology, Capelle a/d IJssel)
Supervisor:	Prof.ir. M.P.J. Stevens
Examiner:	dr.ir. J.P.M. Voeten
Period:	December 1997

The Faculty of Electrical Engineering of Eindhoven University of Technology does not accept any responsibility regarding the contents of Master's Theses

Building distributed Smalltalk/Java applications using

CORBA



by

P.G.A. van Meer

December 1997

A master thesis project performed for the Information- and Communication Systems group at the Departement of Information Technology at the Eindhoven University of Technology. The project was completed in co-operation with ELC Object Technology B.V.

Preface

This report describes my master thesis project for the Departement of Information Technology at the Eindhoven University of Technology. The project was done together with ELC Object Technology, Capelle a/d Ijssel, which supplied me all the facilities to do my research successfully.

Working at ELC Object Technology was a great experience for me, technically and socially. All the employees of ELC Object Technology were very willing to help me with my problems. Their attitude towards technology, especially Object Technology, and towards each other, inspired me to work with passion on my master thesis project. Also, I appreciated a lot that I was able to explore new technologies like distributed object oriented applications in a very creative way. I am convinced that everything I learned at ELC Object Technology forms a good starting point for the rest of my career as an engineer.

I would like to thank prof. ir. M.P.J. Stevens and the other staff members of the Information- and Communication Systems group at the Eindhoven University of Technology for their support.

This master thesis project I want to dedicate to my mother who brought me were I am now. I wish she could have seen me finishing this work and starting my career. She deserved so much more. Even if she is not here any more, she will always remain my inspiration,

Ralf van Meer
Capelle a/d IJssel
December 1997

Summary

Distributed systems and object-oriented programming are very much related to each other. The structure of object oriented applications is very attractive to distribute because of the fact that every component of the application, called 'object', can be distributed, that is, can be moved to another physical location. This makes applications very flexible and opens new possibilities for development and functionality.

The fast expanding Internet adds again another dimension to software distribution. Applications can be distributed now across the Internet, which makes the software accessible for everyone who has access to the Internet.

Until now, for each application, for each programming language and for each platform, there existed another distributed system, which often was not more than just customised TCP/IP communication between different components.

In 1989, the Object Management Group (OMG) started a project to design a standardised distributed system called 'the Common Object Request Broker' (CORBA). This distributed system reached maturity in 1996 and is designed independent of programming languages and operation platforms. The object's interfaces are described using an independent Interface Description Language (OMG IDL) which describes the operations, attributes, exceptions, constants, etc. of an object. By compiling the IDL to specific program language skeletons, the object's implementation can be added to the heterogeneous distributed system.

All the objects added to the distributed system communicate using the CORBA Object Request Broker (ORB). Together with its services, the CORBA ORB ensures that all objects can find and use other object's services. The communication is done using the Internet Inter ORB Protocol (IIOP), also specified by OMG.

In this report, CORBA is used to build distributed applications with two different languages: Smalltalk and Java. Smalltalk is a proven object oriented language, which is very powerful for developing complex business logic. In contrary, Java, which is only two years old and very much supported by the internet browsers to add applications to the Internet homepages, is not mature enough for using it for complex applications. CORBA can bring this two features (a reliable proven language versus an Internet supported language) together by distributing the business logic (Smalltalk) and view logic (Java in an Internet browser) of the application.

The CORBA's language independent architecture is discussed after which CORBA is focussed on the Smalltalk and Java object oriented languages. Features as the IDL Smalltalk- and Java language mappings, commercial implementations of CORBA for Smalltalk and Java, and performance measurements of CORBA are presented and discussed. To understand CORBA even better a description of the implementation of a Smalltalk server ORB is discussed.

Contents

1. INTRODUCTION 6

1.1. Distributed applications 6
1.2. ELC Object Technology 6
1.3. ELC Object Technology and distributed applications 6
1.4. Assignment 7

2. CORBA 8

2.1. CORBA, the Common Object Request Broker Architecture 8
2.2. Object Management Group 8
2.3. The CORBA reference model 8
2.4. The ORB 9
2.5. ORB interfaces 10
2.6. Object Services 11
2.6.1. Naming service 11
2.7. Common facilities 12
2.7.1. 12
2.8. ORB implementations 12
2.9. CORBA compared with other distributed systems 13
2.9.1. CORBA and DCE 13
2.9.2. CORBA and DCOM 14

3. CORBA IIOP PROTOCOL 15

3.1. CORBA GIOP specification 15
3.2. The IOR 16
3.3. CORBA and the OSI model 17
3.3.1. The OSI model 17
3.3.2. CORBA mapped onto the OSI layers 17

4. CORBA FOCUSSED ON SMALLTALK AND JAVA 19

4.1. Mapping OMG IDL to programming languages 19
4.2. IDL to Smalltalk language mapping 20
4.2.1. Smalltalk naming collisions 21
4.2.2. Smalltalk object mappings 21
4.2.3. Smalltalk type mappings 22
4.3. IDL to Java language mapping 22
4.3.1. Java type mappings 23
4.4. The Smalltalk typing problem 23
4.4.1. Solution using the Any-type 23
4.4.2. Solution using VisualAge's Public Interface Editor 25
4.5. Smalltalk class methods 25

5. COMMERCIAL IMPLEMENTATIONS OF CORBA ORBS FOR SMALLTALK AND JAVA 27

5.1. Smalltalk 27
5.2. Java 27
5.3. IONA's ORBIX for VisualAge for Smalltalk 28
5.3.1. Orbix components 28
5.3.2. ORB connection manager 28
5.3.3. Orbix's CORBA services 28
5.4. Visigenic's VisiBroker for Java 29
5.4.1. VisiBroker's features 29
5.4.2. Using VisiBroker to contact objects outside the ORB. 30
5.5. The Performance of a Smalltalk server and Java client 31

5.5.1. Measuring variable method invocations ... *32*
5.5.2. Measuring variable parameters in one invocation ... *34*

6. IMPLEMENTATION OF A CORBA FRAMEWORK ... **36**

6.1. Implementation of the IIOP Protocol in Smalltalk ... 36
6.1.1. CORBAType classes ... *36*
6.1.2. IIOP protocol classes ... *38*
6.2. Implementation of a server ORB in Smalltalk ... 40
6.2.1. Interface repository ... *41*
6.2.2. The ORB ... *44*
6.3. Using VisualAge to specify the interface ... 44

7. CONCLUSIONS AND RECOMENDATIONS ... **46**

APPENDIX A. COMMON DATA REPRESENTATION (CDR) ... **47**

APPENDIX B. ALIGNMENT OF IDL TYPES ... **48**

APPENDIX C. THE CORBA ANY-TYPE ... **50**

APPENDIX D. GIOP MESSAGE DEFINITIONS ... **51**

APPENDIX E. OBJECT SERVICES ... **57**

APPENDIX F. CORBA OBJECT REQUEST BROKER IMPLEMENTATIONS ... **58**

APPENDIX G. SOURCE CODE, CORBATYPE CLASSES ... **60**

APPENDIX H. SOURCE CODE, GIOPMESSAGE CLASSES ... **72**

APPENDIX I. SOURCE CODE, REPOSITORY CLASSES ... **85**

APPENDIX J. PERFORMANCE MEASUREMENT RESULTS ... **95**

APPENDIX K. REFERENCES ... **96**

1. Introduction

1.1. Distributed applications

Distributed systems are becoming more and more reality today. Information can be stored at multiple locations across the network. Doing this makes applications more powerful and flexible to use and opens new possibilities for the application's development and performance. Due to the fast expanding internet infrastructure, new attractive possibilities for distributing your applications arise. That's why the need for distributed applications is still growing enormously.

In a distributed object oriented application environment, the objects of which the application consists, can be widely spread over different locations in the network. The objects can communicate with each other over the network when they need information. For example, an object, which represents the user interface, can communicate with objects that can access a database at a complete different physical location.

For different platforms and programming languages there are now many different systems to achieve object distribution. All systems work well but when you want to use different platforms and programming languages in forming one distributed application, the present techniques fail.

The in 1989 founded Object Management Group solved this problem by creating a standard for distributed systems. This standard, which reached maturity in august 1996 is called the 'Common Object Request Broker Architecture' or abbreviated 'CORBA'. It became mature in its recent release: 'CORBA 2.0'.

1.2. ELC Object Technology

ELC Object Technology is a division of the ELC Information Services IT company with several settlements in The Netherlands. ELC Object Technology builds customised applications using Object Oriented software analysis, design and implementation strategies. Doing this, ELC Object Technology prefers to use the powerful object oriented programming language Smalltalk together with its proven development environment: IBM VisualAge for Smalltalk.

1.3. ELC Object Technology and distributed applications

When building applications, ELC uses a self developed Persistency Framework, which provides a connection between the application and a persistency database. Looking at this configuration, three groups of objects can be distinguished: The objects, which cope with the database (Persistency Framework), the objects that form the actual application (intelligent objects) and the objects that form the user interface. This configuration is called the Three Layers Model: Persistency layer, Business layer, View layer.

ELC Object Technology wants to design its view layer using the platform independent- and user friendly facilities of an internet browser by implementing the view logic using Java applets. The communication between these Java applets and the objects in the business layer (programmed using Smalltalk) has to be established using object distribution. Because the software must be platform independent and has to run in a heterogeneous Smalltalk/Java environment, CORBA should be able to solve this problem. Both Smalltalk and Java have been supplied with CORBA features. Thus a basis for the solution already exists.

1.4. Assignment

Because ELC Object Technology wants to combine the internet browser facilities with the powerful Smalltalk environment, it has formulated the following assignment for this master thesis project:

- **Explore the CORBA standard**. This is described in chapter 2 and 3 in which the CORBA architecture is presented together with comparisons to other distributed systems.
- **Explore the facilities to support CORBA in Smalltalk and Java**. In chapter 4, CORBA is focussed on Smalltalk and Java. The CORBA language mappings for these languages are described and discussed. And several commercial implementations for these languages are discussed
- **Explore the usefulness of these tools to build distributed applications**. In chapter 5, two implementations of CORBA, ORBIX for Smalltalk and Visigenic's Visibroker for Java are discussed.
- **Build a framework to build distributed applications between Smalltalk and Java using the CORBA standard**. In chapter 6, the construction of a Smalltalk CORBA server ORB is described, constructed upon IIOP, together with extensions on the ORBIX CORBA implementation

2. CORBA

2.1. CORBA, the Common Object Request Broker Architecture

CORBA stands for Common Object Request Broker Architecture. CORBA is a standardised way to build distributed applications in multi-language (heterogeneous) and multi-platform environments. To understand CORBA I shall first explain its abbreviation:

- **Common**. Common stands for the group of objects that together form a distributed application. These common objects can differ in implementation language, operation platform and physical location.
- **Object Request Broker**. Object Request Broker or shortly ORB stands for the core part of the distributed system. The ORB can be compared with a telephone exchange, which takes care of the communication between different objects. One object needs services from another object - or more abstract - a client needs the services from an object implementation. The ORB is responsible for sending the request of the client to the object implementation and then returns the reply of the object implementation to the client.
- **Architecture**. Architecture stands for the description of the Common Object Request Broker. It is important to state that CORBA is only architecture, not an implementation, of a distributed object system. The actual architecture of CORBA is specified by OMG, the Object Management Group [OMG 1995a].

2.2. Object Management Group

> *"The Object Management Group, Inc. (OMG) is an international organisation supported by over 500 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object oriented technology in software development. The organisation's goal consists of the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conform these specifications it will be possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems".*

OMG's core part of its business is the Object Management Architecture (OMA). The OMA provides the basic structure upon which all OMG specifications - especially CORBA - are based. For a complete description of the OMA see [OMG 1997a].

2.3. The CORBA reference model

The key to understand the structure of the OMG's CORBA architecture is the CORBA Reference Model. From this model, all CORBA features are specified. It consists of the following components:

- **Object Request Broker**, or short 'the ORB', which enables objects to make and receive requests and replies in a distributed environment without bothering about the receiving object's implementation or location. It is the foundation for building distributed applications and the core part for interoperability between distributed applications in heterogeneous environments. The architecture of the CORBA Object Request Broker is described in this chapter.
- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains. For example, the Life Cycle Service defines conventions for creating, deleting, copying, and moving objects. It does not limit the implementation of the objects in an application. Specifications for Object

Services are contained in CORBAservices: Common Object Services Specification. [OMG 1995b]. An example of a CORBA service, the CORBA Naming Service, is described in this chapter.

- **Common Facilities,** a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility. Information about the Common Facilities will be contained in CORBAfacilities: Common Facilities Architecture. [OMG 1995c]
- **Application Objects,** which are products of a single vendor on in-house development group which controls their interfaces. Application Objects correspond to the traditional notion of applications, so they are *not* standardised by OMG. Instead, Application Objects constitute the uppermost layer of the Reference Model. The Object Request Broker, then, is the core of the Reference. (For more information about the OMG Reference Model and the OMG Object Model, refer to the 'Object Management Architecture Guide' [OMG 1997a]).

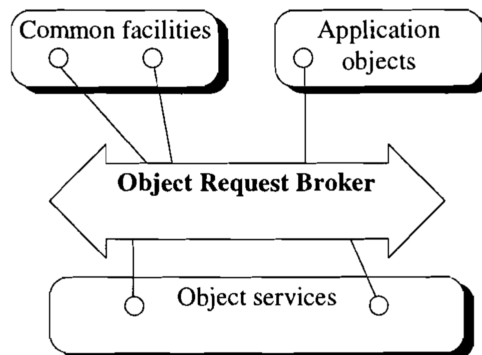


Figure 2-1 The CORBA reference model

2.4. The ORB

The CORBA Object Request Broker (ORB) functions like a telephone exchange, which connects a client to a service (most commonly another person). In distributed applications, this service is represented by an implementation of an object or a group of objects. When a client tries to achieve a service from an object implementation the ORB is responsible for the following tasks:

Request
- To **find** the object implementation for the request
- To **prepare** the object implementation to receive the request
- To **communicate** the data making up the request

Reply
- To **find** the client again for receiving the reply
- To **prepare** the client for receiving the reply
- To **communicate** the reply data to the client

The interface the client and object implementation see is completely independent of **where** they are located, **what** programming language they are implemented in, **which** platforms are used, or any other aspect which is not defined in the interfaces.

If you want to call someone, you need a telephone number to specify another telephone. The client of an ORB can perform a request on an object implementation when it has the following information:

- An **Object Reference** (IOR, Internet Object Reference) that uniquely describes an object subscribed to an ORB

- The **interface** of the object
- The desired **operation/attribute/exception** etc. it wants to perform/obtain.

2.5. ORB interfaces

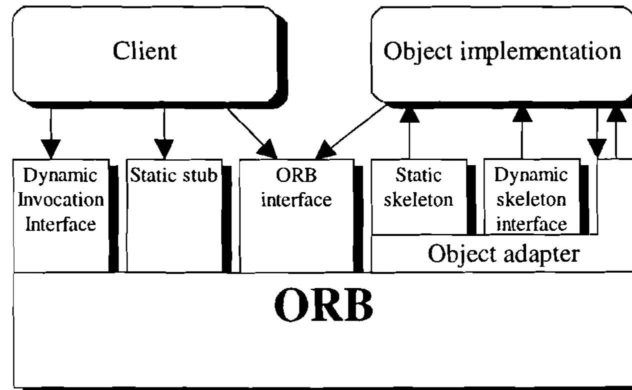


Figure 2-2 The structure of Object Request Broker interfaces

Figure 2-2 shows the structure of a CORBA Object Request Broker (CORBA ORB) together with its interfaces. The interfaces to the ORB are shown between the client/object implementation and the ORB. The arrows indicate whether the ORB is called by the client/object implementation or the ORB performs an upcall using the interface.

To make a request, the client can use the following interfaces:

- **Dynamic Invocation interface** (the same interface independent of the target object's interface). Due to this interface's dynamic structure it is able to communicate with object implementations that have been created during runtime.
- **A static OMG IDL stub** (the specific stub depending on the interface of the target object). Due to this interface's static structure the object implementations interface has to be known before runtime.
- **The ORB interface**. The client can also directly interact with the ORB for some functions.

The object implementation receives a request as an upcall through the following interfaces:

- **A static OMG IDL generated skeleton**. By specifying the interface before runtime using IDL (Interface Description Language) the ORB knows how to cope with the object implementation.
- **Dynamic skeleton interface**. Due to this interface's dynamic structure it is possible to act as a completely new object implementation during runtime.
- **The ORB interface**. Like the client, the object implementation may call the Object Adapter and the ORB for extra support.

So, definitions of the interfaces to and from object implementations can be defined in two ways:

- Interfaces can be defined **statically** in an interface definition language, called the OMG Interface Definition Language (OMG IDL or shorter IDL). This language defines the protocol of objects according to the operations, attributes, exceptions, constants and typedefs it provides. Thus the object is defined by its interface description using IDL.
- Alternatively, or in addition, interfaces can be defined **dynamically**. They can be added to an Interface Repository service in the ORB, which represents the components of an interface as objects, permitting dynamic runtime access to these components. The object implementation information which is provided at installation time (can be during runtime) is stored also in the Implementation Repository and can be used by the ORB for request delivery.

The statically defined IDL interfaces and the dynamically defined interfaces have equivalent expressive power. It makes no difference for a client whether the object implementation expresses its interface using static or dynamic descriptions. And vice versa for the object implementation.

2.6. Object Services

The object services are interfaces that are used by many distributed applications. Object services are location, platform and language independent and provide additional functionality to the ORB. Some services are so crucial that a distributed application could not function normally without them (e.g. Naming service and Life cycle service). Examples of services are shown in Appendix E. Here, only the naming service, which is of essential importance for a CORBA distributed application to function normally, is described as a short example.

2.6.1. Naming service

The naming service locates object implementations with a specified name. This is a fundamental service for distributed object systems because the Interoperable Object Reference (IOR) for the appropriate object implementation is not always available for the client.

The naming service provides a mapping between a name and an IOR. Storing such a mapping in the naming service will be called 'binding an object'. Removing an entry will be called 'unbinding'. Obtaining an IOR that is bound to a name is known as 'resolving the name'.

Names can be hierarchically structured by using contexts. Contexts are similar to directories in file systems and they can contain names as well as subcontexts.

The use of IORs alone to identify objects has two problems:

- IORs as stand-alone entities are difficult for human users to cope with because they are opaque data types and their stringified form (which can be obtained from the ORB interface) is a long sequence of numbers.
- When a service is restarted, its objects mostly have new IORs. But, clients want to use the object implementation's service continuous without having to check whether or not the IOR has been changed.

The typical use of the naming service involves object implementations binding to the naming service when they come into existence and unbinding before they terminate. A client resolves names, which produce objects on which they can invoke operations.

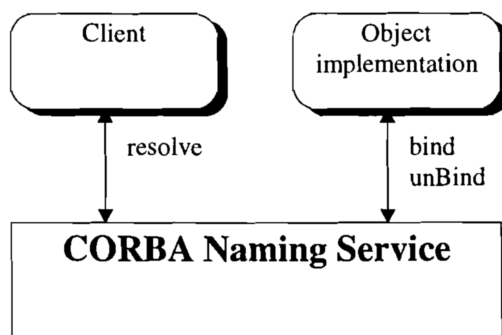


Figure 2-3 CORBA Naming Service

2.7. Common facilities

While the ORB specifies a system's core component, the object services represent its most basic and essential features for functionality. It provides the essential interfaces needed to create an object, introduce it into its environment, use and modify its features. Common Facilities are the final area of the Object Management Architecture to be defined. They fill the gap between the enabling technology defined by CORBA and the Object Services, and the application-specific (not standardised) services.

Some examples of Common Facilities include email and printing. These types of Common Facilities are needed in most application domains. In addition, there are many companies working on more specialised Common Facilities, such as system management.

Common Facilities are separated into two categories:

- **Horizontal Common Facilities**, which are shared by many or most systems. There are four major sets of these facilities: User Interface, Information Management, Systems Management and Task Management.
- **Vertical Market Facilities**, which support the domain-specific tasks that are associated with vertical market segments. Some Vertical Market Facilities may migrate to Horizontal Common Facilities. Services that are common across many vertical facilities areas are candidates for horizontal facility status.

The boundaries separating Common Facilities from Application Objects and from Object Services are quite vague. They reflect the evolution of object system technology. The current placement of the boundaries reflects the current OMG standardisation effort. As experience in an application area matures, areas of potential new Common Facilities will be discovered and defined, just as evolving system infrastructures will gradually incorporate pieces of the Common Facilities domain into their basic Object Service offers.

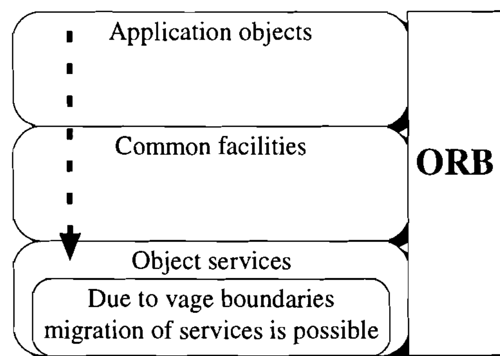


Figure 2-4 Migration of object services

2.8. ORB implementations

In the architecture, the CORBA ORB is not required to be implemented in one stiff way. Rather it is defined by its interfaces, which are defined by OMG. Any ORB implementation that provides the appropriate CORBA 2.0 interface is acceptable and can be called CORBA 2.0 compliant.

Different ORBs may make quite different implementation choices, and, together with the IDL compilers, repositories, and various Object Adapters, provide a set of services to clients and implementations of objects that have different

properties and qualities. There may be multiple ORB implementations (also described as multiple ORBs) which have different representations for IORs and different means of performing invocations on objects. It may be possible for a client to simultaneously have access to two IORs managed by different ORB implementations. When two ORBs are intended to work together, those ORBs must be able to distinguish their IORs. It is not the responsibility of the client to do so. There are now several commercial implementations, which are CORBA 2.0 compliant (see Appendix F).

2.9. CORBA compared with other distributed systems

Because ELC had to decide what policy to choose for building distributed applications I investigated for other distributed systems. Two distributed systems, besides CORBA, appeared to be interesting for building professional distributed applications:

- DCE (Distributed Computing Environment)
- DCOM (Distributed Common Object Model)

2.9.1. CORBA and DCE

DCE (Distributed Computing Environment) supports the construction and integration of C-based client/server applications in heterogeneous distributed environments. DCE has been implemented and designed by the Open Software Foundation (OSF).

The most crucial difference between DCE and CORBA is that DCE was designed to support procedural programming, while CORBA was designed to support object-oriented programming. CORBA supports all of the characteristics of object oriented programming styles, with the possibility of creating new objects at runtime.

Distributed procedural programming environments such as DCE support a different set of capabilities than object oriented distributed environments. However, DCE does have additional capabilities that begin to overlap with traditional capabilities of object-oriented systems:

- A DCE client can determine at runtime the specific servers to which it will bind and make RPCs (however the interfaces supported by those servers must be fixed at compile time).
- A DCE server may generate so called object UUIDs (universal unique identifiers), to denote different resources managed by the server. A client that does an RPC to the server can use an object UUID to identify a specific resource. For example, a print server might generate object UUIDs for the different printers it controls, and a client submitting a print request would specify the desired printer. UUID can be compared with CORBA Object References.
- A DCE server may also generate so called object type UUIDs, associate each object UUID with an object type UUID, and register a separate set of RPC handlers for each object type UUID. When a client does an RPC to the server and specifies an object UUID, the specific function that is invoked in the server depends on the object type with which the object UUID is associated. For example, the print server might associate one object type UUID with RPC handlers that support line printers and another object type UUID with a corresponding set of RPC handlers that supports PostScript printers.

Although the most significant difference between DCE and CORBA is the style of programming, there are still other differences between CORBA and DCE as shown in Table 2-1.

CORBA	DCE
Object Management Group (OMG)	Open software foundation (OSF)
IDL based on C++ syntax	IDL based on C syntax
IDL supports multiple inheritance	IDL supports no inheritance
Support of dynamic invocation	Does not support dynamic invocation
Based on an ORB which controls the server activation	Based on RPC. The user has to activate the server
Does not support the pointer type (for C)	Supports the pointer type
Only supports request/reply context	Supports real application contexts
	Supports pipelining for large data structures

Table 2-1 CORBA against DCE

2.9.2. CORBA and DCOM

DCOM (Distributed Common Object Model) is the distributed extension of the Component Object Model (COM) which grew from Microsoft's work on OLE (Object Linking Embedding). The full set of these technologies is called ActiveX.

The differences between CORBA and DCOM are less clear than the differences between CORBA and DCE. Because DCE is designed for procedural programming an important distinction between DCE and CORBA can instantly be made.

However, DCOM is, like CORBA, designed to build distributed applications using an object-oriented environment. The key decision which system to use cannot be made on programming style in this case.

CORBA implementations exist on almost every platform, and have been used successfully in many large projects. But its complexity and the industry's failure in the past to agree on interoperability between CORBA ORBs have cost its wide commercial acceptance.

DCOM is still new as a commercial product. Microsoft is working to make COM and DCOM generally available on the Macintosh. A Solaris version of DCOM developed in partnership with Software A.G with general availability since for April 1997. Beta versions for Digital Unix, Linux (on Intel platforms), and IBM mainframes are expected soon this year (1997).

CORBA is more mature as a cross-platform technology, while DCOM has an army of developers who already know COM programming. COM also benefits from user-friendly Windows tools. In [OMG 1995a] OMG describes the standard to interoperate with DCOM.

In Table 2-2 the most important differences between CORBA and DCOM are listed.

CORBA	DCOM
Object Management Group (OMG)	Microsoft
Many different vendors who build for -and develop CORBA -> increases quality of product.	Only Microsoft
More mature, since 1992	Since 1996

Table 2-2 CORBA against DCOM

For practical tests comparing CORBA and DCOM see [MON 1997] and [POM 1997].

3. CORBA IIOP protocol

When separate ORBs want to use each other's subscribed objects they need to communicate with each other. Therefore there has to be a protocol for communication among ORBs so they can interoperate. OMG specified a protocol called GIOP (General Inter ORB protocol). The GIOP is an abstract protocol, which does not care about the transport layer (though it makes some assumptions to it). For the actual communication using the internet, which uses the TCP/IP transport protocol, OMG specified the IIOP (Internet Inter ORB Protocol). IIOP is currently the only implementable protocol OMG specified.

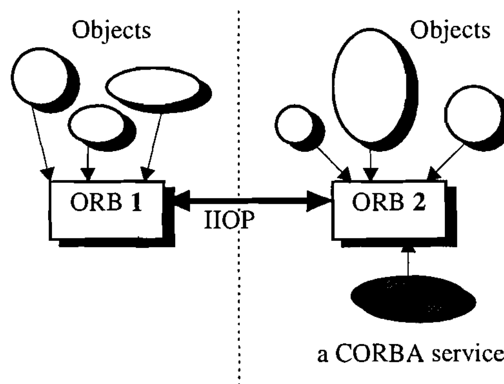


Figure 3-1 ORBs communicating using IIOP

While describing the IIOP, first an overview of the GIOP specification is presented after which the additional feature on GIOP, specified by the IIOP is described. This additional feature is the Interoperable Object Reference, the IOR.

3.1. CORBA GIOP specification

The GIOP protocol consists of a set of 7 different messages, which are client/server specific. While describing these GIOP messages, it is necessary to define client and server roles. A client is the party that opens a connection and sends requests. A server is the party that accepts connections and receives requests. So, the connection will always be in one direction. This makes things more easy because dealing with a unidirectional network while using more connections (each with its own direction) is similar to dealing with a bi-directional network.

The 7 GIOP message are summarised in Table 3-1, which shows the message type names, whether the message can be sent by client, server, or both, and the value used to identify the message type in GIOP message headers.

Message type	Originator	Value
Request	Client	0
Reply	Server	1
CancelRequest	Client	2
LocateRequest	Client	3
LocateReply	Server	4
CloseConnection	Server	5
MessageError	Client/Server	6

Table 3-1 GIOP Message types

As shown in Table 3-1 a GIOP message (MessageTypeHeader) can contain seven different types of messages. The complete GIOP message is build out of three parts:

- **MessageHeader**. This is a 12-octet header, which contains data such as byte order, message type and total message size.
- **MessageTypeHeader**. The MessageTypeHeader can be one of the seven messages as shown in Table 3-1. The MessageTypeHeader can be empty in case of the CloseConnection or MessageError message.
- **MessageTypeBody**. The message body contains the actual bulk of data of the message. The Request, Reply or LocateReply message can have a message body.

In Appendix D I shall describe the properties of GIOP messages using Pseudo-IDL. This is also used in [OMG 1995a] to define the GIOP messages and has proven to be quite satisfying in the specification for the GIOP protocol due to the fact that GIOP is used to send messages on behalf of OMG IDL specified interfaces.

3.2. *The IOR*

IIOP adds the transport layer dependent features to the GIOP specification. Most important it describes the Interoperable Object Reference (IOR) which locates objects across a TCP/IP network.

The data structure of the IOR, which is shown in Figure 3-2 is not to be used internally to any given ORB, and is not intended to be visible to application level ORB programmers. It should be used only when a request is invoked on an object subscribed to another ORB.

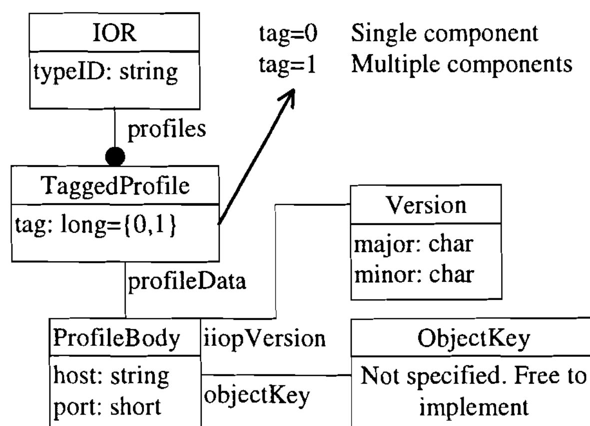


Figure 3-2 Sturucture of the IOR

IORs can be 'stringified' by the ORB interface using the 'object_to_string' operation. They can be turned into a CORBA object using the ORB interface's 'string_to_object' operation.

This stringified representation ensures that ORBs could address directly objects in a foreign ORB. Normally, a CORBA service, that is subscribed to a foreign ORB, can be obtained using an IOR string. When this remote CORBA service is a Naming service, objects binded to this service can be accessed without using IORs.

3.3. CORBA and the OSI model

Now that CORBA and its inter-ORB protocol (IIOP) are described, CORBA ORBs can be mapped onto the seven different layers of the OSI model to get a better understanding of the design structure of CORBA.

3.3.1. The OSI model

The OSI model (Open Systems Interconnection) deals with connecting systems that can share their resources to each other. It has seven layers that all represent a different level of abstraction, see [TAN 1996].

7. **The application layer**. This layer represents a certain application that can use the network to deliver its services (e.g. electronic mail, ftp and telnet).
6. **The presentation layer**. This layer maps data types (such as String or Integers) into a protocol to send it onto the network.
5. **The session layer**. This layer adds some low-level services to the Transport layer such as dialogue control and synchronisation.
4. **The transport layer**. This layer sets up connections with other communications sessions and makes sure every message gets where is has to be.
3. **Network layer**. This layer controls the network traffic. It takes care of routing and buffering problems and defines descriptors for source and destinations locations.
2. **Data link layer**. This layer makes sure that the 'raw' bits travel across the wire in the right order and without transmission errors.
1. **Physical layer**. This layer communicates 'raw' bits across the wire.

3.3.2. CORBA mapped onto the OSI layers

7. **Application layer**. The ORB interface represents the application layer. The distributed application's stubs and skeletons send their requests and replies to the ORB, using the ORB interface, which starts entering the communication process.
6. **Presentation layer** . The marshalling of primitive data types and CORBA objects represented as IORs into IIOP represents the presentation layer. This marshalled data types are the parameters for the requested or replied methods. The marshalling is done by the ORB.
5. **The session layer**. The connection management of the ORB. A CORBA implementation (e.g. Orbix, see paragraph 5.3) is free to implement its own connection management. OMG only specifies some assumptions regarding the transport behaviour (connection-oriented, reliable, byte-stream communication and error notification. See [OMG 1995a]).

4. - 1. The **transport-**, **network-**, **data link-** and **physical layer** represent the TCP/IP protocol upon which IIOP is build.

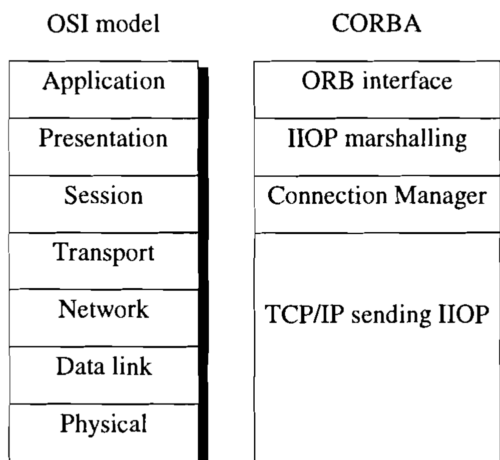


Figure 3-3 CORBA mapping onto the OSI model

4. CORBA focussed on Smalltalk and Java

Before I start discussing CORBA concerning Smalltalk and Java, first OMG IDL has to be discussed, in the context of these two languages, to get an idea about the features IDL consists of.

4.1. Mapping OMG IDL to programming languages

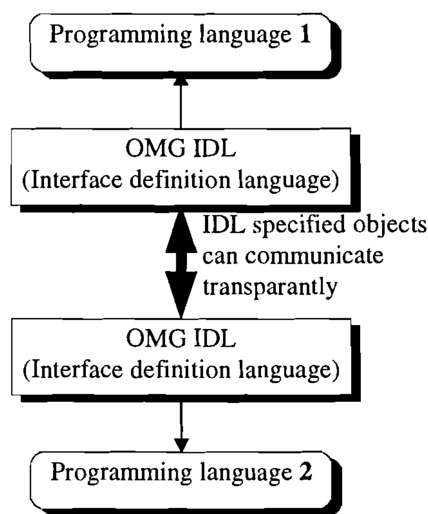


Figure 4-1 Language mappings to IDL

The OMG Interface Definition Language (OMG IDL or IDL) defines the types of objects by specifying their interfaces. An interface consists of a set of named operations (and the parameters to those operations), attributes, exceptions, constants etc. Note that IDL provides the descriptive framework for the objects to be manipulated by the ORB, it is **not** necessary there is implementation source code available for the ORB to work. As long as the equivalent information is available in the form of stub routines (static) or a runtime interface repository (dynamic), a particular ORB may be able to function correctly.

IDL is the medium a particular object implementation tells its possible clients what operations are available and how they should be invoked. So actually IDL describes the object's protocol. From the IDL definitions, it is possible to map CORBA objects into particular programming languages or object systems, as shown in Figure 4-1.

The next example is an IDL source code that describes something like a person-database:

```
module PersonDataStorage{
      interface Address{
            attribute short houseNumber;
            ...
      };
      interface Person{
            Address getAddress();
            void setAddress(in Address anAddress);
            string name();
            ...
      };
};
```

Example 4-1 IDL source code

In Example 4-1, the IDL source code describes a person database. The interfaces (protocols) of two objects ('Address' and 'Person') are described in the module 'PersonDataStorage'. With its interface description, the object implementation 'Address' is defined to be an implementation that has the attribute 'houseNumber'. The 'houseNumber' attribute is determined to return a short typed variable that likely contains the house number of the 'Address' object implementation.

The other object implementation called 'Person' contains information about a person. It can accept the messages: 'getAddress()', 'setAddress(in Address anAddress)' and 'name()'. The 'getAddress()' method returns a parameter that is an 'Address' object. In practice this return parameter will be an IOR (Internet Object Reference) to a new instantiated 'Address' object. The other methods in this interface speak for themselves.

The actual programming language for the 'Address' and 'Person' object is not important is this case. Just because the object implementation's interface is described in IDL makes it accessible for every client through a CORBA ORB.

A particular mapping of OMG IDL to a programming language should be the same for all ORB implementations. Language mapping includes the definition of the language-specific data types to access objects through the ORB.

Originally OMG supported three language mappings in the release of CORBA 2.0 [OMG 1995a]:

- Mapping for C
- Mapping for C++
- Mapping for Smalltalk

Later OMG adopted other language mappings into its standard:

- Ada Language Mapping, 19 march 1996
- C++ Language Mapping 1.1, 19 march 1996
- IDL COBOL Mapping, 11 march 1997
- IDL Java Mapping 1.0, 24 June 1997

4.2. IDL to Smalltalk language mapping

The OMG IDL to Smalltalk mapping was part of the initial release of the CORBA 2.0 specification by OMG.

The mapping of OMG IDL to Smalltalk was designed with the following goals:

- A minimum protocol that additional classes to the Smalltalk Common Base classes need for achieving the IDL mapping is described in [OMG 1995a, chapter 19].
- Whenever possible, OMG IDL types are mapped directly to existing, portable Smalltalk classes (from the Smalltalk Common Base), see [GOL 1980].
- The Smalltalk mapping only describes the public (client) interface to Smalltalk classes and objects supporting IDL. Individual IDL compilers or CORBA implementations might define additional private interfaces.
- Because of the dynamic nature of Smalltalk, the mapping of the any (and union) type do not need an explicit mapping. Instead, the value of the any (and union) type can be passed directly to the object after conversion by the ORB.
- The explicit passing of environment and context values on operations is not required.
- Except in the case of object references, no memory management is required for data parameters and return results from operations. All such Smalltalk objects reside within Smalltalk memory, so garbage collection will reclaim their storage when the ORB) no longer references them.

4.2.1. Smalltalk naming collisions

One enormous shortcoming of OMG IDL is the operation name conversion. OMG has specified the mapping of an IDL operation to a Smalltalk method selector as follows:

IDL operation declaration	Smalltalk selector
method();	#method
method(in any arg1);	#method: anAny
method(in any arg1,in any with);	#method: anAny1 with: anAny2

The above does not seem to be a problem but when IDL is generated from Smalltalk source code (the other way around), many problems arise because, in the OMG specification of IDL, OMG does not allow two operations declarations to have the same name. method(), method(in any arg1) and method(in any arg1, in any with) are therefore not accepted together in one interface declaration. I solved this problem by specifying the IDL operation naming as follows:

Smalltalk selector	IDL operation declaration
#method	method();
#method: aParameter	method(in any arg1);
#method: aParameter1 with: aParameter2	methodWith(in any arg1, in any with);

As can be seen, this solution does not eliminate naming collision between #message and #message:. But, in practice, this proved to work out well.

The use of underscore characters in OMG IDL identifiers is not allowed in all Smalltalk implementations. Thus, a conversion algorithm is required to convert names used in OMG IDL to valid Smalltalk identifiers. To convert an OMG IDL identifier to a Smalltalk identifier, remove each underscore and capitalise the following letter (if it exists). Notice that naming collisions are possible when you apply the above description (when 'method_with();' and 'methodWith();' are both in the same interface description).

4.2.2. Smalltalk object mappings

Each OMG IDL interface defines the operations that IORs with that interface must support. In Smalltalk, each OMG IDL interface defines the methods that IORs with that interface must respond to. Implementations can map IDL interfaces to:

- a single Smalltalk class for every interface (most common)

- all IDL interfaces to a single Smalltalk class
- several Smalltalk classes to several OMG IDL interfaces.

The design goals permits the mapping to ignore memory management, since Smalltalk handles this itself (through garbage collection).

A CORBA object is represented in Smalltalk as an Interoperable Object Reference (IOR) which uniquely describes an object in an ORB. The object must respond to all messages defined by that CORBA object's interface. An IOR can have a value, which indicates that it represents no CORBA object. This value is the standard Smalltalk value nil.

4.2.3. Smalltalk type mappings

IDL type	Smalltalk type class
array	Array
boolean	Boolean ('true' or 'false' objects)
char	Character
float, double	Float
(unsigned) long, (unsigned) short, octet	Integer
sequence	OrderedCollection
string	String
exception	Dictionary
struct	Dictionary
any	Smalltalk object that can be mapped into an IDL type
constant	Smalltalk CORBAConstants dictionary
enum	Smalltalk objects that implement the CORBAEnum protocol
union	Smalltalk object that implement the CORBAUnion protocol
Object Reference (IOR)	Smalltalk object that responds to the IOR's interface
IDL operation	Smalltalk message
IDL attribute	Smalltalk getter- and setter methods

See for a detailed description of the Smalltalk IDL type mapping [OMG 1995a, chapter 20].

4.3. IDL to Java language mapping

The OMG IDL specification is modelled after C++. It includes such constructs as typedef, enum, const, attribute, struct, module, and interface. Additionally, it contains several data types, such as byte, long, string, and float. Because the Java language is also modelled after C++, the IDL to Java mapping is quite straightforward.

The mapping of OMG IDL to Java (1.0.2), see [OMG 1997b], was designed with the following goals:
- Client-side and server-side source code portability (transparency)
- ORB replaceability
- Binary compatibility between client stubs (and server skeletons) and ORBs.

4.3.1. Java object mappings

An IDL interface is mapped to a public Java interface with the same name, and an additional "helper" Java class with the suffix Helper appended to the interface name which takes care of the 'narrowing' of a CORBA object to the specific class.

Because IDL supports multiple inheritance, the IDL interfaces are mapped to the Java interface which also support multiple inheritance. Java classes do not support multiple inheritance.

The design goals permits the mapping to ignore memory management, since Java handles this itself (through garbage collection).

A CORBA object is represented in Java as an Interoperable Object Reference (IOR) which uniquely describes an object in an ORB. The object must respond to all messages defined by that CORBA object's interface. An IOR can have a value, which indicates that it represents no CORBA object. This value is the standard Java value null.

4.3.2. Java type mappings

The Object Management Group has established standards for mapping IDL into the Java language. The full specification maps the entire IDL set into associated Java keywords.

IDL type	Java type
array	fixed array of the type
boolean	boolean
char	char
float	float
double	double
long	int
octet	byte
sequence	array of the type
short	short
string	java.lang.String
exception	CORBA.Exception
struct	A class representing the struct
any	CORBA.Any
constant	public static final
enum	A class representing the enum containing ints
union	A class representing the union
Object Reference (IOR)	Java object that responds to the IOR's interface
IDL operation	Java operations
IDL attribute	Java getter- and setter operation

IDL numeric data types can be signed or unsigned. Since Java only supports signed numbers, all unsigned IDL numeric identifiers will convert to the Java data type that is one size larger than it's signed counterpart. For example, unsigned char values in IDL convert to signed int variables in Java.

For more information about the IDL to Java mapping see [OMG 1997b].

4.4. The Smalltalk typing problem

When you want to distribute Smalltalk Objects with CORBA one important problem arises. CORBA describes the interfaces of objects using IDL. This is a **strongly typed** language, which specifies the input- and output types of methods and attributes of objects. Smalltalk, in contrast, is **totally untyped**. The input parameters and the result type of a method performed on an object are only known during runtime. When you want to create IDL from your Smalltalk objects during edit time, typed IDL can not be generated from your Smalltalk classes.

4.4.1. Solution using the Any-type

Taking a first look at this problem the solution seems to be easy. The CORBA IDL supports the Any-type, which can contain *any* type you want. In the Any-type, a TypeCode is included which describes one of the 22 possible primitive CORBA-types (including IORs, see Appendix C). Because of this feature, the Any-type should be perfect to use for every Smalltalk method's in- and output parameter. During runtime, the ORB can determine the in- and output parameter types of methods and convert them to an Any-type corresponding to the Smalltalk-CORBA type mapping (see [OMG 1995a]). Also the ORB should only receive Any-types and convert them into the appropriate Smalltalk type objects.

But, when you are using a typed programming language - such as Java - at the other side of your distributed environment, using the Any-type is not a good solution. Every time you provide a method with a parameter, you have to typecast the parameter-types to the Any-type. Consequently your Java code will look like this:

```
// IDL for Calculator which is implemented in Smalltalk
//    module Smalltalk{
//          interface Calculator{
//                any sum(any,any); //According to Smalltalk's nature,
                                           use the Any-type
//                ...
//          };
//    };

public void anOperation{
      Calculator aCalculator=new Calculator();
      int a=3;
      int b=5;
      int c=aCalculator.sum(
            new CORBA.Any().from_long(a),
            new CORBA.Any().from_long(b)
            ).to_long;
}
```

When building your application according to Java's language nature, you wish the source code would look like this:

```
// IDL for Calculator which is implemented in Smalltalk
//    module Smalltalk{
//          interface Calculator{
//                long sum(long,long);
//                ...
//          };
//    };

public void anOperation{
      Calculator aCalculator=new Calculator();
      int a=3;
      int b=5;
      int c=aCalculator.sum(a,b);
}
```

In this last case the Smalltalk methods should be typed because the interface is described in this way. In other words: The type of the input- and output parameters of Smalltalk methods must be known before runtime.

4.4.2. Solution using VisualAge's Public Interface Editor

In VisualAge, the Public Interface Editor can be used to specify the interface for every object. Using this feature, IDL can be generate from this Public Interface Editor. The implementation of this VisualAge interface to CORBA IDL converter is described in paragraph 6.3.

4.5. Smalltalk class methods

An important disadvantage of OMG IDL (and also the VisualAge Public Interface Editor) is that it cannot specify the class methods of a class (in Java, static methods). This is due to the fact that IDL considers a class not as an object but as a distinct phenomena which is only used to describe the behaviour of an object implementation which is always an instance of a class. This description of behaviour, or protocol description, is an interface in IDL.

In Smalltalk, everything is an object. Classes are also objects, which are instances of an abstract class called *Metaclass.* So class methods in Smalltalk are actually instance methods of an instance of Metaclass.

When you want to distribute the Smalltalk class methods (or Java static methods) of an object there are two possibilities:

- **Delegation**. A method should be implemented as an instance method of the object, which delegates a message to the object's class (*^self class theMethod*).
- An separate IDL interface for the **class protocol**. The class protocol of the object can be described using a separate IDL interface, see Example 4-1.

```
module ClassProtocols{
	interface CalculatorClass{
	//Protocol description for the Calculator class methods
		Calculator new(); //Constructor method
	};
};
```

Example 4-1 Class protocol description

5. Commercial implementations of CORBA ORBs for Smalltalk and Java

5.1. Smalltalk

Monday, the 8th of September I went to the IBM international training centre, la Hulpe, Belgium, for a training in the IBM Component Broker. The IBM Component Broker is the solution of IBM for distributed application environments. It is the succeeding generation of IBM's DSOM (Distributed Systems Operations & Management) product, which was suspended when the Component Broker was accounced. The IBM Component Broker is more than just a CORBA compliant ORB. It provides extensive features to access databases and supports many CORBA services. The product is very attractive for ELC Object Technology because of its full integration into the VisualAge development environment.

But, The IBM Component Broker for Smalltalk will only be completely available the second half of 1998 so I had to concentrate on other Smalltalk CORBA implementations for the moment.

Two other options for a VisualAge for Smalltalk CORBA implementation were still open. The GEMStone ORB that demands a total GEMStone development platform (very expensive), and IONA's Orbix.

I decided to research IONA's Orbix because:

- It is a proven implementation of CORBA (also for many other languages and platforms)
- It is completely programmed in VisualAge for Smalltalk
- It is not so expensive

5.2. Java

To test a Java implementation of a CORBA compliant ORB I chose Visigenic's VisiBroker. There are several other Java implementations of CORBA available but VisiBroker turned out to be a very consequent CORBA 2.0 compliant implementation which uses the IIOP protocol not only for Inter ORB communication but also inside its ORB to communicate between client- and a server implementations both subscribed to VisiBroker. Netscape 4.0 now has integrated Visigenic's VisiBroker into the browser to provide applet's CORBA support.

I decided to research Visigenic's VisiBroker because:

- It is a proven implementation (Netscape 4.0 has integrated it into their browser)
- It is completely programmed in Java (1.1)
- It is freely available for evaluation purposes

5.3. IONA's ORBIX for VisualAge for Smalltalk

5.3.1. Orbix components

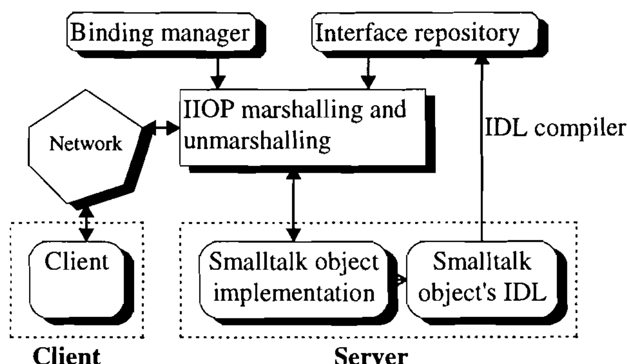


Figure 5-1 Orbix components

Figure 5-1 shows the different components of the Orbix CORBA implementation:

- **IIOP marshalling and unmarshalling**. This component marshals the different Smalltalk types into an IIOP byte stream conform the presentation layer of the OSI model, see paragraph 3.3.2.
- **Interface repository and binding manager**. The interface repository contains all the information about the object's interfaces. The binding manager, which is not a standard CORBA component specified by OMG, maps IDL types to Smalltalk types (interfaces to classes, operations to methods etc.).
- **The IDL compiler** compiles IDL, which *can* be generated from a Smalltalk object implementation (only any types), into the interface repository.

5.3.2. ORB connection manager

The ORB connection manager takes care of the TCP/IP communication. It awaits network events of incoming messages and then sets up a TCP/IP socket connection with the requesting party. This way a separate communication process for every client can be achieved. To limit the amount of opened connections, the connection manager ages the physical connections every time a certain heartbeat interval expires. The connection's status degrades from #connected, #unknown, #inDoubt to #unresponsive. Thus, logical connections always remain but physical connections are discarded when they have not been used for a certain time.

5.3.3. Orbix's CORBA services

Orbix for VisualAge for Smalltalk supports three standard CORBA services:

- **Naming service** for binding object implementations to a static name.
- **Life cycle service** for creating, moving, deleting and destroying object implementations.
- **Event service** for sending asynchronous events to other object implementations (push- and pull model).

For more information about Orbix for Smalltalk and it's services see: [ION 1997].

5.4. Visigenic's VisiBroker for Java

The first step in creating an application with VisiBroker is to specify all of the objects' interfaces using the OMG's Interface Definition language (IDL). The IDL mappings for the Java language, as implemented by the VisiBroker, are summarised in the VisiBroker for Java Reference Manual (see [VIS 1997a]). The interface specification is used by the VisiBroker idl2java compiler to generate stub source code for the client application and skeleton source code for the object implementation. The stub source code is used by the client application for all method invocations. The skeleton source code, along with regular source code, is used to create the server that implements the object. The source code for the client and object, once completed, is used as input to the Java compiler to produce a Java application and an object server. The things described above are shown in Figure 5-2.

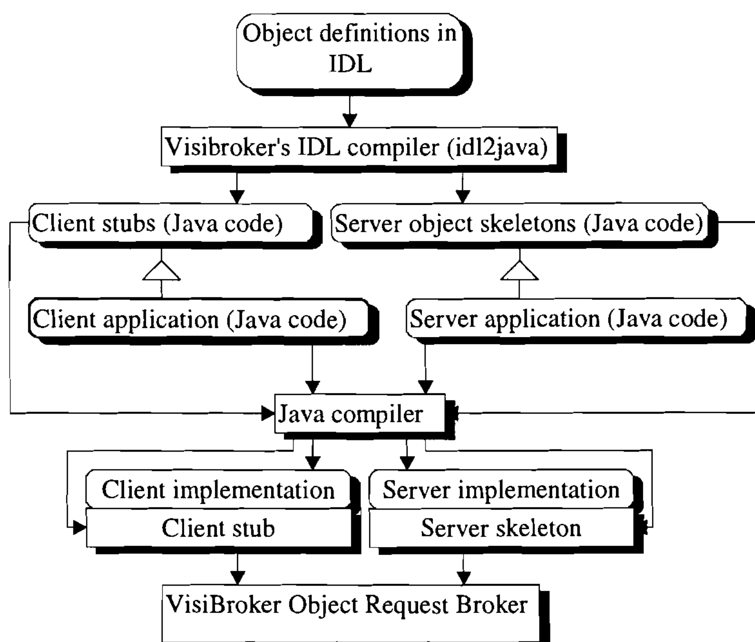


Figure 5-2 Creating a distributed application using VisiBroker for Java.

5.4.1. VisiBroker's features

In addition to providing the features defined in the CORBA specification, VisiBroker offers enhancements that increase application performance and reliability.

Fault Tolerance
VisiBroker can determine if the connection between your client application and an object server has been lost, due to a server crash or network failure. When a failure is detected, an attempt is made to restart the server or to connect your client to a suitable server on a different host. The connections are managed by the OSAgent.

IIOP GateKeeper
The IIOP GateKeeper allows Java clients and servers running within any Java 1.0+ compatible browser to be transparently available to other CORBA objects, even when an applet firewall is in place. The GateKeeper can be used simultaneously as an HTTP daemon. This eliminates the requirement for a separate HTTP server during the

application development phase. The GateKeeper, which 'tunnels' the IIOP protocol needed by applets is shown in Figure 5-3.

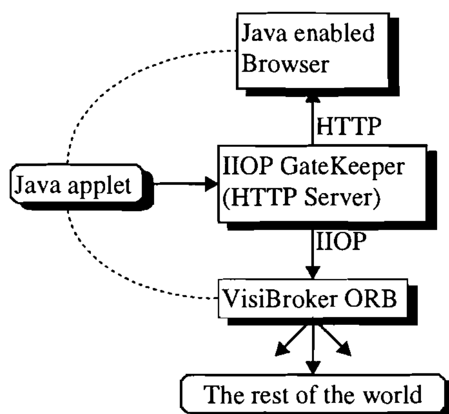


Figure 5-3 The IIOP Gatekeeper

Optimised Binding
When your application binds to an object, VisiBroker can select and establish the most efficient communication mechanism. Depending on the platform and the location of the requested object, the bind may be established through a local pointer reference or a remote TCP/IP connection.

Web Naming
Web Naming allows you to associate Uniform Resource Locators (URLs) with objects, allowing an object reference to be obtained by specifying an URL.

Defining interfaces without IDL
VisiBroker's java2iiop utility allows you to use the Java language to define interfaces, instead of using IDL. You can use the java2iiop utility if you have existing Java code that you wish to adapt to use distributed objects or if you do not have the time to learn IDL. The java2iiop utility generates the necessary container classes, client stubs, and server skeletons from Java code.

Enhanced thread and connection management
Connection management is dependent on what thread policy the user selects. VisiBroker provides two thread policies to choose from: thread-per-session or thread pooling (limited number of threads). The user selects the thread policy and then VisiBroker automatically selects the most efficient way to manage connections between client applications and servers, see [VIS 1997b].

IDL to Java mapping
CORBA-based products are used in a multi-platform world. Distributed objects are developed using a variety of platforms and programming languages. Programming languages define one-to-one relationships from IDL constructs to programming constructs. VisiBroker for Java conforms to the OMG IDL/Java Language Mapping Specification [OMG 1997b] In this mapping, for each IDL construct the corresponding Java construct is described.

For more information about the mapping specification, refer to The OMG IDL/Java Language Mapping Specification, available from Visigenic's web site at <*http://www.visigenic.com*> or [OMG 1997b].

5.4.2. Using VisiBroker to contact objects outside the ORB.

To contact objects outside the VisiBroker's ORB you have to work with IOR's that are configured with the appropriate host and port number. As can be seen in Figure 3-2, the IOR is build out of multiple profiles, which

contain an ObjectLocator (or ProfileBody). The ObjectLocator is the key-part of the IOR because it contains the following important information:

- The host where the object is located (coded as a dotted decimal address like '194.165.88.49').
- The port the object can be accessed on (a TPC/IP port number).
- The ObjectKey which denotes (in an opaque way) the desired object .

To assemble an IOR you could build an IOR builder, which constructs an IOR out of its sub-components (TaggedProfiles, ObjectLocators and ObjectKeys). But, in VisiBroker these components already exist in private object space in the form of separate classes. To build my IOR I decided to decompile the VisiBroker's IOR class together with its sub-components to use it for building my own IORs. Doing this resulted in the following code to build an IOR in the VisiBroker:

```
// Example Java source code to build an IOR

import pomoco.GIOP; //The VisiBroker GIOP package

public IOR buildIOR(String repositoryId,String interfaceName,
            String objectName,String host,int port){
      ObjectKey anObjectKey=new ObjectKey(interfaceName,objectName);
      ObjectLocator anObjectLocator=new
            ObjectLocator(host,port,anObjectKey.asOctetSequence());
      IOR anIOR=new IOR(repositoryId,anObjectLocator);
      return(anIOR);
}
```

Example 5-1 Java source to build an IOR

The ObjectKey class I implemented myself because of the fact that this ObjectKey should be an ObjectKey conform the non-VisiBroker ORB (Smalltalk). You are completely free how to implement the ObjectKey as long as it is encapsulated as a CDR-OctetSequence [OMG 1995a, chapter 12]. The 'asOctetSequence' operation returns a VisiBroker OctetSequence class, which can be used to build the ObjectLocator. For example the ObjectKey for objects subscribed to the Orbix Smalltalk ORB have an ObjectKey which is a 'long' representing the OID (Object ID), a unique number for every object subscribed to the Orbix ORB.

The IOR that is returned from the example above, can be converted to a *CORBA.Object* by sending the message *to_Object()* to the IOR. This CORBA.Object can be used transparently as an object that has the interface described by the CORBA IDL.

5.5. The Performance of a Smalltalk server and Java client

For testing the performance of distributed applications using CORBA I tested a Smalltalk server (IONA's Orbix) with a Java client (Visigenic's VisiBroker). Doing this I used an object with the following interface description:

```
module Performance{
      typedef sequence <octet> sequenceOctet;
      interface ByteAcceptor{
            boolean acceptByte(in octet anOctet);
            boolean acceptBytes(in sequenceOctet aSequence);
      };
};
```

Two tests were performed:

- Measuring the time of a variable number of method invocations (using the 'acceptByte' operation).
- Measuring the time for one method invocation with variable input data (using the 'acceptBytes' operation).

5.5.1. Measuring variable method invocations

For measuring the time of a variable number of method invocations, I wrote a Java client program that invokes the method and calculates the delta time at the moment this method returns true. To measure only the invocation time I implemented the 'acceptByte' routine in Smalltalk as simple as possible:

```
acceptByte: anOctet
      ^true
```

Some problems I had to expect measuring this performance were:

- The operating system, Windows 95, can interrupt the method invocations which makes the result not exactly linear.
- The testing will be done locally. When doing this test at two separate physical locations, network overhead will also introduce unexpected results.
- However the source code to achieve the measurement is kept as simple as possible, it will add its unavoidable time consuming part to the result.

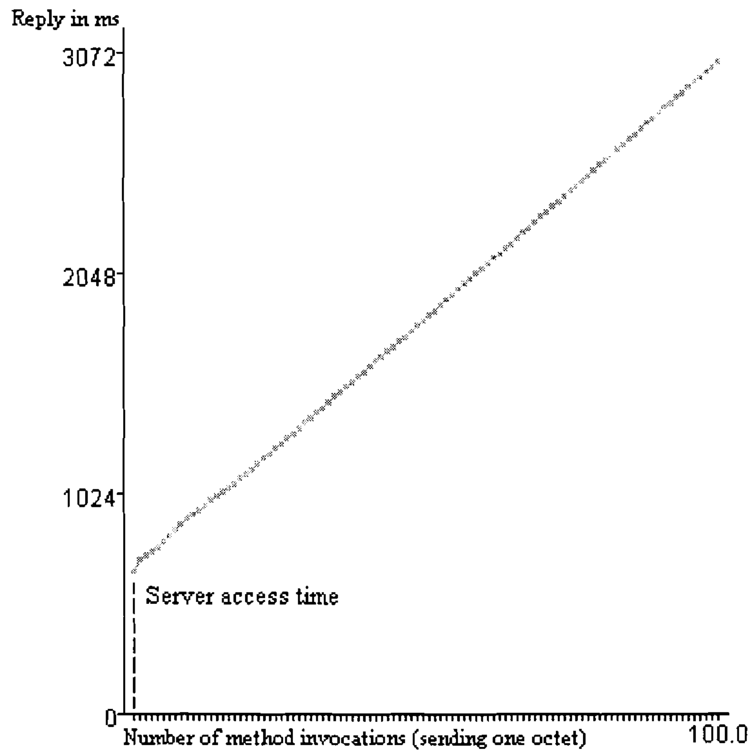


Figure 5-4 Method invocations against time

The initial offset of the linear line can be explained because Orbix sets up its connection the moment it receives an invocation. The measurement's points have been measured separately from each other, this means, the Orbix server shuts down for every measurement point and has to be restarted for a new series of method invocations.

Four measurements where done for one number of method invocations. This way, the average of those four measurements cancels the operating system interventions (like random swapping of system resources). The graph of Figure 5-4 shows the measured values of Table 5-1 and interpolates (linear) the intermediate values. The complete measurements are shown in Appendix J.

Number of method invocations	Average time in ms (from four measures)
1	660
2	715
5	770
10	908
20	1113
50	1828
100	3038

Table 5-1 Measured values for variable method invocations

5.5.2. Measuring variable parameters in one invocation

For measuring the time of a variable number of input octets in one method invocation, I wrote a Java client program that invokes the method with a certain number of input octets, and calculates the delta time at the moment this method returns true. To measure only the invocation time I implemented the 'acceptBytes' routine in Smalltalk as simple as possible:

```
acceptBytes: anOctetSequence
      ^true
```

The problems I had to expect measuring this performance are the same as in the previous experiment.

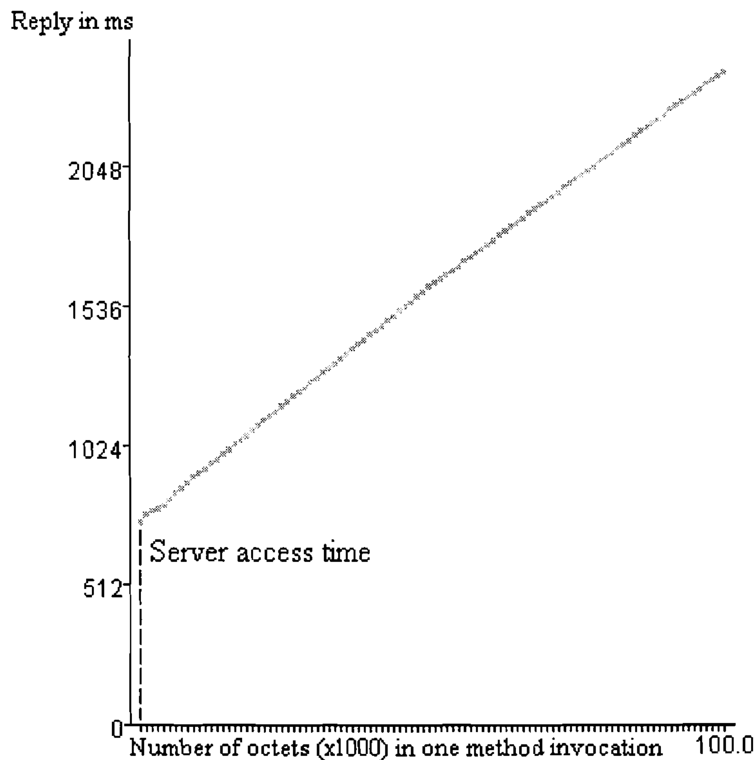


Figure 5-5 Number of input octets for one method invocation against time

The initial offset of the linear line can be explained because Orbix sets up its connection the moment it receives an invocation. The measurement's points have been measured separately from each other, this means, the Orbix server shuts down for every measurement point and has to be restarted for a new method invocation.

Four measurements where done for one method invocation. This way, the average of those four measurements cancels the operating system interventions (like random swapping of system resources). The graph of Figure 5-5 shows the measured values of Table 5-2 and interpolates (linear) the intermediate values. See also Appendix J.

Number of input octets	Average time in ms (from four measures)
1000	742
2000	770
5000	810
10.000	908
20.000	1085
50.000	1610
100.000	2403

Table 5-2 Measured values for variable input octets in one method invocation

6. Implementation of a CORBA framework

Before I started research on Orbix I implemented the IIOP protocol in Smalltalk to communicate with Visigenic's VisiBroker. Doing this gave me a better look upon the internal funcionallity of a CORBA ORB. By extending this IIOP protocol implementation with an interface repository and an object server I implemented a Smalltalk server ORB. Features of this server ORB, like generating IDL from Smalltalk code, were later added to the Orbix CORBA implementation to improve the integration with the VisualAge environment.

6.1. Implementation of the IIOP Protocol in Smalltalk.

To test the IIOP protocol I implemented this protocol in Smalltalk for sending Smalltalk CORBAType instances (which are build out of Smalltalk objects). First, these CORBATypes are described after which the implementation of the IIOP protocol using these CORBATypes is described.

6.1.1. CORBAType classes

To type Smalltalk I decided to develop classes that bridge the type conversion between Smalltalk and IDL/Java. These classes are called the CORBAType classes. Every Smalltalk class can be converted into a CORBAType class, which has the following properties:

- It can be marshalled into an IIOP byte stream (conform the type's alignment, see Appendix B).
- It can be converted into an IDL string for generating IDL source code.
- It can be converted into a Java string for generating Java 'helper' classes.

CORBAType is an abstract class, which describes the behaviour of its subclasses. This behaviour consists of several methods to convert the CORBAType into an IDLString (for generating IDL), a JavaString (for generating Java Helper classes) and an IIOP ByteArray for sending the object across the wire. The design of the CORBAType class with its subclasses is shown in Figure 6-1.

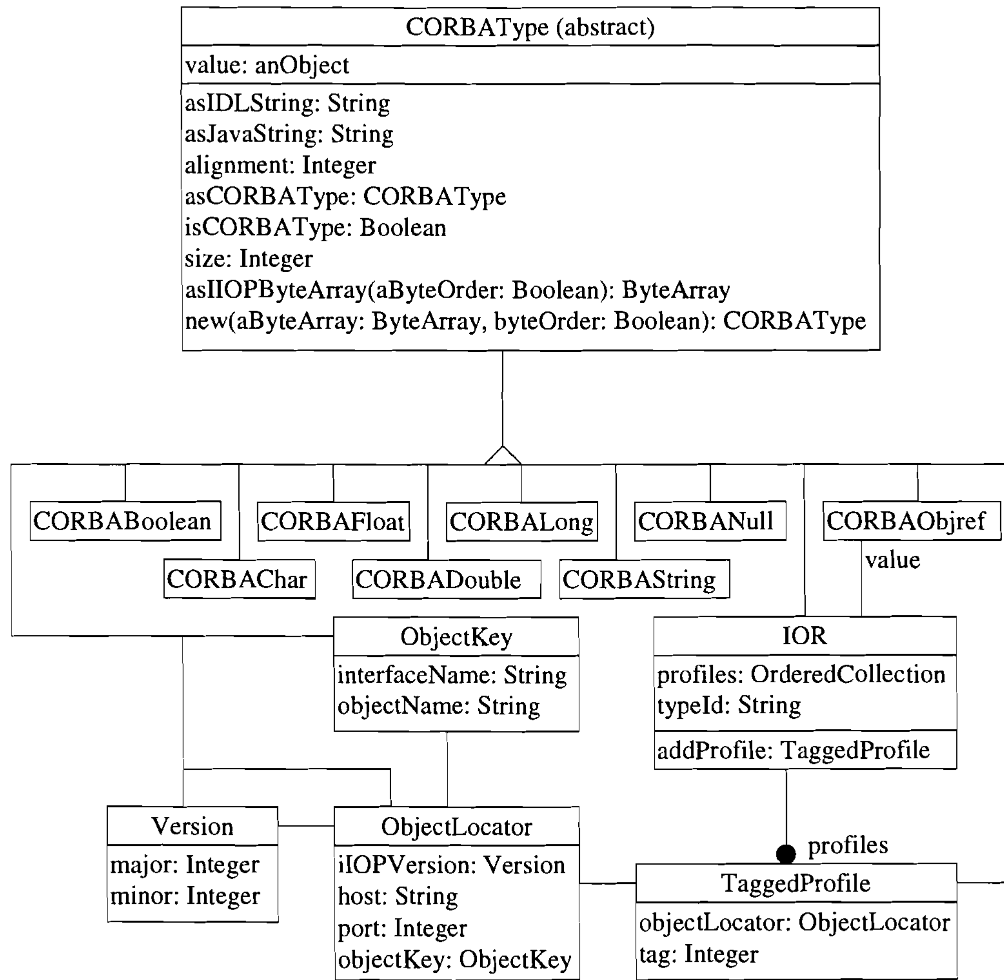


Figure 6-1 Smalltalk CORBAType classes

All the Smalltalk class objects can be converted to a CORBAType class using the method *asCORBAType*. For example, sending this message to a Smalltalk *Integer* class returns a CORBALong class. Every Smalltalk primitive type class was extended with such an 'asCORBAType' method. Also the Smalltalk Object class was extended. The last returns not a CORBAType class but a CORBAType instance of CORBAObjref which value holds an IOR with information about the Smalltalk object. The correspondence between the Smalltalk classes, CORBATypes and IDL/Java types are shown in Table 6-1

Smalltalk Type	Smalltalk CORBAType	IDL Type	Java Type
Boolean	CORBABoolean	boolean	boolean
Character	CORBAChar	char	char
Float	CORBAFloat or CORBADouble	float or double	float or double
Integer	CORBALong	long	int
String	CORBAString	string	java.lang.String
UndefinedObject	CORBANull	Object (undefined IOR)	null
Object (none of the above)	instance of CORBAObjref	Object	CORBA.Object

Table 6-1 Smalltalk type conversions to CORBA and Java

When the asCORBAType method is send to a Smalltalk instance, an instance of CORBAType is returned. For example, the result '1.23 asCORBAType' returns an instance CORBAFloat, which value is a Float containing 1.23. When you send the *asIIOPByteArray* method to this CORBAFloat a ByteArray is returned which can directly be send to a CORBA compliant ORB.

Resuming, the asCORBAType method is implemented in every Smalltalk primitive class as class- and instance method. The method is also implemented as class- and instance method in Object.

Beside this, the asIIOPByteArray method is also implemented as class- and instance method in Object. This method first converts the objects into a CORBAType (using the asCORBAType method) after which the asIIOPByteArray method is performed. This way you can send 'asIIOPByteArray' to every Smalltalk object, which makes it possible to send every Smalltalk object across the wire using IIOP.

6.1.2. IIOP protocol classes

Due to the Object Oriented structure of the IIOP/GIOP specification it was quite easy to implement this protocol in Smalltalk. The protocol can be implemented using the following object model:

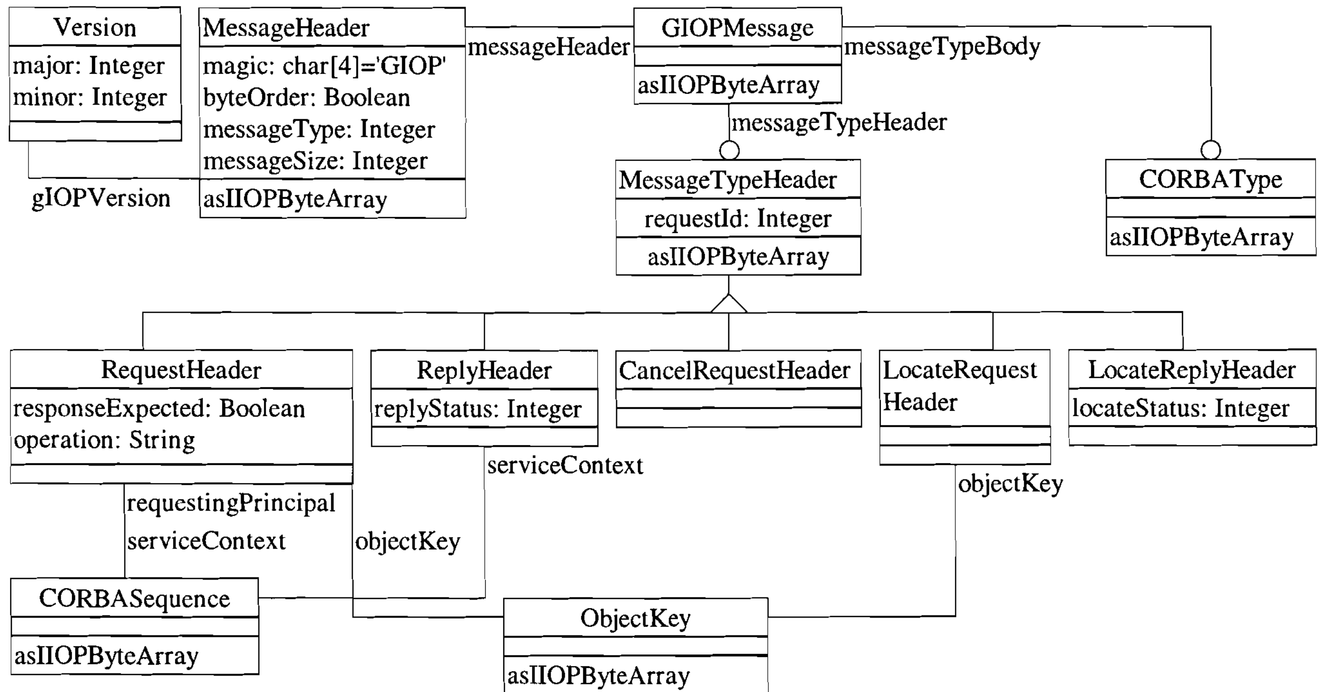


Figure 6-2 Object Model of the IIOP implementation

When taking a close look at Figure 6-2 and comparing it with the GIOP specifications in Appendix D (and [OMG 1995a, chapter 12]), it becomes clear that every Pseudo-IDL specification of the GIOP message components (MessageHeader, RequestHeader etc.) can be assigned to a class. Following closely the OMG specification I implemented the IIOP protocol in this way. I used Smalltalk upper/lower case conventions by writing OMG defined variables like 'request_id' as 'requestId' to achieve a consequent implementation.

GIOPMessage class

The complete GIOPMessage is contained in the GIOPMessage class (Figure 6-2) which has three instances:

- **messageHeader**. This instance contains an instance of the MessageHeader class in which the message header data for the GIOP message is assembled.
- **messageTypeHeader**. This instance can contain an instance of the classes: RequestHeader, ReplyHeader, CancelRequestHeader, LocateRequestHeader or LocateReplyHeader. Although the GIOP message type can also be a CloseConnection message or a MessageError message these message are encoded as an empty messageTypeHeader instance (nil). In this case, the message type follows from the messageType instance in the MessageHeader.
- **messageTypeBody**. This instance can contain an instance of the subclasses of CORBAType. The class CORBAType functions as an abstract class that provides the common behaviour of the classes that can form the GIOP message body. The message body can also contain a ByteArray of 'raw' data. Because ByteArray is a class already implemented in Smalltalk this class cannot be inherited from the MessageTypeBody class. In this case the common behaviour is implemented as an extension to the ByteArray class). When the messageTypeBody contains a 'nil' this denotes that no message body is present in the GIOP message.

The methods of the GIOPMessage class contain methods to port a GIOPMessage from and to a Smalltalk ByteArray, which can be sent or received directly using a TCP/IP connection. The method 'asIIOPByteArray' converts a GIOPMessage into a ByteArray. The 'new: aByteArray' method, which is the overruled Smalltalk construction method, converts a ByteArray into a GIOPMessage.

Converting a GIOPMessage into a ByteArray seems a quite difficult task. The different message types have to be dealt with separately, the byte order has to be correct (big-endian or little-endian) and the byte alignment, described in Appendix B, has to be taken into account.

By implementing an 'asIIOPByteArray' method in every message object, this problem can be solved. However, every 'asIIOPByteArray' method other than the 'asIIOPByteArray' method in the GIOPMessage class and the MessageHeader class has to have an input parameter 'byteOrder' because of the fact that the byte order for the entire GIOP message is determined in the MessageHeader class. Because in every GIOPMessage class there exists a messageHeader instance containing a MessageHeader the byte order is also known in the GIOPMessage class.

When implementing the 'asIIOPByteOrder' method described above in every message object, the GIOPMessage class can send this message to its messageTypeHeader and messageTypeBody instance. Doing this, the GIOPMessage gets transparently the ByteArrays it needs to convert itself into an IIOP ByteArray.

It is also possible to work the other way around. When ByteArray is present (received from a TCP/IP stream), the GIOPMessage can be instantiated using this ByteArray. Again this seems to be a quite difficult task but by implementing this class constructor method in every message object and sending it the convenient ByteArray, the problem can be solved quite easy.

MessageTypeHeader class

The MessageTypeHeader is an abstract class of which its methods are defined to be the subclasses' responsibility. It contains one instance: requestId. This instance is a common instance that is defined in every message type header. It represents a unique number for every outstanding request (which has not yet been replied). The subclasses of the MessageTypeHeader are:

- RequestHeader
- ReplyHeader
- CancelRequestHeader
- LocateRequestHeader

- LocateReplyHeader

According to the GIOP message type header specifications (see Appendix D and [OMG 1995a, chapter 12]) the CloseConnection message header and MessageError message header are missing. This is due to the fact that these two messages do not require a message header. Both message are represented by the Smalltalk UndefinedObject (nil) and are coded in the MessageHeader messageType instance.

MessageTypeHeader subclasses

The **RequestHeader** class is used to represent a GIOP message's request message. It contains instances according to the defined properties of the request message (see Appendix D). The serviceContext and the requestingPrincipal instances are encoded using the CORBASequence class. This CORBASequence class contains an OrderedCollection of CORBAOctet classes which composes the contents of the RequestHeader class. Doing this is called (CDR-)encapsulation [OMG 1995a].

The **ReplyHeader** class is used to represent a GIOP message's reply message. It contains instances according to the defined properties of the reply message (see Appendix D). Like the RequestHeader class there are instances which are (CDR-)encapsulated using the CORBASequence class.

The **CancelRequestHeader** class is used to represent a GIOP message's cancelrequest message. It is a class without instances. The inheritance relation to the abstract MessageTypeHeader class defines the only instance (requestId). The reason this class is implemented is because of the subclass responsibility of the 'asIIOPByteArray' method that is therefore implemented in the CancelRequestHeader class.

The **LocateRequestHeader** class is used to represent a GIOP message's LocateRequest message. It contains one own instance, objectKey, which is encapsulated when sending the 'asIIOPByteArray' method.

The **LocateReplyHeader** class is used to represent a GIOP message's LocateReply message. It contains the instance locateStatus, which determines the contents of the messageTypeBody instance of the GIOPMessage class.

6.2. Implementation of a server ORB in Smalltalk

The basics of the server ORB implementation in Smalltalk lays in three components:

- The Smalltalk typing using the CORBAType (sub)classes.
- The implementation of the IIOP protocol using the GIOPMessage class.
- The implementation of an interface repository.

The CORBAType classes and the implementation of the IIOP protocol are described in paragraph 6.1.1 and 6.1.2. In this paragraph the implementation of the interface repository is described. Together with the CORBAType- and GIOPMessage classes this forms the basics of a complete CORBA server ORB (of coarse without the standard OMG defined CORBA services).

6.2.1. Interface repository

Repository
moduleDefs: OrderedCollection
addModuleDef(aModuleDef: ModuleDef)
resolveInterface(aRepositoryID: String): InterfaceDef

moduleDefs

ModuleDef
name: String
interfaceDefs: OrderedCollection
asIDLSourceCodeForJava: IDLSourceCode
asJavaSourceCodes: OrderedCollection
findInterfaceDef(anInterfaceDef: InterfaceDef): InterfaceDef
allInterfaceDefs: OrderedCollection
addInterfaceDef(anInterfaceDef: InterfaceDef)

interfaceDefs

interfaceDefs

InterfaceDef
name: String
interfaceDefs: OrderedCollection
operationDefs: OrderedCollection
asIDLSourceCodeForJava(aModuleName: String): IDLSourceCode
asJavaSourceCodes(aModuleName: String): OrderedCollection
findInterfaceDef(anInterfaceDef: InterfaceDef): InterfaceDef
findParentInterfaceDef(anInterfaceDef: InterfaceDef): InterfaceDef
allInterfaceDefs: OrderedCollection
addInterfaceDef(anInterfaceDef: InterfaceDef)
addOperationDef(anOperationDef: OperationDef)

SourceCode (abstract)
sourceCode: String
Save(fileName: String)

IDLSourceCode
moduleName: String

JavaSourceCode
className: String

operationDefs

OperationDef
name: String
smalltalkName: Symbol
returnParameter: ParameterDef
parameterDefs: OrderedCollection
asIDLString: String
asIDLStringForJava: String
asJavaOperationString: String
addParameterDef: ParameterDef

parameterDefs

returnParameter

ParameterDef
name: String
type: CORBAType
asIDLString: String
asIDLStringForJava: String

type

CORBAType

Figure 6-3 Object model of the interface repository

The interface repository contains all the interface information about the objects that can be accessed through the ORB.

The Repository object contains the complete interface repository. It consists of modules, which act like a set of interface descriptions. Using the '*resolveInterface*' method returns an interface definition that corresponds to a RepositoryID. A RepositoryID is defined by OMG and looks like, for example: 'IDL:ModuleName/InterfaceName:1.0'. When you specify such a RepositoryID, the Repository returns an interface definition 'InterfaceName' in the module 'ModuleName'.

The **ModuleDef** object can contain several interface definitions. You can lookup an interface using the *findInterfaceDef* method on the ModuleDef. *allInterfaceDefs* returns an OrderedCollection containing all interface definitions defined in the current module.

The **InterfaceDef** object contains the actual interface definition. It defines the operations that can be performed using this interface. Also, an InterfaceDef can contain other interface definitions. This means the current interface definition is a super interface for the in itself specified interface definitions. Like the ModuleDef object it is possible to find interfaces contained in the current InterfaceDef and to get an OrderedCollection with all the specified interface definitions in the current InterfaceDef.

The **OperationDef** object describes an operation definition by specifying its input- and output parameters. An OperationDef has, beside its name, also a smalltalkName. This name corresponds to the actual Smalltalk selector for the method, which is described by the operation definition. The reason for this is that the operationName cannot contain colons and should therefore be expressed in another way. Because IDL does not support operations with the same name (even if the parameters are different!) I chose the solution for conversion presented in paragraph 4.2.1. The Smalltalk Naming applied here is similar to the binding manager of the Orbix CORBA implementation.

The **ParameterDef** object describes the in- and output parameters of an operation definition. The type is described using a CORBAType class (or CORBAObjref instance, see Table 6-1). The name of the parameter is assigned using the Smalltalk operation definition (see the OMG IDL <-> Smalltalk mapping, [OMG 1995a, chapter 19]).

The ModuleDef and InterfaceDef have methods called: *asIDLSourceCodeForJava* and *asJavaSourceCodes*. These methods return an instance of a subclass of SourceCode which contains respectively an IDLSourcesCode or a collection of JavaSourceCode. This SourceCode object can directly be saved to disk (using the *save* method in SourceCode) and compiled with an IDL compiler or Java compiler.

The method asIDLSourceCodeForJava has the postfix 'ForJava' because it takes care of interface/operation names when they look like Java reserved words (such as 'new' and 'do'). In this case these names are converted to RESERVEDWORD<name> ('new' becomes 'RESERVEDWORDnew' and 'do' becomes 'RESERVEDWORDdo').

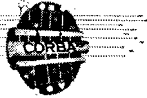
The 'asJavaSourceCodes' method returns a collection of JavaSourceCode (one for every interface). These Java sources are a subclass of the stub generated by the VisiBroker's 'idl2java' compiler. They are not necessary for contacting the Smalltalk ORB but make life more easy because they support:

- **Transparent constructors**. When you instanciate a CORBA object you have to use the '-Helper' class which enables you to 'narrow' a CORBA object obtained from an IOR to the actual object type. In the extra Java subclass you can use a transparent constructor which takes care of this problem using the IOR builder described in paragraph 5.4.2.
- **Garbage collection facility**. When Java garbage collects, it sends the message 'finalize()' to the object before it is killed. In the extra Java subclass the finalize method is overruled to send a message to the Smalltalk ORB that the object is garbage collected after which the ORB can remove it.

The asJavaSourceCodes method uses a class called JavaDefaults. This class contains several constants needed for compiling Java code.

JavaDefaults
ClassMethods: *Names* companyName: String defaultObjectName(interfaceName): String garbageCollectMethodName: String newMethodName: String productName: String ('sORBet') remoteInstanceName: String *Package* corbaPackage: String iorPackage: String objectLocatorPackage: String *Post/Prefix* skeletonPostfix: String skeletonPrefix: String varPostfix: String varPrefix: String *SpecialMethods* finalizeMethod: String

Figure 6-4 JavaDefaults class



6.2.2. The ORB

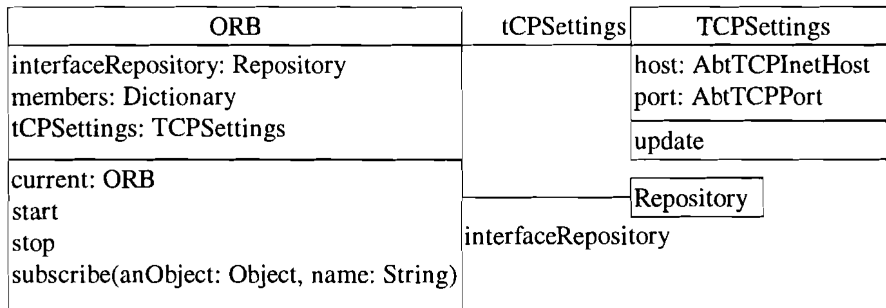


Figure 6-5 The ORB class

In Figure 6-5 the ORB class is shown. It contains the following instances:

- **interfaceRepository**. This instance contains the interface repository in which the interface description of all the possible distributed objects are described.
- **members**. This instance contains a Dictionary in which all the to the ORB subscribed object instances are stored. The ORB can only send messages to these instances. When an instance is created out of a subscribed object (using the transparent constructors in Java), the ORB assigns this instance with a transient name obtained from the ORB to the members dictionary. This transient name is a unique number because it is auto-incremented when it is requested from the ORB.
- **tCPSettings**. This instance contains a TCPSettings object in which all the TCPSettings (host and port) of the ORB are stored. You can *update* the TCPSettings. This will assign the TCP settings with the local host.

The object server can be started and stopped and objects can be subscribed to the ORB. Being subscribed makes the objects visible to the ORB and accessible for remote users.

6.3. Using VisualAge to specify the interface

VisualAge for Smalltalk can support an interface definition for every class using the *Public Interface Editor*. This Public Interface Editor can specify the public attributes (getter- and setter methods), actions (methods) and events for a class. When specifying an interface, every class is supported with a class-method called: *IS_instanceInterfaceSpec.*

When this method is performed on the class it returns an instance of *AbtInterfaceSpec* (see Figure 6-6) which contains all the action-, attribute- and event descriptions specified in the Public Interface Editor. This way it is possible to type every Smalltalk class which makes it possible to convert the interface specification into using other types than only the Any-type. Also the interface repository described in 6.2.1 can be filled with this information to make the object accessible by the ORB. The IDL generated from the *AbtInterfaceSpec* can also be used to generate IDL for Orbix.

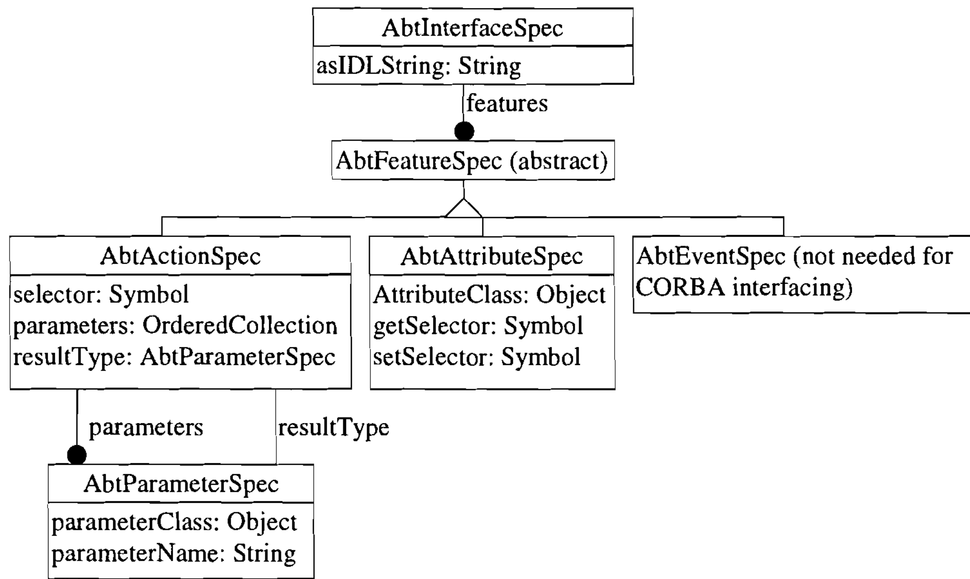


Figure 6-6 The VisualAge AbtInterfaceSpec class

A disadvantage of the VisualAge Public Interface Editor is that it cannot specify the result type of an action. In contrast, the *AbtActionSpec* class, which is used to describe the interface specification of an action actually contains a resultType instance (see Figure 6-6). I solved this problem by extending the Public Interface Editor with the possibility to specify the result type of an action.

7. Conclusions and recommendations

To build distributed Smalltalk/Java applications CORBA turns out to be the solution. A Java ORB, Visigenic's VisiBroker, now is implemented in the Netscape 4.0 browser and many CORBA implementations are now available for a large range of different programming languages. Compared with other distributed systems, this moment CORBA is the absolute winner when object oriented languages are concerned. Microsoft's DCOM has not reached the mature level of CORBA and is not widely supported at the moment. When the time has come that DCOM will be used, CORBA already has specified a bridge to DCOM.

There are many commercial implementations of CORBA available today. However, Smalltalk seems to be a problem in this area. An extensive search for commercial implementations of CORBA for VisualAge for Smalltalk has lead to three products: The IBM Component Broker, The GEMStone ORB and IONA's Orbix. The Component Broker turns out to become *the* VisualAge for Smalltalk CORBA implementation for the future but the product's timeline shows that it will be only available late 1998. For the moment Orbix is a good alternative. Its integration into the VisualAge's development system a quite poor. For this reason I extended Orbix with several features, e.g. generating IDL form the VisualAge's Public Interface Editor.

For Java a larger range of CORBA implementations is available. I chose Visigenic's VisiBroker because it is the most supported Java CORBA implementation at the moment. It is integrated into the Netscape 4.0 browser and implemented completely in Java. Also the intra-ORB communication is achieved with the IIOP protocol which makes the implementation very straightforward.

The performance of the communication between the Smalltalk Orbix ORB and the Java VisiBroker ORB is measured in this report. Many restrictions on this results can be made because of the operation system overhead, network bottlenecks and application overhead. Conform [DOU 1997], CORBA does not ensure any kind of quality of service. It depends on the implementation of the ORB, operating system (which provides the OS systems calls for TCP/IP communication) and network environment what the result of the measurements will be. The results obtained in paragraph 5.5 are just an indication of what can be expected from a normal 150 MHz Pentium, windows 95, environment. When the results from paragraph 5.5 are compared with the results in [DOU 1997] the restrictions of these arguments on the performance become clear.

When Smalltalk logic is distributed using CORBA one major problem arises. Smalltalk is totally untyped while the Interface Description Language is completely typed. This means that when a Smalltalk application is distributed, IDL can not simply be generated from your Smalltalk code. There has to be some way to specify the typing of Smalltalk methods. In this report it is done using the VisualAge's Public Interface Editor. Another disadvantage is the Smalltalk to IDL naming conversion. OMG only specifies an IDL to Smalltalk mapping. When the opposite direction (Smalltalk to IDL) is concerned, many naming collisions appear, see paragraph 4.2.1.

When Java logic is distributed using CORBA the typing problem described above is not the problem. Since IDL and Java are both designed after C++ , the IDL types clearly match the Java types. One disadvantage for pure object oriented programmers (like Smalltalk programmers) is that primitive IDL data types (such as 'long' and 'float') map to non-objects in Java ('int' and 'float'). This makes distribution between Smalltalk and Java not as transparent as it could have been.

In chapter 6, a Smalltalk implementation of a CORBA server ORB is described. When I implemented the IIOP protocol in Smalltalk to test if the VisiBroker used IIOP conform the OMG standard, it became clear that the implementation of the IIOP is the key for a complete server ORB. When primitive data types, and Interoperable Object References (IORs), can be marshalled into an IIOP byte stream, the only things that lack from a CORBA server ORB is an interface repository and a connection manager (for client/server communication on TCP/IP). This can also be seen in Figure 5-1 in which the components of the Orbix CORBA implementation are shown. Of coarse, the OMG defined standard CORBA services are still to be developed.

Appendix A.Common Data Representation (CDR)

CDR is a transfer syntax, mapping from data types defined in OMG IDL to a binary, low-level representation for transfer between agents. CDR has the following features:

- **Variable byte ordering.** Machines with a common byte order may exchange messages without byte swapping. When communicating machines have different byte order, the message originator determines the message byte order, and the receiver is responsible for swapping bytes to match its native ordering. Each GIOP message (and CDR encapsulation) contains a flag that indicates the appropriate byte order. The supported byte orders are: Big-Endian (high order byte first) and Little-Endian (low order byte first).
- **Aligned primitive types.** Primitive OMG IDL data types are aligned on their natural boundaries within GIOP messages, permitting data to be handled efficiently by architectures that enforce data alignment in memory (see Appendix B).
- **Complete OMG IDL Mapping.** CDR describes representations for all OMG IDL data types, including transferable pseudo-objects such as TypeCodes (see Appendix C). Where necessary, CDR defines representations for data types whose representations are undefined or implementation-dependent in the CORBA Core specifications.

Appendix B. Alignment of IDL types

Alignment of IDL primitive datatypes

In order to allow primitive data to be moved into and out of octet streams with instructions specifically designed for those primitive data types, in CDR all primitive data types must be aligned on their natural boundaries. For example the alignment boundary of a primitive datum is equal to the size of the datum in octets. Any primitive of size *n* octets must start at an octet stream index that is a multiple of *n*. In CDR, *n* is one of 1, 2, 4, or 8.
Where necessary, an alignment gap precedes the representation of a primitive datum. The value of octets in alignment gaps is undefined. A gap must be the minimum size necessary to align the following primitive. Table 7-1 gives alignment boundaries for OMG-IDL primitive types. Alignment is defined above as being relative to the beginning of an octet stream. The first octet of the stream is octet index zero (0). Any data type may be stored starting at this index. Such octet streams begin at the start of an GIOP message header and at the beginning of an encapsulation, even if the encapsulation itself is nested in another encapsulation.

Type	Octet alignment
char	1
octet	1
short	2
unsigned short	2
long	4
unsigned long	4
float	4
double	8
boolean	1
enum	4

Table 7-1 Alignment requirements for OMG IDL primitive data types

How the primitive datatypes show in Table 7-1 are configured into their aligned bytes is specified in [OMG 1995a].

Alignment of IDL constructed datatypes

OMG IDL constructed datatypes are built from OMG IDL primitive data types using facilities defined by the OMG IDL language. Constructed type have no alignment restrictions beyond those of their primitive components of which they consist. The constructed datatypes are defined using the IDL primitive datatypes as follows:

- **Struct**. The components of a structure are encoded in their order of their declaration in the structure. Each component is encoded as defined for its data type.
- **Union**. Unions are encoded as the discriminant tag of the type specified in the union declaration, followed by the representation of any selected member, encoded as its type indicates.
- **Array**. Arrays are encoded as the array elements in sequence. As the array length is fixed, no length values are encoded. Each element is encoded as defined for the type of the array. In multidimensional arrays, the elements are ordered so the index of the first dimension varies most slowly, and the index of the last dimension varies most quickly.
- **Sequence**. Sequences are encoded as an unsigned long value, followed by the elements of the sequence. The initial unsigned long contains the number of elements in the sequence. The elements of the sequence are encoded as specified for their type.
- **String**. Strings are encoded as an unsigned long containing the length of the string, followed by the individual characters in the string, encoded according the ISO Latin-1 character set. The length (initial unsigned long) and

string representation include a terminating null character, so that conventional C-string handling library routines (e.g., strcpy) may be used in the encoded message buffer.

- **Enum**. Enum values are encoded as unsigned longs. The numeric values associated with enum identifiers are determined by the order in which the identifiers appear in the enum declaration. The first enum identifier has the numeric value zero (0). Successive enum identifiers are take ascending numeric values, in order of declaration from left to right.

Appendix C. The CORBA Any-type

Any values are encoded as a TypeCode (encoded as described above) followed by the value type parameters followed by the encoded value. The TypeCodes are listed in Table C-1.

TCKind	Integer	Value Type Parameters
tk_null	0	empty – none –
tk_void	1	empty – none –
tk_short	2	empty – none –
tk_long	3	empty – none –
tk_ushort	4	empty – none –
tk_ulong	5	empty – none –
tk_float	6	empty – none –
tk_double	7	empty – none –
tk_boolean	8	empty – none –
tk_char	9	empty – none –
tk_octet	10	empty – none –
tk_any	11	empty – none –
tk_TypeCode	12	empty – none –
tk_Principal	13	empty – none –
tk_objref	14	complex string (repository ID), string(name)
tk_struct	15	complex string (repository ID), string (name), ulong (count) {string (member name), TypeCode (member type)}
tk_union	16	complex string (repository ID), string(name), TypeCode (discriminant type), long (default used), ulong (count) discriminant type (label value), string (member name), TypeCode (member type)}
tk_enum	17	complex string (repository ID), string (name), ulong (count) {string (member name)}
tk_string	18	simple ulong (max length)
tk_sequence	19	complex TypeCode (element type), ulong (max length)
tk_array	20	complex TypeCode (element type), ulong (length)
tk_alias	21	complex string (repository ID), string (name), TypeCode
tk_except	22	complex string (repository ID), string (name), ulong (count) {string (member name), TypeCode (member type)}
– none –	0xffffffff	simple long (indirection)

Table C-1, TypeCode enum values, parameter list types, and parameters

Appendix D.GIOP Message definitions

GIOP Message Header

All GIOP messages begin with the following header, defined in OMG IDL:

```
module GIOP { // Pseudo-IDL
      enum MsgType {
            Request, Reply, CancelRequest,
            LocateRequest, LocateReply,
            CloseConnection, MessageError
      };
      struct MessageHeader {
            char              magic [4];
            Version           GIOP_version;
            boolean           byte_order;
            octet             message_type;
            unsigned long     message_size;
      };
};
```

The message header clearly identifies GIOP messages. It is defined to be byte-ordering independent, since the header itself defines the byte ordering of subsequent message elements. The members of the header are:

- **magic.** The magic identifies GIOP messages. The value of this member is always the four (uppercase) characters "GIOP".
- **GIOP_version.** The GIOP_version contains the version number of the GIOP protocol being used in the message. The current version of GIOP is "1.0".
- **byte_order.** The byte_order indicates the byte ordering used in subsequent elements of the message (including the message_size field). A value of FALSE (0) indicates Big-Endian byte ordering, and TRUE (1) indicates Little-Endian byte ordering.
- **message_type.** The message_type indicates the type of the message, according to Table 3-1. These correspond to enum values of type MsgType. Remember the enum values are of type long.
- **message_size.** The message_size contains the length of the message following the message header, in octets. This count includes any alignment gaps.

In Figure D-1, a typical representation of a GIOP MessageHeader is showed.

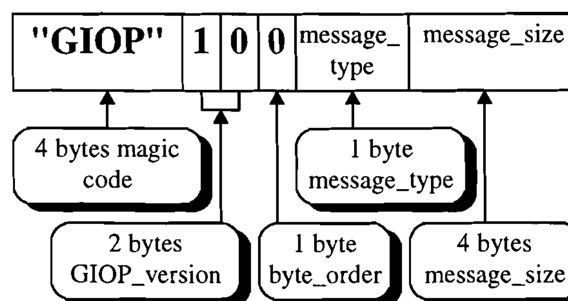


Figure D-1, Representation of the GIOP MessageHeader

Request Header

The request header is specified as follows:

```
module GIOP { // Pseudo-IDL
      struct RequestHeader {
            IOP::ServiceContextList service_context;
            unsigned long                 request_id;
            boolean                       response_expected;
            sequence<octet>               object_key;
            string                        operation;
            Principal                     requesting_principal;
      };
};
```

The members of this RequestHeader struct have the following definitions:

- **service_context.** The service_context contains ORB service data being passed from the client to the server. The context is only defined for a single client/server message. No global environment context is supported.
- **request_id.** The request_id is used to associate reply messages with request messages. The client (requester) is responsible for generating this values.
- **response_expected.** The response_expected variable is set to TRUE if the request is expected to have an reply according to its request_id. The value is FALSE if the operation is defined as one-way, or if the operation is invoked with the DII (dynamically) and the invocation flag includes the INV_NO_RESPONSE flag.
- **object_key.** The object_key uniquely identifies the object which is the target of the invocation. This value is only meaningful to the server and is not interpreted or modified by the client as it is opaque.
- **operation.** Operation contains the name of the operation being invoked. In the case of
- attribute accessors, the names are _get_<attribute> and _set_<attribute>. The case of the operation or attribute name must match the case of the operation name specified in the OMG IDL source for the interface being used.
- **requesting_principal**. The requesting_principal contains a value identifying the requesting principal. It is provided to support the BOA::get_principal operation.

The parameters in the request are encoded in the request body. The request body includes the parameters encoded in the order in which they are declared in the operation's IDL specification (from left to right conform the alignment). After this an optional Context pseudo object is encoded as a CORBA Context object (see [OMG 1995a]).

Reply Header

Reply messages are sent in response to request messages. Replies include inout- and out parameters, operation results, and may include exception values. Reply messages may also provide object location information in the form of an Object Reference.

The reply header is specified as follows:

```
module GIOP { // Pseudo-IDL
      enum ReplyStatusType {
            NO_EXCEPTION,
            USER_EXCEPTION,
            SYSTEM_EXCEPTION,
            LOCATION_FORWARD
      };
      struct ReplyHeader {
            IOP::ServiceContextList       service_context;
            unsigned long                 request_id;
            ReplyStatusType               reply_status;
      };
};
```

The members of the ReplyHeader have the following definitions:

- **service_context.** The service_context contains ORB service data being passed from the server to the client, encoded as described in "GIOP Message Transfer" on page 12-3.
- **request_id.** The request_id is used to give replies to the appropriate requests. It contains the same request_id value as the corresponding request.
- **reply_status.** The reply_status indicates the completion status of the associated request, and also determines part of the reply body contents. If no exception occurred and the operation completed successfully, the value is NO_EXCEPTION and the body contains return values. Otherwise the body contains an exception, or directs the client to resend the request to an object at some other location.

The parameters in the return of the reply are encoded in the reply body. The reply body can include parameters or values in three different forms according to the reply_status variable:

- If the reply_status value is NO_EXCEPTION, the body is encoded as if it were a structure holding first the operation's return value, then any inout and out parameters in the order in which they appear in the operation's IDL definition (from left to right). Notice that this structure can be empty.
- If the reply_status value is USER_EXCEPTION or SYSTEM_EXCEPTION, then the reply body contains the exception that was raised by the operation, encoded as a CORBA exception object (see [OMG 1995a]).
- If the reply_status value is LOCATION_FORWARD, then the body contains an Object Reference. The client ORB is responsible for re-sending the original request to that (different) object denoted by the Object Reference. This resending is transparent to the client program making the request.

CancelRequest Header

CancelRequest messages may be sent from clients to servers. CancelRequest messages notify a server that the client is no longer expecting a reply for a specified pending Request or LocateRequest message.

The cancel request header is defined as follows:

```
module GIOP { // Pseudo-IDL
      struct CancelRequestHeader {
            unsigned long request_id;
      };
};
```

The value of the **request_id** is the same as the value specified in the original Request or LocateRequest message. When a client issues a cancel request message, it serves in an advisory capacity only. The server is not required to acknowledge the cancellation, and may subsequently send the corresponding reply. The client should have no expectation about whether a reply (including an exceptional one) arrives.

LocationRequest Header

LocateRequest messages may be sent from a client to a server to determine the following regarding a specified object reference:

- Whether the object reference is **valid**.
- Whether the current server is **capable** of directly receiving requests for the object reference.
- To what address requests for the object reference should be **forwarded** if the server is not capable to handle the request.

Note that this information is also provided through the Request message, but that some clients might prefer not to support retransmission of potentially large messages that might be implied by a LOCATION_FORWARD status in a Reply message.

The LocateRequest header is defined as follows:

```
module GIOP { // Pseudo-IDL
        struct LocateRequestHeader {
                unsigned long     request_id;
                sequence <octet> object_key;
        };
};
```

The members are defined as follows:

- **request_id** is used to associate *LocateReply* messages with *LocateRequest* ones. The client (requester) is responsible for generating values.
- **object_key** identifies the object being located. In an IIOP context, this value is obtained from the object_key field from the encapsulated ProfileBody in the IIOP profile of the IOR for the target object. This value is only meaningful to the server and is not interpreted or modified by the client.

LocationReply Header

LocateReply messages are sent from servers to clients in response to LocateRequest messages.

The locate reply header is defined as follows:

```
module GIOP { // Pseudo-IDL
      enum LocateStatusType {
            UNKNOWN_OBJECT,
            OBJECT_HERE,
            OBJECT_FORWARD
      };
      struct LocateReplyHeader {
            unsigned long     request_id;
            LocateStatusType  locate_status;
      };
};
```

The members have the following definitions:

- **request_id** is used to associate replies with requests. This member contains the same request_id value as the corresponding *LocateRequest* message.
- **locate_status**. The value of this member is used to determine whether a LocateReply body exists. Values are:
  UNKNOWN_OBJECT The object specified in the corresponding LocateRequest message is unknown to the server; no body exists.
  OBJECT_HERE This server (the originator of the LocateReply message) can directly receive requests for the specified object; no body exists.
  OBJECT_FORWARD A LocateReply body exists.

LocateReply Body
The body is empty unless the LocateStatus value is OBJECT_FORWARD. In this case the body contains an object reference (IOR) that may be used as the target for requests to the object specified in the LocateRequest message.

CloseConnection Header

CloseConnection messages are sent only by servers. They inform clients that the server intends to close the connection and must not be expected to provide further responses. Moreover, clients know that any requests for which they awaiting replies will never be processed, and may safely be reissued (on another connection). The CloseConnection message consists **only** of the GIOP message header, identifying the message type.

MessageError Header

The MessageError message is sent in response to any GIOP message whose version number or message type is unknown to the recipient, or any message is received whose header is not properly formed (e.g., has the wrong magic value). Error handling is context-specific. The MessageError message consists **only** of the GIOP message header, identifying the message type.

IIOP Internet Object References

Individual objects, that are accessible through the IIOP, are denoted using an IIOP profiled Object Reference which is specified in paragraph 3.2. In this profile the TCP/IP properties of the IIOP can be seen because the location of the object is specified using a TCP/IP 'host' and 'port' number.

The host identifies the internet host to which IIOP mapped GIOP messages for the specified object may be sent. In order to promote a very large (internet-wide) scope for the object reference, this will typically be the fully qualified domain name of the host. However according internet standards, the host string may also contain a host address expressed in standard 'dotted decimal' form (e.g., '194.165.88.49').

The port contains the TCP/IP port number (at the specified host) where the receiver is listening for incoming requests.

```
module IIOP { // Pseudo-IDL
      struct Version {
            char major;
            char minor;
      };
      struct ProfileBody { // ObjectLocator
            Version            iiop_version;
            string             host;
            unsigned short     port;
            sequence<octet>    object_key;
      };
};
```

IIOP Profiled Object Reference

Appendix E. Object services

- **Naming service**. The naming service is described separately in paragraph 2.6.1. See also [VOG 1997, Chapter 8].
- **Trading service**. The trading provides a service to bind object implementations by properties instead of by name.
- **Event service**. The event service provides several ways to integrate events into your distributed application
- **Life cycle service**. The Life Cycle Service defines conventions for creating, deleting, copying and moving objects. Because CORBA-based environments support distributed objects, life cycle services define services and conventions that allow clients to perform life cycle operations on objects in different locations.
- **Persistent object service**. The Persistent Object Service (POS) provides a set of common interfaces to the mechanisms used for retaining and managing the persistent state of objects.
- **Transaction service**. The Transaction Service supports multiple transaction models to perform transactions on objects.
- **Concurrency control service**. The Concurrency Control Service enables multiple clients to co-ordinate their access to shared resources. Co-ordinating access to a resource means that when multiple, concurrent clients access a single resource, any conflicting actions by the clients are reconciled so that the resource remains in a consistent state.
- **Relationship service**. The Relationship Service allows entities and relationships to be explicitly represented. Entities are represented as CORBA objects. The service defines two new kinds of objects: relationships and roles. A role represents a CORBA object in a relationship. The Relationship interface can be extended to add relationship-specific attributes and operations. In addition, relationships of arbitrary degree can be defined. Similarly, the Role interface can be extended to add role-specific attributes and operations.
- **Externalisation service**. The Externalisation Service defines protocols and conventions for externalising and internalising objects. Externalising an object is to record the object state in a stream of data (in memory, on a disk file, across the network, and so forth) and then be internalised into a new object in the same or a different process. The externalised object can exist for arbitrary amounts of time, be transported by means outside of the ORB, and be internalised in a different, disconnected ORB. For portability, clients can request that externalised data be stored in a file whose format is defined with the Externalisation Service Specification.
- **Query service**. The purpose of the Query Service is to allow users and objects to invoke queries on collections of other objects. The queries are declarative statements with predicates and include the ability to specify values of attributes; to invoke arbitrary operations; and to invoke other Object Services.
- **Licensing service**. The Licensing Service provides a mechanism for producers to control the use of their intellectual property. Producers can implement the Licensing Service according to their own needs, and the needs of their customers, because the Licensing Service does not impose it own business policies or practices.
- **Property service**. Provides the ability to dynamically associate named values with objects outside the static IDL-type system. Defines operations to create and manipulate sets of name-value pairs or name-value-mode tuples. The names are simple OMG IDL strings. The values are OMG IDL Anys. The use of type any is significant in that it allows a property service implementation to deal with any value that can be represented in the OMG IDL-type system (see Appendix C). The modes are similar to those defined in the Interface Repository AttributeDef interface.
- **Time service**. Enables the user to obtain current time together with an error estimate associated with it.
- **Security service**. The security service provides functionality that restricts certain users to access an object implementation.

Appendix F. CORBA Object Request Broker implementations

- **CHORUS/COOL**. The CHORUS/COOL ORB by Chorus Systems is a CORBA 2.0 compliant Object Request Broker for Distributed real-time Embedded Systems. It is optimised for the CHORUS componentized operating system technology, but it also supports a variety of popular operating systems like SunOS, Linux, Windows NT/95. The CHORUS IDL Compiler (CHIC) generates standard C++ code.
- **Corbus**. Corbus by BBN Corporation is a CORBA 2.0 compliant distributed object-oriented system, with support for multithreaded servers which provide scaleable object location and controlled sharing of services. It provides C, C++ and Common LISP (non-standard) language bindings.
- **DAIS**. DAIS by ICL is a set of CORBA based software tools to create and run a distributed application. The DAIS Run-time Libraries contain a CORBA conformant Object Request Broker (ORB). The DAIS ORB is distributed, resides within all client and server modules, is scalable on the whole network and avoids a single point of failure. DAIS supports the following languages: C, C++, Java, Cobol (in development) and Eiffel (in development).
- **Distributed Smalltalk (DST)**. Distributed Smalltalk by ParcPlace-Digitalk, Inc. (formerly known as HP Distributed Smalltalk) supports the full OMG CORBA 2.0 specifications and the following CORBA services: naming, event, lifecycle, transaction and concurrency. Advanced tools like a remote object debugger, a Smalltalk to IDL generator and an ORB monitor are also included.
- **Java IDL**. The Java IDL by Sun Microsystems allows you to define remote interfaces in the IDL interface definition language. These IDL definitions can then be compiled with the idl2java stub generator tool to generate Java interface definitions and Java client and server stubs. Java IDL is currently available on Win32/x86 and Solaris/SPARC. Jorba
- **ObjectBroker**. ObjectBroker by BEA Systems, Inc. (formerly called DEC ObjectBroker) is a CORBA 2.0 Object Request Broker with full CORBA compliant C++ language bindings. It allows unmodified CORBA objects to be accessed via OLE Automation, CORBA custom interfaces, and OLE custom interfaces. It also supports DCE'S Generic Security Services API (GSSAPI), which allows the use of both DCE-based security (kerberos) and other third-party authentication packages.
- **OmniBroker**. OmniBroker by Object-Oriented Concepts, Inc. is a CORBA 2.0 compliant ORB with complete C++ language mapping and IIOP as native protocol. It supports the Dynamic Skeleton Interface and comes with a COS compliant Naming Service.
- **ORB Plus**. ORB Plus by Hewlett Packard is a fully threaded implementation of the CORBA 2.0 specification and includes the following CORBAservices: Events, Naming, and Lifecycle. Developers in the HP-UX environment have at their disposal the additional capability of the DCE CIOP (Distributed Computing Environment Common Inter-ORB Protocol), which can be used as an alternative to the CORBA 2.0 IIOP. A special transport abstraction layer, part of the ORB Plus architecture, allows the DCE CIOP and the COBRA 2.0 IIOP to be used simultaneously.
- **Orbix**. Orbix from IONA Technologies is a full and complete implementation of CORBA. Orbix runs on more than 20 operating systems with seamless interworking guaranteed across all supported platforms. It supports C++, Ada95 and Smalltalk. Orbix Programming Guide gives a complete introduction in programming with Orbix using C++. Orbix for Windows closely integrates Microsoft OLE and ActiveX technology with CORBA to provide interworking between the Component Object Model (COM) and CORBA.
- **OrbixWeb**. OrbixWeb is a full Java implementation of Orbix . It has been specially modularised and optimised for operation over large networks.
- **PowerBroker CORBAplus**. PowerBroker CORBAplus by Expersoft is a comprehensive implementation of the CORBA 2.0 specification. It includes asynchronous requests, multi-threaded support and also delivers visual tools for editing the Interface Repository. PowerBroker CORBAplus is currently available for Windows 95/NT, Solaris, HP-UX and AIX. PowerBroker CORBAplus for OLE automatically converts CORBA interfaces into OLE Automation interfaces for inclusion into Windows OLE applications. This implementation meets the OMG COM/CORBA Mapping specification, automating the interaction between OLE clients and CORBA objects. CORBAplus for OLE also gives Visual Basic programmers direct, transparent access to CORBA objects and services, providing Windows desktop clients with unprecedented levels of interoperability and flexibility.

PowerBroker CORBAplus for Java is a CORBA 2.0-compliant Object Request Broker with Java to IDL language mapping support.

- **SOMobjects**. SOMobjects form the basis for IBM's implementation of CORBA. SOM provides an object-structured protocol that allows applications to access and use objects, regardless of the programming language in which they are created (Derived class implementers can use different languages from those used by base class implementers). The SOMobjects Toolkit is available for AIX, OS/2 and Windows 3.1/95/NT. IBM Distributed Smalltalk by IBM Corporation extends VisualAge for Smalltalk to support distribution of objects on different computers on a network using IBM's Distributed System Object Model (DSOM). These objects can send standard Smalltalk messages to one another, regardless of their physical location. They can also freely send other Smalltalk objects as arguments, and receive objects as results. The different parts of an application can be located on any computer in the network that is running IBM Distributed Smalltalk.
- **VisiBroker.** The VisiBroker family features an agent based, multi-threaded architecture with automatic configuration and smart binding. It also provides load balancing and high availability, enabling easy object migration and replication. VisiBroker for C++ by Visigenic Software, Inc. (formerly called ORBeline) is a CORBA 2.0 Object Request Broker. A key benefit of Visigenic is that its inter-ORB communication is implemented using the IIOP protocol. VisiBroker for Java by Visigenic Software, Inc. (formerly called Black Widow) is a CORBA 2.0 Object Request Broker written completely in Java with also an intern implementation of the IIOP protocol. It consists of a development and a run-time component.

Appendix G. Source code, CORBAType classes

CorbaType

```
Object subclass: #CORBAType
  instanceVariableNames: 'value '
  classVariableNames: ''
  poolDictionaries: ''
```

CORBAType public class methods

```
alignInteger: anInteger
        |result|
        result:=anInteger-1.
        [(result\\self alignment)=0] whileFalse:[result:=result+1].
        ^result+1.

corbaTypeDictionary
        ^(Dictionary new
                at: #void put: CORBAVoid;
                at: #long put: CORBALong;
                at: #string put: CORBAString;
                at: #octet put: CORBAOctet;
                at: #short put: CORBAShort;
                at: #Object put: CORBAObject;
                at: #float put: CORBAFloat;
                yourself)

fromString: aString
        |aCORBATypeClass aModuleName anInterfaceName|
        aCORBATypeClass:=self corbaTypeDictionary at: aString asSymbol
        ifAbsent:[
                ^CORBAObjref fromString: aString
        ].
        ^aCORBATypeClass new.
```

CORBAType public methods

```
alignInteger: anInteger
        |result|
        result:=anInteger-1.
        [(result\\self alignment)=0] whileFalse:[result:=result+1].
        ^result+1.

alignment
        ^1

asCORBAType
        ^self

copy
        ^(self class new value: self value;yourself)

isCORBAType
        ^true

printIDLString
        |aString|
        aString:=self class printString.
        aString:=aString copyFrom:6 to: aString size.
        aString at:1 put: (aString at:1) asLowercase.
        ^aString
```

```
size
        ^self error: (self printString,' should implement #size').

value
        ^value

value: anObject
        value:=anObject
```

CORBABoolean

```
CORBAType subclass: #CORBABoolean
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
```

CORBABoolean public class methods

```
asIDLString
        ^'boolean'

asJavaString
        ^'boolean'

new: aByteArray byteOrder: aByteOrder
        ((aByteArray at:1)=1) ifTrue:[
                ^self new
                        value: true;
                        yourself.
        ]
        ifFalse:[
                ^self new
                        value: false;
                        yourself
        ].
```

CORBABoolean public methods

```
alignment
        ^1

asIIOPByteArray: aByteOrder
        self value ifTrue:[^#[1]]
        ifFalse:[^#[0]].

size
        ^1
```

CORBAChar

```
CORBAType subclass: #CORBAChar
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
```

CORBAChar class public class methods

```
asIDLString
        ^'char'

asJavaString
        ^'char'

new: aByteArray byteOrder: aByteOrder
```

```
        ^self new value: (aByteArray at:1) asCharacter;yourself
```

CORBAChar public methods

```
alignment
        ^1

asIDLString
        ^'char'

asIIOPByteArray: aByteOrder
        ^(ByteArray new:1) at:1 put: self value value;yourself

size
        ^1
```

CORBAFloat

```
CORBAType subclass: #CORBAFloat
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
```

CORBAFloat class public class methods

```
asIDLString
        ^'float'

asJavaString
        ^'float'

new: aByteArray byteOrder: aByteOrder
        |aCORBAFloat sign exponent fraction|
        aCORBAFloat:=self new.
        sign:= ((aByteArray at: 1)// 128).
        exponent:= (
                (((aByteArray at: 1) - (sign*128))*2) +
                ((aByteArray at: 2)//128)
        ).
        fraction:= (
                (( (aByteArray at:2) - (( (aByteArray at:2) //128 )*128) )*256*256) +
                ((aByteArray at:3)*256) +
                (aByteArray at:4)
        ).
        aCORBAFloat value: (
                ((fraction asFloat / (2 raisedTo: 23)) +1)*
                (2 raisedTo: (exponent - 127)) *
                (-1 raisedTo: sign)
        ).
        ^aCORBAFloat
```

CORBAFloat public methods

```
alignment
        ^4

asIIOPByteArray: byteOrder
        |aCollection fraction anUnsignedFloat exponent sign|
        aCollection:=OrderedCollection new.
        anUnsignedFloat:=self value copy.
        sign:=0. "positive"
        (anUnsignedFloat < 0) ifTrue:[
                anUnsignedFloat:=(0 asFloat -anUnsignedFloat).
                sign:=1. "negative"
        ].
        exponent:=(anUnsignedFloat ln / 2 ln) truncated.
        fraction:=(((anUnsignedFloat / (2 raisedTo: exponent))-1)*(2 raisedTo: 23)) rounded.
```

```
        exponent:=exponent+127.
        aCollection add: (sign*128+(exponent // 2)).
        aCollection add: ((exponent \\ 2)*128+(fraction // 65536)).
        aCollection add: ((fraction - ((fraction // 65536)*65536)) // 256).
        aCollection add: (fraction \\ 256).
        ^aCollection asByteArray

size
        ^4
```

CORBADouble

```
CORBAType subclass: #CORBADouble
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''

CORBADouble public class methods

asIDLString
        ^'double'

asJavaString
        ^'double'

new: aByteArray byteOrder: aByteOrder
        |aCORBADouble sign fraction exponent|
        aCORBADouble:=self new.
        sign:= ((aByteArray at: 1)// 128).
        exponent:= (
                (((aByteArray at: 1) - (sign*128))*16) +
                ((aByteArray at: 2)//16)
        ).
        fraction:= (
                (( (aByteArray at:2) - (( (aByteArray at:2) //16 )*16) )*256*256*256*256*256*256) +
                ((aByteArray at:3)*256*256*256*256*256) +
                ((aByteArray at:4)*256*256*256*256) +
                ((aByteArray at:5)*256*256*256) +
                ((aByteArray at:6)*256*256) +
                ((aByteArray at:7)*256) +
                (aByteArray at:8)
        ).
        aCORBADouble value: (
                ((fraction asFloat / (2 raisedTo: 52)) +1)*
                (2 raisedTo: (exponent - 1023)) *
                (-1 raisedTo: sign)
        ).
        ^aCORBADouble
```

CORBADouble public methods

```
alignment
        ^8

asIIOPByteArray: aByteOrder
        ^(CORBAFloat new value: self value) asIIOPByteArray: aByteOrder

size
        ^8
```

CORBALong

```
CORBAType subclass: #CORBALong
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
```

CORBALong public class methods

```
alignment
        ^4

asIDLString
        ^'long'

asJavaString
        ^'int'

new: aByteArray byteOrder: aByteOrder
        aByteOrder ifFalse:[
                "Big-endian"
                ^self new
                        value: (
                                ((aByteArray at:1) *256*256*256)+
                                ((aByteArray at:2) *256*256)+
                                ((aByteArray at:3) *256)+
                                (aByteArray at:4)
                        )
        ]
        ifTrue:[
                "Little-endian"
                ^self new
                        value: (
                                ((aByteArray at:4) *256*256*256)+
                                ((aByteArray at:3) *256*256)+
                                ((aByteArray at:2) *256)+
                                (aByteArray at:1)
                        )
        ]
```

CORBALoug public methods

```
alignment
        ^self class alignment

asIDLString
        ^'long'

asIIOPByteArray: aByteOrder
        |temp aCollection|
        aByteOrder ifFalse:[
                "Big-endian"
                temp:=self value asInt32.
                aCollection:=OrderedCollection new.
                aCollection add: (temp // (256*256*256)).
                temp:=temp-((aCollection last)*256*256*256).
                aCollection add: (temp//(256*256)).
                temp:=temp-((aCollection last)*256*256).
                aCollection add: (temp//256).
                temp:=temp-((aCollection last)*256).
                aCollection add: temp.
                ^aCollection asByteArray.
        ]
        ifTrue:[
                "Little-endian"
                aCollection:=ByteArray new:4.
                temp:=self value asInt32.
                aCollection at:4 put: (temp // (256*256*256)).
                temp:=temp-((aCollection at:4)*256*256*256).
                aCollection at:3 put: (temp//(256*256)).
                temp:=temp-((aCollection at:3)*256*256).
                aCollection at:2 put: (temp//256).
                temp:=temp-((aCollection at:2)*256).
                aCollection at:1 put: temp.
```

```
                ^aCollection asByteArray
        ].

asJavaString
        ^'int'

asSmalltalkObject: aByteArray byteOrder: aByteOrder
        ^Integer new: aByteArray byteOrder: aByteOrder

size
        ^4

value
        ^value

value: anInteger
        value:=anInteger asInteger
```

CORBANull

```
CORBAType subclass: #CORBANull
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
```

CORBAObjref

```
CORBAType subclass: #CORBAObjref
   instanceVariableNames: 'repositoryId name '
   classVariableNames: ''
   poolDictionaries: ''

CORBAObjref public class methods

asIDLString
        ^'Object'

asJavaString
        ^'CORBA.Object'

for: anInterfaceName moduleName: aModuleName
        |aModuleIndex theModuleDefs|
        theModuleDefs:=ORB current interfaceRepository moduleDefs.
        aModuleIndex:=theModuleDefs findFirst: [:each|each=(ModuleDef new name: aModuleName;yourself)].
        (aModuleIndex=0) ifTrue:[
                "Module not found, return an CORBAObjref class"
                ^self
        ]
        ifFalse:[
                ((theModuleDefs at: aModuleIndex) findInterface: (InterfaceDef new name: anInterfaceName;yourself)) isNil ifTrue:[
                        "interface not in repository, return CORBAObjref class"
                        ^self
                ]
                ifFalse:[
                        "interface in repository, return instance of CORBAObjref"
                        ^self new
                        value: (IOR
                                moduleName: aModuleName
                                interfaceName: anInterfaceName
                                objectName: ORB current newTransientName
                                host: ORB current tCPSettings host
                                port: ORB current tCPSettings port portNumber);
                        yourself
                ]
        ]

fromModuleName: aModuleName interfaceName: anInterfaceName
```

```
        |anIOR|
        anIOR:=IOR
                fromModuleName: aModuleName
                interfaceName: anInterfaceName.
        ^CORBAObjref new
                value: anIOR;
                repositoryId: anIOR typeId;
                name: ((anIOR profiles contents at:1) objectLocator objectKey objectName);
                yourself.

fromString: aString
        |aModuleName anInterfaceName anIOR|
        aModuleName:=aString chopTillColon.
        anInterfaceName:=aString copyFrom: (aModuleName size +3) to: aString size.

        ^self fromModuleName: aModuleName interfaceName: anInterfaceName

new: aByteArray byteOrder: aByteOrder
        "
                Build from CDR OctetSequence encapsulation
        "
        |aCORBAObjref|
        aCORBAObjref:=self new.
        aCORBAObjref value: (IOR new: aByteArray byteOrder: aByteOrder).
        aCORBAObjref repositoryId: aCORBAObjref value typeId.
        aCORBAObjref name: (aCORBAObjref value profiles at:1) objectLocator objectKey objectName.
        ^aCORBAObjref.
```

CORBAObjref public methods

```
alignment
        ^self value alignment

asIDLString
        ^((self value profiles at:1) objectLocator objectKey interfaceName)

asIIOPByteArray: aByteArray
        " the value will contain an IOR"
        ^self value asIIOPByteArray: aByteArray

asJavaString
        |anInterfaceName|
        anInterfaceName:=
                ((self value profiles at:1) objectLocator objectKey interfaceName).
        ^anInterfaceName chopTillColon,'.',
        (anInterfaceName copyFrom: (anInterfaceName chopTillColon size +3) to: anInterfaceName size)

name
        ^name

name: aCORBAString
        name:=aCORBAString

printIDLString
        (self value profiles size=0) ifTrue:[^'objref Smalltalk::UndefinedObject'].
        ^'objref ',(self value profiles at:1) objectLocator objectKey interfaceName

printString
        ^'CORBAObjref value=',self value printString

repositoryId
        ^repositoryId

repositoryId: aCORBAString
        repositoryId:=aCORBAString

size
        ^self value size
```

```
value
        value isNil ifTrue:[self value: IOR new].
        ^value
```

CORBAString

```
CORBAType subclass: #CORBAString
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''

CORBAString public class methods

alignment
        ^4

asIDLString
        ^'string'

asJavaString
        ^'java.lang.String'

new: aByteArray byteOrder: aByteOrder
        |aCORBAString aStringArray length|
        aCORBAString:=self new.
        length:=(CORBALong new: aByteArray byteOrder: aByteOrder) value.
        (length=0) ifTrue:[aCORBAString value:''.^aCORBAString].
        aStringArray:=aByteArray
                copyFrom: 5
                to: (3+length).
        ^self new value: aStringArray asString;yourself.
```

CORBAString public methods

```
alignment
        ^4

asIDLString
        ^'string'

asIIOPByteArray: aByteOrder
        |aCollection|
        (self value size=0) ifTrue:[^#[0 0 0 0]].
        aCollection:=OrderedCollection new
                addAll: ((self value size+1) asIIOPByteArray: aByteOrder);
                addAll: self value asByteArray;
                add: 0;
                yourself.
        ^aCollection asByteArray

asJavaString
        ^'java.lang.String'

asSmalltalkObject: aByteArray byteOrder: aByteOrder
        ^String new: aByteArray byteOrder: aByteOrder

size
        (self value size=0) ifTrue:[^4]
        ifFalse:[
                ^self value size+5.
        ]

value
        value isNil ifTrue:[self value: ''].
        ^value
```

IOR

```
CORBAType subclass: #IOR
  instanceVariableNames: 'typeId profiles '
  classVariableNames: ''
  poolDictionaries: ''!
```

IOR class public class methods

```
fromModuleName: aModuleName interfaceName: anInterfaceName
        ^self
                moduleName: aModuleName
                interfaceName: anInterfaceName
                objectName: ('Default',anInterfaceName)
                host: ORB current tCPSettings host dottedDecimalAddress
                port: ORB current tCPSettings port portNumber

interfaceName: anInterfaceName objectName: anObjectName host: aHost port: aPortNumber
        |aTypeId|
        aTypeId:=('Smalltalk::',anInterfaceName) asRepositoryId.
        ^(self new
                typeId: aTypeId;
                addProfile: (TaggedProfile new
                        tag: 0;
                        objectLocator: (ObjectLocator new
                                host: aHost;
                                port: aPortNumber;
                                objectKey: (ObjectKey new
                                        interfaceName: anInterfaceName;
                                        objectName: anObjectName;
                                        yourself);
                                yourself);
                        yourself);
                yourself)

moduleName: aModuleName interfaceName: anInterfaceName objectName: anObjectName host: aHost port: aPortNumber
        |aTypeId|
        aTypeId:=(aModuleName,'::',anInterfaceName) asRepositoryId.
        ^(self new
                typeId: aTypeId;
                addProfile: (TaggedProfile new
                        tag: 0;
                        objectLocator: (ObjectLocator new
                                host: aHost;
                                port: aPortNumber;
                                objectKey: (ObjectKey new
                                        interfaceName: (aModuleName,'::',anInterfaceName);
                                        objectName: anObjectName;
                                        yourself);
                                yourself);
                        yourself);
                yourself)

new: aByteArray byteOrder: aByteOrder
        "
        Build IOR from encapsulated datastructure: sequence<octet>
                boolean                         byteOrder
                string                          typeId
                sequence<taggedProfile>         profiles
        "
        |anIOR index|
        anIOR:=self new.
        index:=9. "Skip the sequence length and byteOrder"
        ((Boolean new: (aByteArray from:5) byteOrder: false)=aByteOrder) ifFalse:[
                self error: 'ByteOrders are not compliant'
```

```
    ].
    anIOR typeId:
            (CORBAString new: (aByteArray from:index) byteOrder: aByteOrder) value.
    index:=index+anIOR typeId asCORBAType size.
    index align: 4.

    anIOR profiles:
            (CORBASequence new: (aByteArray from:index) byteOrder: aByteOrder type: TaggedProfile) value.
    ^anIOR
```

IOR public methods

```
addProfile: aTaggedProfile
    self profiles add: aTaggedProfile

alignment
    ^4

asIIOPByteArray: aByteOrder
    |aStruct|
    aStruct:=CORBAStruct new
            add: (self typeId asCORBAType);
            add: (CORBASequence new value: self profiles);
            yourself.
    ^aStruct asIIOPByteArray: aByteOrder

asIORString: aByteOrder
    |aString|
    aString:='IOR:'.
    (self asIIOPByteArray: aByteOrder) do:[:each|
            ((each printStringRadix: 16 showRadix: false) size = 1) ifTrue:[
                    aString:=aString, '0' , (each printStringRadix: 16 showRadix: false)
            ]
            ifFalse:[
                    aString:=aString, (each printStringRadix: 16 showRadix: false)
            ].
    ].
    ^aString

printString
    ^'IOR: typeId=', self typeId, ' objectKey=',(self profiles at:1) objectLocator objectKey printString

profiles
    profiles isNil ifTrue:[self profiles: OrderedCollection new].
    ^profiles

profiles: anOrderedCollection
    profiles:=anOrderedCollection

size
    ^(self asIIOPByteArray: false) size

typeId
    ^typeId

typeId: aString
    typeId:=aString
```

TaggedProfile

```
CORBAType subclass: #TaggedProfile
  instanceVariableNames: 'tag objectLocator '
  classVariableNames: ''
  poolDictionaries: ''
```

TaggedProfile public class methods

```
alignInteger: anInteger
	^(anInteger align: 4)

new: aByteArray byteOrder: aByteOrder
	|aTaggedProfile index|
	aTaggedProfile:=self new.
	index:=1.
	aTaggedProfile tag:
		(CORBALong new: aByteArray byteOrder: aByteOrder) value.
	index:=(index+4) align: 4.

	aTaggedProfile objectLocator: (ObjectLocator
		new: (aByteArray from: index)
		byteOrder: aByteOrder
	).

	^aTaggedProfile
```

TaggedProfile public methods

```
alignment
	^4

asCORBAType
	^self

asIIOPByteArray: aByteOrder
	^(CORBAStruct new
		add: (CORBALong new value: self tag);
		add: (self objectLocator);
		yourself) asIIOPByteArray: aByteOrder

objectLocator
	objectLocator isNil ifTrue:[self objectLocator: ObjectLocator new].
	^objectLocator

objectLocator: anObjectLocator
	objectLocator:=anObjectLocator

size
	^(self asIIOPByteArray: false) size

tag
	tag isNil ifTrue:[self tag: 0].
	^tag

tag: anInteger
	tag:=anInteger
```

ObjectLocator

```
CORBAType subclass: #ObjectLocator
  instanceVariableNames: 'host port objectKey iIOPVersion '
  classVariableNames: ''
  poolDictionaries: ''
```

ObjectLocator public class methods

```
new: aByteArray byteOrder: aByteOrder
	|anObjectLocator index|
	anObjectLocator:=self new.
	index:=5. "Ignore the sequence length"

	((CORBABoolean new: (aByteArray from:index) byteOrder: aByteOrder) value = aByteOrder) ifFalse:[
		self error: 'The ByteOrder of the ObjectLocator differs from the MessageHeader'.
```

```
    ].
    index:=index+1.

    anObjectLocator iIOPVersion:
            (Version new: (aByteArray from:index) byteOrder: aByteOrder).
    index:=(index+2) align:4.

    anObjectLocator host:
            (CORBAString new: (aByteArray from: index) byteOrder: aByteOrder) value.
    index:=(index+anObjectLocator host size +5) align: 2.

    anObjectLocator port:
            (CORBAShort new: (aByteArray from: index) byteOrder: aByteOrder) value.
    index:=(index+2) align: 4.

    anObjectLocator objectKey: (ObjectKey
            new: (aByteArray from: index)
            byteOrder: aByteOrder).

    ^anObjectLocator
```

ObjectLocator public methods

```
asIIOPByteArray: aByteOrder
    |aStruct|
    aStruct:=CORBAStruct new
            add: (CORBALong new value:0); "size will be calculated later"
            add: (CORBABoolean new value: aByteOrder);
            add: self iIOPVersion;
            add: (CORBAString new value: self host);
            add: (CORBAShort new value: self port);
            add: self objectKey;
            yourself.
    aStruct value at:1 put: ((aStruct size -4) asCORBAType).
    ^aStruct asIIOPByteArray: aByteOrder

host
    host isNil ifTrue:[self host: ORB current tCPSettings host dottedDecimalAddress].
    ^host

host: aString
    host:=aString

iIOPVersion
    iIOPVersion isNil ifTrue:[self iIOPVersion: Version new].
    ^iIOPVersion

iIOPVersion: aVersion
    iIOPVersion:=aVersion

objectKey
    objectKey isNil ifTrue:[self objectKey: ObjectKey new].
    ^objectKey

objectKey: anObjectKey
    objectKey:=anObjectKey

port
    port isNil ifTrue:[self port: ORB current tCPSettings port portNumber].
    ^port

port: anInteger
    port:=anInteger
```

Appendix H. Source code, GIOPMessage classes

```
Object subclass: #GIOPMessage
  instanceVariableNames: 'messageHeader messageTypeBody messageTypeHeader '
  classVariableNames: 'Socket '
  poolDictionaries: ''
```

GIOPMessage comment: 'PoolDictionary definition:

```
        Smalltalk at: #MsgType put:(
        EsPoolDictionary new
                at: "RequestHeader" put: 0;
                at: "ReplyHeader" put: 1;
                at: "CancelRequestHeader" put:2;
                at: "LocateRequestHeader" put:3;
                at: "LocateReplyHeader" put:4;
                at: "CloseConnection" put:5;
                at: "MessageError" put:6;
                yourself)
'
```

GIOPMessage public class methods

```
clearSocket
        Socket:=nil.

locateRequestFrom: aDirectedMessage
        |aGIOPMessage|
        aGIOPMessage:=self new.

        aGIOPMessage messageTypeHeader: LocateRequestHeader new.
        aGIOPMessage messageTypeHeader objectKey: (ObjectKey new
                interfaceName: ('Test::',aDirectedMessage receiver class printString);
                objectName: aDirectedMessage receiver printString;
                yourself
        ).

        aGIOPMessage messageHeader: (MessageHeader new
                magic: 'GIOP';
                byteOrder: false;
                messageType: (aGIOPMessage messageTypeDict at: #LocateRequestHeader);
                messageSize:
                        (aGIOPMessage messageTypeHeader asIIOPByteArray: false) size;
                yourself
        ).

        ^aGIOPMessage

new: aByteArray
        |aGIOPMessage aByteArrayCopy|
        aGIOPMessage:=self new.
        aGIOPMessage messageHeader: (MessageHeader new: aByteArray).
        aByteArrayCopy:=aByteArray
                chopFromBegin: 12
                align: 1.

        (aByteArrayCopy size > 0) ifTrue:[
                (aGIOPMessage messageHeader messageType=(aGIOPMessage messageTypeDict at:#RequestHeader)) ifTrue:[
                        aGIOPMessage messageTypeHeader: (RequestHeader
                                new: aByteArrayCopy
                                byteOrder: aGIOPMessage messageHeader byteOrder)
                ].
                (aGIOPMessage messageHeader messageType=(aGIOPMessage messageTypeDict at:#ReplyHeader)) ifTrue:[
                        aGIOPMessage messageTypeHeader: (ReplyHeader
                                new: aByteArrayCopy
```

```
						byteOrder: aGIOPMessage messageHeader byteOrder)
				].
				(aGIOPMessage messageHeader messageType=(aGIOPMessage messageTypeDict at:#CancelRequestHeader)) ifTrue:[
					aGIOPMessage messageTypeHeader: (CancelRequestHeader
						new: aByteArrayCopy
						byteOrder: aGIOPMessage messageHeader byteOrder)
				].
				(aGIOPMessage messageHeader messageType=(aGIOPMessage messageTypeDict at:#LocateRequestHeader)) ifTrue:[
					aGIOPMessage messageTypeHeader: (LocateRequestHeader
						new: aByteArrayCopy
						byteOrder: aGIOPMessage messageHeader byteOrder)
				].
				(aGIOPMessage messageHeader messageType=(aGIOPMessage messageTypeDict at:#LocateReplyHeader)) ifTrue:[
					aGIOPMessage messageTypeHeader: (LocateReplyHeader
						new: aByteArrayCopy
						byteOrder: aGIOPMessage messageHeader byteOrder)
				].
				(aGIOPMessage messageHeader messageType=(aGIOPMessage messageTypeDict at:#CloseConnection)) ifTrue:[
					aGIOPMessage messageTypeHeader: nil
				].
				(aGIOPMessage messageHeader messageType=(aGIOPMessage messageTypeDict at:#MessageError)) ifTrue:[
				aGIOPMessage messageTypeHeader: nil
				].
				aByteArrayCopy:=aByteArrayCopy
					chopFromBegin:
						(aGIOPMessage messageTypeHeader asIIOPByteArray: aGIOPMessage messageHeader byteOrder)
size
					align:1.
				(aByteArrayCopy size > 0) ifTrue:[
					(aGIOPMessage messageTypeHeader class=LocateReplyHeader) ifTrue:[
						(aGIOPMessage messageTypeHeader locateStatus=2) ifTrue:[
							"MessageBody is an IOR"
							aGIOPMessage messageTypeBody: (IOR
								new: aByteArrayCopy
								byteOrder: aGIOPMessage messageHeader byteOrder)
						].
					]
					ifFalse:[
						"MessageBody is a parameter Struct"
						aGIOPMessage messageTypeBody: aByteArrayCopy
					].
				].
		].
		^aGIOPMessage

requestFrom: aDirectedMessage
	|aGIOPMessage|
	aGIOPMessage:=self new.

	aGIOPMessage messageTypeHeader: RequestHeader new.
	aGIOPMessage messageTypeHeader
		objectKey: (ObjectKey new
			interfaceName: ('Test::',aDirectedMessage receiver class printString);
			objectName: aDirectedMessage receiver printString;
			yourself);
		operation: aDirectedMessage selector asString;
		yourself.

	aGIOPMessage messageTypeBody: Struct new.
	aDirectedMessage arguments do:[:each|
		aGIOPMessage messageTypeBody add: each
	].

	aGIOPMessage messageHeader: (MessageHeader new
		magic: 'GIOP';
		byteOrder: false;
		messageType: (aGIOPMessage messageTypeDict at: #RequestHeader);
		messageSize: (
```

```
                ((aGIOPMessage messageTypeHeader asIIOPByteArray: false) size) +
                ((aGIOPMessage messageTypeBody asIIOPByteArray: false) size)
        );
        yourself
    ).

    ^aGIOPMessage
```

GIOPMessage public methods

```
asIIOPByteArray
    |aCollection aHeader aTypeHeader aTypeBody aByteOrder|
    aByteOrder:=self messageHeader byteOrder.

    aTypeHeader:=self messageTypeHeader asIIOPByteArray: aByteOrder.
    aTypeBody:=self messageTypeBody asIIOPByteArray: aByteOrder.
    self messageHeader messageSize: (aTypeHeader size + aTypeBody size).
    self messageHeader messageType:
        (self messageTypeDict at:(self messageTypeHeader class printString asSymbol)).
    aHeader:=self messageHeader asIIOPByteArray.

    aCollection:=OrderedCollection new

        "MessageHeader"
        addAll: aHeader;

        "MessageTypeHeader"
        addAll: aTypeHeader;

        "MessageTypeBody"
        addAll: aTypeBody;
        yourself.

    ^aCollection asByteArray

byteOrder
    self messageHeader isNil
    ifTrue:[^false]
    ifFalse:[^self messageHeader byteOrder].

createReturnMessage: anORB

    "The MessageTypeHeaders return a GIOPMessage because the MessageHeader
        depends on the message type"
    ^self messageTypeHeader
        createReturnMessage: anORB
        messageTypeBody: self messageTypeBody
        byteOrder: self messageHeader byteOrder.

messageHeader
    ^messageHeader

messageHeader: aMessageHeader
    messageHeader:=aMessageHeader

messageSize
    ^((self messageTypeHeader asIIOPByteArray: self byteOrder) size +
    (self messageTypeBody size))

messageTypeBody
    "Return the value of messageTypeBody."
    messageTypeBody isNil ifTrue:[self messageTypeBody: #[]].
    ^messageTypeBody

messageTypeBody: aMessageBody
    "Save the value of messageTypeBody."

    messageTypeBody := aMessageBody
```

```
messageTypeDict
	^(Dictionary new
		at:#RequestHeader put:0;
		at:#ReplyHeader put:1;
		at:#CancelRequestHeader put:2;
		at:#LocateRequestHeader put:3;
		at:#LocateReplyHeader put:4;
		at:#CloseConnection put:5;
		at:#MessageError put:6;
		yourself)

messageTypeHeader
	"Return the value of messageTypeHeader."

	^messageTypeHeader

messageTypeHeader: aMessageTypeHeader
	"Save the value of messageTypeHeader."

	messageTypeHeader := aMessageTypeHeader

sendMessage: aHost port: aPort
	|myConnectionSpec myBuffer a|
	Socket isNil ifTrue:[
		myConnectionSpec:=AbtTCPConnectionSpec new
			hostType: #AbtTCPInetHost;
			hostId: aHost;
			port: aPort;
			yourself.

		Socket:=(AbtSocket new
			connectUsing: myConnectionSpec;
			yourself
		).
	].

	Socket sendData: (OrderedCollection new
		addAll: self messageHeader asIIOPByteArray;
		addAll: (self messageTypeHeader asIIOPByteArray: self messageHeader byteOrder);
		addAll: (self messageTypeBody asIIOPByteArray: self messageHeader byteOrder);
		yourself) asByteArray.

	[(myBuffer:=Socket receive) length=0] whileTrue:[].
	^myBuffer asByteArray

sendMyMessage
	|a|
	self messageTypeHeader: (RequestHeader new
		requestId: 1;
		responseExpected: true;
		objectKey: (ObjectKey new
			interfaceName: 'Ralf::AnyReturner';
			objectName: 'ElcIIOPProtocol';
			yourself
		);
		operation: 'name';
		yourself
	).
	self messageTypeBody: #[].
	self messageHeader: (MessageHeader new
		byteOrder: false;
		messageType: 0;
		messageSize: self messageSize;
		yourself
	).
	a:=self asDirectedMessage.
```

MessageHeader

```
Object subclass: #MessageHeader
  instanceVariableNames: 'magic gIOPVersion byteOrder messageType messageSize '
  classVariableNames: ''
  poolDictionaries: 'MsgType '
```

MessageHeader public class methods

```
new: aByteArray
        |aMessageHeader aByteArrayCopy|
        aMessageHeader:=self new.
        ((aByteArray copyFrom:1 to:4) asString='GIOP') ifFalse:[
                self error:'Not a GIOP message'
        ].
        aMessageHeader magic: 'GIOP'.
        aByteArrayCopy:=aByteArray
                chopFromBegin: 4
                align:1.

        aMessageHeader gIOPVersion:
                (Version new: aByteArrayCopy).
        aByteArrayCopy:=aByteArrayCopy
                chopFromBegin:2
                align:1.

        aMessageHeader byteOrder:
                (Boolean new: aByteArrayCopy byteOrder: false).
        aByteArrayCopy:=aByteArrayCopy
                chopFromBegin:1
                align:1.

        aMessageHeader messageType:
                (aByteArrayCopy at:1).
        aByteArrayCopy:=aByteArrayCopy
                chopFromBegin:1
                align:1.

        aMessageHeader messageSize:
                (Integer new: aByteArrayCopy byteOrder: aMessageHeader byteOrder).

        ^aMessageHeader
```

MessageHeader public methods

```
asIIOPByteArray
        |aCollection|
        aCollection:=OrderedCollection new
                addAll: self magic asByteArray;
                addAll: self gIOPVersion asIIOPByteArray;
                addAll: self byteOrder asIIOPByteArray;
                add: (self messageType);
                addAll: (self messageSize asIIOPByteArray: self byteOrder);
                yourself.
        ^aCollection asByteArray

byteOrder
        byteOrder isNil ifTrue:[self byteOrder: false].
        ^byteOrder

byteOrder: aBoolean
        "       false           Big Endian
                true            Little Endian"
```

```
        byteOrder:=aBoolean

gIOPVersion
        gIOPVersion isNil ifTrue:[self gIOPVersion: Version new].
        ^gIOPVersion

gIOPVersion: aVersion
        gIOPVersion:=aVersion

magic
        magic isNil ifTrue:[self magic: 'GIOP'].
        ^magic

magic: aString
        magic:=aString

messageSize
        ^messageSize

messageSize: anInteger
        messageSize:=anInteger

messageType
        ^messageType

messageType: anInteger
        messageType:=anInteger
```

MessageTypeHeader

```
Object subclass: #MessageTypeHeader
  instanceVariableNames: 'requestId '
  classVariableNames: 'Language '
  poolDictionaries: ''
```

MessageTypeHeader public methods

```
asIIOPByteArray: aByteArray
        self shouldNotImplement

requestId
        ^requestId

requestId: anInteger
        requestId:= anInteger
```

RequestHeader

```
MessageTypeHeader subclass: #RequestHeader
  instanceVariableNames: 'objectKey operation principal responseExpected serviceContext '
  classVariableNames: ''
  poolDictionaries: ''!
```

RequestHeader public class methods

```
new: aByteArray byteOrder: aByteOrder
        |aRequestHeader index|
        aRequestHeader:=self new.
        index:=1.

        aRequestHeader serviceContext:
                (CORBASequence new: aByteArray byteOrder: aByteOrder type: CORBAOctet).
        index:=(index+
                (aRequestHeader serviceContext asIIOPByteArray: aByteOrder) size)
                align: 4.
```

```
	aRequestHeader requestId:
		(CORBALong new: (aByteArray from:index) byteOrder: aByteOrder) value.
	index:=index+4.

	aRequestHeader responseExpected:
		(CORBABoolean new: (aByteArray from:index) byteOrder: aByteOrder) value.
	index:=(index+1) align:4.

	aRequestHeader objectKey: (ObjectKey
		new: (aByteArray from: index)
		byteOrder: aByteOrder).
	index:=(index+aRequestHeader objectKey size) align: 4.

	aRequestHeader operation:
		(CORBAString new: (aByteArray from: index) byteOrder: aByteOrder) value.
	index:=index+
		(aRequestHeader operation size +5) align:4.

	aRequestHeader requestingPrincipal:
		(CORBASequence new: (aByteArray from:index) byteOrder: aByteOrder type:CORBAOctet).

	^aRequestHeader.
```

RequestHeader public methods

```
addServiceContext: aServiceContext
	self serviceContext add: aServiceContext

asIIOPByteArray: aByteOrder
	^(CORBAStruct new
		add: self serviceContext asCORBAType;
		add: self requestId asCORBAType;
		add: self responseExpected asCORBAType;
		add: self objectKey asCORBAType;
		add: self operation asCORBAType;
		add: self requestingPrincipal asCORBAType;
		yourself) asIIOPByteArray: aByteOrder

createReturnMessage: anORB messageTypeBody: aByteArray byteOrder: aByteOrder
	(Language='Java') ifTrue:[
		^(self createReturnMessageJava: anORB messageTypeBody: aByteArray byteOrder: aByteOrder)
	].
	(Language='Smalltalk') ifTrue:[
		^(self createReturnMessageSmalltalk: anORB messageTypeBody: aByteArray byteOrder: aByteOrder)
	]

createReturnMessageJava: anORB messageTypeBody: aByteArray byteOrder: aByteOrder

	|aReplyMessage anInterfaceDef anOperationName anOperationDef aDirectedMessage
		result aNewName anIOR aMessageBody aGIOPMessage|
	aReplyMessage:=ReplyHeader new
		requestId: self requestId;
		serviceContext: self serviceContext;
		yourself.

	anInterfaceDef:=anORB interfaceRepository resolveInterface:
		self objectKey interfaceName asRepositoryId.

	(self operation beginsWith:'RESERVEDWORD') ifTrue:[
		anOperationName:=self operation copyFrom: ('RESERVEDWORD' size+1) to: self operation size
	]
	ifFalse:[
		anOperationName:=self operation copy
	].

	anOperationDef:=anInterfaceDef operationDefs at:(
		anInterfaceDef operationDefs findFirst:[:a|a name=anOperationName]).
```

```
	aDirectedMessage:=DirectedMessage
		selector: anOperationDef smalltalkName
		arguments: (aByteArray asArgumentArray: anOperationDef parameterDefs byteOrder: aByteOrder)
		receiver: (anORB members at: self objectKey objectName asSymbol).

	(aDirectedMessage selector=#garbageCollect) ifTrue:[
		"kill the object if it is not a default factory"
		(self objectKey objectName beginsWith: 'Default') ifFalse:[
			anORB members
				removeKey: self objectKey objectName asSymbol
				ifAbsent:[].
			Transcript cr;show: ('Garbage collected ',self objectKey objectName)
		].
		result:=CORBAVoid new.
	]
	ifFalse:[
		(aDirectedMessage selector=#new) ifTrue:[

			result:=(anORB members at: self objectKey objectName asSymbol) class new.
		]
		ifFalse:[
			result:=aDirectedMessage send.
		].
	].

	"Depending on the result, set the reply status"
	aReplyMessage replyStatus: 0. "NO_EXCEPTION"

	(anOperationDef returnParameter type~=result asCORBAType class) ifTrue:[
		(anOperationDef returnParameter type class~=result asCORBAType class) ifTrue:[
			self error: 'Method result does not correspond to the interface definition'
		].
	].

	"Subscribe the IOR's and build messagebody"
	(result asCORBAType class=CORBAObjref) ifTrue:[
		aMessageBody:=result asCORBAType.
		(aMessageBody value profiles at:1) objectLocator objectKey objectName: anORB newTransientName printString.
		anORB subscribe: result name: (aMessageBody value profiles at:1) objectLocator objectKey objectName.
	]
	ifFalse:[
		(anOperationDef returnParameter type=CORBAVoid)
		ifTrue:[aMessageBody:=#[]]
		ifFalse:[aMessageBody:=result asCORBAType].
	].

	aGIOPMessage:=GIOPMessage new
		messageTypeHeader: aReplyMessage;
		messageTypeBody: (aMessageBody);
		yourself.
	aGIOPMessage messageHeader: (MessageHeader new
			messageType: 1; "Reply message"
			messageSize: aGIOPMessage messageSize;
			byteOrder: aByteOrder;
			yourself).
	^aGIOPMessage

createReturnMessageSmalltalk: anORB messageTypeBody: aByteArray byteOrder: aByteOrder

	|aReplyMessage anInterfaceDef anOperationName anOperationDef aDirectedMessage
		result aNewName anIOR aMessageBody aGIOPMessage|
	aReplyMessage:=ReplyHeader new
		requestId: self requestId;
		serviceContext: self serviceContext;
		yourself.

	aDirectedMessage:=DirectedMessage
		selector: self operation asSymbol
```

```
			arguments: (aByteArray asArgumentArrayFromAny: aByteOrder)
			receiver: (anORB members at: self objectKey objectName asSymbol).

		result:=aDirectedMessage send.

		"Depending on the result, set the reply status"
		aReplyMessage replyStatus: 0. "NO_EXCEPTION"

		"Always return an IOR"
		anORB subscribe: result name: (aNewName:=anORB newTransientName printString).
		anIOR:=(IOR new
			typeId: ('IDL:',self objectKey interfaceName moduleName,'/',result class printString,':1.0');
			addProfile: (TaggedProfile new
				tag: 0;
				objectLocator: (ObjectLocator new
					host: anORB tCPSettings host dottedDecimalAddress;
					port: anORB tCPSettings port portNumber;
					iIOPVersion: (Version new major: 1;minor: 0;yourself);
					objectKey: (ObjectKey new
						interfaceName: (self objectKey interfaceName moduleName,'::',result class printString);
						objectName: aNewName;
						yourself);
					yourself);
				yourself);
			yourself).
		aMessageBody:=anIOR.

		aGIOPMessage:=GIOPMessage new
			messageTypeHeader: aReplyMessage;
			messageTypeBody: aMessageBody;
			yourself.
		aGIOPMessage messageHeader: (MessageHeader new
				messageType: 1; "Reply message"
				messageSize: aGIOPMessage messageSize;
				byteOrder: aByteOrder;
				yourself).
		^aGIOPMessage

objectKey
		"Return the value of objectKey."
		objectKey isNil ifTrue:[self objectKey: ObjectKey new].
		^objectKey

objectKey: anObjectKey
		"Save the value of objectKey."

		objectKey := anObjectKey

operation
		"Return the value of operation."

		^operation

operation: aString
		"Save the value of operation."

		operation := aString.

printString
		^Time now printString,'RequestHeader: requestId=',self requestId printString,' operation=',self operation,' objectKey=',self objectKey
printString

requestId
		requestId isNil ifTrue:[self requestId: 0].
		^requestId

requestId: anInteger
		requestId:=anInteger
```

```
requestingPrincipal
	"Return the value of principal."
	principal isNil ifTrue:[self requestingPrincipal: CORBASequence new].
	^principal

requestingPrincipal: aSequence
	"Save the value of principal."
	principal := aSequence

responseExpected
	"Return the value of responseExpected."
	responseExpected isNil ifTrue:[self responseExpected: true].
	^responseExpected

responseExpected: aBoolean
	"Save the value of responseExpected."

	responseExpected := aBoolean.

serviceContext
	"Return the value of serviceContext."
	serviceContext isNil ifTrue:[self serviceContext: CORBASequence new].
	^serviceContext

serviceContext: aSequence
	"Save the value of serviceContext."

	serviceContext := aSequence.
```

ReplyHeader

```
MessageTypeHeader subclass: #ReplyHeader
  instanceVariableNames: 'replyStatus serviceContext '
  classVariableNames: ''
  poolDictionaries: ''!
```

ReplyHeader public class methods

```
new: aByteArray byteOrder: aByteOrder
	|aReplyHeader index|
	aReplyHeader:=self new.
	index:=1.

	aReplyHeader serviceContext:
		(CORBASequence new: aByteArray byteOrder: aByteOrder type:CORBAOctet).
	index:=(index+
		(aReplyHeader serviceContext asIIOPByteArray: aByteOrder) size)
		align: 4.

	aReplyHeader requestId:
		(CORBALong new: (aByteArray from: index) byteOrder: aByteOrder).
	index:=(index+4) align:4.

	aReplyHeader replyStatus:
		(CORBALong new: (aByteArray from: index) byteOrder: aByteOrder).

	^aReplyHeader
```

ReplyHeader public methods

```
asIIOPByteArray: aByteOrder
	^(CORBAStruct new
		add: self serviceContext;
		add: self requestId asCORBAType;
		add: self replyStatus asCORBAType;
```

```
                    yourself) asIIOPByteArray: aByteOrder

printString
        ^Time now printString,'ReplyHeader: requestId=',self requestId printString, ' replyStatus=',self replyStatus printString

replyStatus
        ^replyStatus

replyStatus: anInteger
        replyStatus:=anInteger

serviceContext
        serviceContext isNil ifTrue:[self serviceContext: CORBASequence new].
        ^serviceContext

serviceContext: aSequence
        serviceContext:= aSequence
```

CancelRequestHeader

```
MessageTypeHeader subclass: #CancelRequestHeader
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
```

CancelRequestHeader public class methods

```
new: aByteArray byteOrder: aByteOrder
        |aCancelRequestHeader|
        aCancelRequestHeader:=self new.
        aCancelRequestHeader requestId:
                (Integer new: aByteArray byteOrder: aByteOrder).
        ^aCancelRequestHeader
```

CancelRequestHeader public methods

```
asIIOPByteArray: aByteOrder
        ^(Struct new
                add: self requestId;
                yourself) asIIOPByteArray: aByteOrder
```

LocateRequestHeader

```
MessageTypeHeader subclass: #LocateRequestHeader
  instanceVariableNames: 'objectKey '
  classVariableNames: ''
  poolDictionaries: ''
```

LocateRequestHeader public class methods

```
new: aByteArray byteOrder: aByteOrder
        |aLocateRequestHeader aByteArrayCopy|
        aLocateRequestHeader:=self new.

        aLocateRequestHeader requestId:
                (CORBALong new: aByteArray byteOrder: aByteOrder) value.
        aByteArrayCopy:=aByteArray
                chopFromBegin:4
                align:4.

        aLocateRequestHeader objectKey:(ObjectKey
                new: aByteArrayCopy
                byteOrder: aByteOrder).

        ^aLocateRequestHeader
```

LocateRequestHeader public methods

```
asIIOPByteArray: aByteOrder
        ^(OrderedCollection new
                addAll: (self requestId asIIOPByteArray: aByteOrder);
                addAll: (self objectKey asIIOPByteArray: aByteOrder);
                yourself) asByteArray

createReturnMessage: anORB messageTypeBody: aByteArray byteOrder: aByteOrder
        "Creates a server message (LocateReply) from this message"
        |aMessage aGIOPMessage|
        aMessage:=LocateReplyHeader new.
        aMessage requestId: self requestId.
        (anORB members includesKey: self objectKey objectName asSymbol) ifTrue:[
                aMessage locateStatus: 1. "OBJECT_HERE"
        ]
        ifFalse:[
                aMessage locateStatus: 0. "UNKNOWN_OBJECT"
        ].
        aGIOPMessage:=(GIOPMessage new
                messageTypeHeader: aMessage;
                messageTypeBody: nil;
                yourself).
        aGIOPMessage messageHeader: (MessageHeader new
                messageType: (aGIOPMessage messageTypeDict at: #LocateReplyHeader);
                messageSize: aGIOPMessage messageSize;
                byteOrder: aByteOrder;
                yourself).
        ^aGIOPMessage

objectKey
        ^objectKey

objectKey: aSequence
        objectKey:=aSequence

printString
        ^'LocateRequestHeader: requestId=',self requestId,' objectKey=',self objectKey printString
```

LocateReplyHeader

```
MessageTypeHeader subclass: #LocateReplyHeader
  instanceVariableNames: 'locateStatus '
  classVariableNames: ''
  poolDictionaries: ''
```

LocateReplyHeader public class methods

```
new: aByteArray byteOrder: aByteOrder
        |aLocateReplyHeader aByteArrayCopy|
        aLocateReplyHeader:=self new.

        aLocateReplyHeader requestId:
                (Integer new: aByteArray byteOrder: aByteOrder).
        aByteArrayCopy:=aByteArray
                chopFromBegin: 4
                align: 4.

        aLocateReplyHeader locateStatus:
                (Integer new: aByteArrayCopy byteOrder: aByteOrder).
        ^aLocateReplyHeader
```

LocateReplyHeader public methods

```
asIIOPByteArray: aByteOrder
```

```
    ^(Struct new
            add: self requestId;
            add: self locateStatus;
            yourself) asIIOPByteArray: aByteOrder

locateStatus
    ^locateStatus

locateStatus: anInteger
    "0 UNKNOWN_OBJECT"
    "1 OBJECT_HERE"
    "2 OBJECT_FORWARD (IOR in the Message body"
    locateStatus:=anInteger

printString
    ^'LocateReplyHeader: requestId=',self requestId,' locateStatus=',self locateStatus
```

Appendix I. Source code, Repository classes

Repository

```
Object subclass: #Repository
  instanceVariableNames: 'moduleDefs '
  classVariableNames: ''
  poolDictionaries: ''
```

Repository public methods

```
addModuleDef: aModuleDef
        self moduleDefs add: aModuleDef

moduleDefs
        moduleDefs isNil ifTrue:[self moduleDefs: OrderedCollection new].
        ^moduleDefs

moduleDefs: aCollection
        moduleDefs:=aCollection!

resolveInterface: aRepId
        "       aRepId should by a repository Id like: 'IDL:ModuleName/InterfaceName:1.0'.
                This method returns the InterfaceDef specified by aRepId
        "

        |aModuleName anInterfaceName i iOld aModuleDef anInterfaceDef operationDef aTreeNode result|
        i:=1.
        [(aRepId at:i)=$/] whileFalse:[i:=i+1].

        aModuleName:=aRepId copyFrom: 5 to: (i-1).

        iOld:=i.
        [(aRepId at:i)=$:] whileFalse:[i:=i+1].
        anInterfaceName:=aRepId copyFrom: (iOld+1) to: (i-1).

        aModuleDef:=self moduleDefs at: (
                self moduleDefs findFirst:[:each|each name=aModuleName]
        ).
        anInterfaceDef:=aModuleDef findInterface: (InterfaceDef new
                name: anInterfaceName;
                yourself).
        ^anInterfaceDef.
```

ModuleDef

```
Object subclass: #ModuleDef
  instanceVariableNames: 'interfaceDefs name '
  classVariableNames: ''
  poolDictionaries: 'CldtConstants '
```

ModuleDef public methods

```
= aModuleDef

        ^(self name=aModuleDef name)

addInterfaceDef: anInterfaceDef
        self interfaceDefs add: anInterfaceDef

allInterfaceDefs
        |aCollection|
        aCollection:=OrderedCollection new addAll: self interfaceDefs;yourself.
        self interfaceDefs do:[:each|
```

```
            aCollection addAll: each allInterfaceDefs
    ].
    ^aCollection

asIDLSourceCodeForJava
    |aString|
    aString:='module ',self name,'{',13 asCharacter asString, 10 asCharacter asString.
    self interfaceDefs do:[:each|
            aString:=aString, '  interface ',each name,';',13 asCharacter asString, 10 asCharacter asString.
    ].
    self interfaceDefs do:[:each|
            aString:=aString, (each asIDLSourceCodeForJava: '') sourceCode.
    ].
    aString:=aString,'};'.
    ^(IDLSourceCode new
            moduleName: self name;
            sourceCode: aString;
            yourself).

asJavaSourceCodes
    |aCollection|
    aCollection:=OrderedCollection new.
    self interfaceDefs do:[:each |
            aCollection addAll: (each asJavaSourceCodes: self name)
    ].
    ^aCollection

copy
    |aModuleDef|
    aModuleDef:=ModuleDef new.
    aModuleDef name: self name.
    self interfaceDefs do:[:each|
            aModuleDef addInterfaceDef: each copy
    ].
    ^aModuleDef.

findInterface: anInterfaceDef
    |result|
    self interfaceDefs do:[:each|
            (result:=each findInterface: anInterfaceDef) isNil ifFalse:[^result].
    ].
    ^nil

interfaceDefs
    interfaceDefs isNil ifTrue:[self interfaceDefs: OrderedCollection new].
    ^interfaceDefs.

interfaceDefs: aCollectionOfInterfaceDefs

    interfaceDefs := aCollectionOfInterfaceDefs

name
    name isNil ifTrue:[self name: ''].
    ^name

name: aString
    name:=aString
```

InterfaceDef

```
Object subclass: #InterfaceDef
  instanceVariableNames: 'name operationDefs interfaceDefs '
  classVariableNames: ''
  poolDictionaries: 'CldtConstants '
```

InterfaceDef public class methods

```
fromSmalltalkClass: aClass
	|anOperationDef anInterfaceDef anOrderedDictionary|
	anInterfaceDef:=self new.
	anInterfaceDef name: aClass printString.
	(aClass respondsTo: #IS_instanceInterfaceSpec) ifTrue:[
		(anOrderedDictionary:=aClass IS_instanceInterfaceSpec features) doWithIndex:[:each :index|
			(each class=AbtActionSpec) ifTrue:[
				anInterfaceDef addOperationDef: (OperationDef fromAbtActionSpec: each selector:
(anOrderedDictionary keyAtIndex: index)).
			].
			(each class=AbtAttributeSpec) ifTrue:[
				(OperationDef fromAbtAttributeSpec: each selector: (anOrderedDictionary keyAtIndex: index))
do:[:eachOperation|
					anInterfaceDef addOperationDef: eachOperation
				].
			].
		].
	].
	^anInterfaceDef

fromSmalltalkClass: aSmalltalkClass returnParameter: aReturnParameter
	|anInterfaceDef theSubclasses anInterfaceName|
	anInterfaceDef:=InterfaceDef new.
	(aSmalltalkClass class=Metaclass) ifTrue:[
		anInterfaceName:=aSmalltalkClass printString chopTillSpace,'Class'.
	]
	ifFalse:[
		anInterfaceName:=aSmalltalkClass printString.
	].
	anInterfaceDef name: anInterfaceName.
	((aSmalltalkClass methodsArray select:[:a|a~=nil]) collect:[:a|a selector]) do:[:each|
		anInterfaceDef addOperationDef: (OperationDef new
			name: each asIDLOperationString;
			smalltalkName: each;
			parameterDefs: each asParameterDefs;
			returnParameter: aReturnParameter copy;
			yourself)
	].

	"When no new method is implemented in a MetaClass, implement it"
	(aSmalltalkClass class=Metaclass) ifTrue:[
		(anInterfaceDef operationDefs includes: (OperationDef new name:'new')) ifFalse:[
			anInterfaceDef addOperationDef: (OperationDef new
				name: 'new';
				smalltalkName: #new;
				parameterDefs: OrderedCollection new;
				returnParameter: aReturnParameter copy;
				yourself)
		].
	].

	theSubclasses:=aSmalltalkClass subclasses.
	(theSubclasses size>0) ifTrue:[
		theSubclasses do:[:each|
			anInterfaceDef addInterfaceDef: (self fromSmalltalkClass: each returnParameter: aReturnParameter copy)
		].
		^anInterfaceDef
	]
	ifFalse:[
		anInterfaceDef interfaceDefs: OrderedCollection new.
		^anInterfaceDef
	].
```

InterfaceDef public methods

```
< anInterfaceDef
	anInterfaceDef isNil ifTrue:[^false]
```

```
	ifFalse:[
		^(self name<anInterfaceDef)
	]

<= anInterfaceDef
	(self = anInterfaceDef) ifTrue:[^true].
	^(self < anInterfaceDef).

= anInterfaceDef
	(anInterfaceDef class~=InterfaceDef) ifTrue:[^false].
	^(self name=anInterfaceDef name)

addInterfaceDef: anInterfaceDef
	self interfaceDefs add: anInterfaceDef

addOperationDef: anOperationDef
	self operationDefs add: anOperationDef

addOperationFromString: aString usingSmalltalkName: aSmalltalkName
	|aStringCopy returnParameterString operationName parameterStringCollection aReturnParameter aParameterCollection
aParameterDef|
	aStringCopy:=aString chopBeginEnd.
	returnParameterString:=aStringCopy chopTillSpace.
	operationName:=(aStringCopy:=(aStringCopy copyFrom: (aString chopTillSpace size +1) to: aString size) chopBeginEnd)
chopTillBracketOpen.
	parameterStringCollection:=aStringCopy betweenBrackets asSeperatorByCommaCollection.

	aReturnParameter:=ParameterDef fromString: returnParameterString.
	aParameterCollection:=OrderedCollection new.
	parameterStringCollection do:[:each|
		(aParameterDef:=ParameterDef fromString: each) isNil ifFalse:[
			aParameterCollection add: aParameterDef
		].
	].
	self addOperationDef: (OperationDef new
		name: operationName;
		returnParameter: aReturnParameter;
		parameterDefs: aParameterCollection;
		smalltalkName: aSmalltalkName;
		yourself).

allInterfaceDefs
	|aCollection|
	aCollection:=(OrderedCollection new addAll: self interfaceDefs;yourself).
	self interfaceDefs do:[:each|
		aCollection addAll: each allInterfaceDefs
	].
	^aCollection

asIDLSourceCodeForJava: aParentInterfaceName
	|aString|
	((aParentInterfaceName isNil)|(aParentInterfaceName='')) ifTrue:[aString:='	interface ',self name,'{',LineDelimiter.]
	ifFalse:[
		aString:='	interface ',self name,':',aParentInterfaceName,'{',LineDelimiter.
	].
	(self operationDefs asSortedCollection:[:a :b|a<b]) do:[:each|
		aString:=aString, each asIDLStringForJava
	].
	aString:=aString, '	};', LineDelimiter.
	self interfaceDefs do:[:each|
		aString:=aString, (each asIDLSourceCodeForJava: self name) sourceCode.
	].
	^(IDLSourceCode new
		moduleName: '';
		sourceCode: aString;
		yourself).

asJavaSourceCodes: aModuleName
```

```
        |aString aCollection|
        aCollection:=OrderedCollection new.
        aString:='class ',self name,' extends ', aModuleName,'.',JavaDefaults skeletonPrefix,self name,JavaDefaults
skeletonPostfix,'{',LineDelimiter.
        aString:=aString, '    ', 'public ',aModuleName,'.',self name,' ',JavaDefaults remoteInstanceName,';',LineDelimiter.
        aString:=aString, (JavaDefaults
                constructors: self name
                moduleName: aModuleName
                host: ORB current tCPSettings host dottedDecimalAddress
                port: ORB current tCPSettings port portNumber
                tab: 1).
        aString:=aString, (JavaDefaults finalizeMethod: 1).
        self operationDefs do:[:each|
                        aString:=aString, each asJavaOperationString
        ].
        aString:=aString,'}',WINLineDelimiter.

        self interfaceDefs do:[:each|
                aCollection addAll: (each asJavaSourceCodes: aModuleName)
        ].

        aCollection add: (JavaSourceCode new
                sourceCode: aString;
                className: self name;
                yourself).
        ^aCollection.

copy
        |anInterfaceDef|
        anInterfaceDef:=InterfaceDef new.
        anInterfaceDef name: self name.
        self operationDefs do:[:each|
                anInterfaceDef addOperationDef: each copy
        ].
        self interfaceDefs do:[:each|
                anInterfaceDef addInterfaceDef: each copy
        ].
        ^anInterfaceDef.

findInterface: anInterfaceDef
        "returns the interface in the tree conform an interfaceDef"
        |result|
        (self=anInterfaceDef) ifTrue:[^self]
        ifFalse:[
                self interfaceDefs do:[:each|
                        (result:=each findInterface: anInterfaceDef) isNil ifFalse:[^result].
                ].
        ].
        ^nil

findParentInterface: anInterfaceDef
        "find the parent of anInterfaceDef"
        (self interfaceDefs includes: anInterfaceDef) ifTrue:[^self]
        ifFalse:[
                self interfaceDefs do:[:each|
                        (each findParentInterface: anInterfaceDef) isNil ifFalse:[^each].
                ].
        ].
        ^nil

getChildren
        "Return the value of getChildren."

        ^interfaceDefs

hasChildren
        "Return the value of hasChildren."
        ^self interfaceDefs isEmpty not
```

```
identifier
        ^name

interfaceDefs
        interfaceDefs isNil ifTrue:[self interfaceDefs: OrderedCollection new].
        ^interfaceDefs

interfaceDefs: aCollectionOfInterfaceDefs
        interfaceDefs:=aCollectionOfInterfaceDefs.

name
        name isNil ifTrue:[self name: ''].
        ^name

name: aString
        name:=aString

operationDefs
        operationDefs isNil ifTrue:[self operationDefs: OrderedCollection new].
        ^operationDefs

operationDefs: aCollection
        operationDefs:=aCollection
```

OperationDef

```
Object subclass: #OperationDef
  instanceVariableNames: 'name parameterDefs returnParameter smalltalkName '
  classVariableNames: ''
  poolDictionaries: 'CldtConstants '
```

OperationDef public class methods

```
finalizeOperation
        |aString|
        aString:='   protected void finalize(){',WINLineDelimiter.
        aString:=aString,'       try{',WINLineDelimiter.
        aString:=aString,'         _remoteInstance.garbageCollect();',WINLineDelimiter.
        aString:=aString,'       }',WINLineDelimiter.
        aString:=aString,'       catch(CORBA.SystemException e){',WINLineDelimiter.
        aString:=aString,'         System.err.println(e);',WINLineDelimiter.
        aString:=aString,'       }',WINLineDelimiter.
        aString:=aString,'   }',WINLineDelimiter.
        ^aString.

fromAbtActionSpec: anAbtActionSpec selector: aSelector
        |anOperationDef|
        anOperationDef:=self new.
        anAbtActionSpec parameters isNil ifFalse:[
                anAbtActionSpec parameters do:[:eachParameter|
                        anOperationDef addParameterDef: (ParameterDef new
                                type: eachParameter parameterClass asCORBAType;
                                name: eachParameter parameterName;
                                yourself).
                ].
        ].
        anAbtActionSpec resultType isNil ifTrue:[
                anOperationDef returnParameter: (ParameterDef new
                        type: Object asCORBAType).
                ]
                ifFalse:[
                        anOperationDef returnParameter: (ParameterDef new
                                type: anAbtActionSpec resultType parameterClass asCORBAType
                        ).
                ].
        anOperationDef name: aSelector asString omitColons.
```

```
        anOperationDef smalltalkName: aSelector.
        ^anOperationDef.

fromAbtAttributeSpec: anAbtAttributeSpec selector: aSelector
        "Returns a collection with operationDefs"
        |anOperationCollection|
        anOperationCollection:=OrderedCollection new.
        anAbtAttributeSpec getSelector isNil ifFalse:[
                anOperationCollection add: (OperationDef new
                        returnParameter: (ParameterDef new
                                type: anAbtAttributeSpec attributeClass asCORBAType;
                                yourself);
                        name: aSelector asString omitColons;
                        smalltalkName: aSelector;
                        yourself).
        ].
        anAbtAttributeSpec setSelector isNil ifFalse:[
                anOperationCollection add: (OperationDef new
                        returnParameter: (ParameterDef new
                                type: CORBAVoid;
                                yourself);
                        addParameterDef: (ParameterDef new
                                type: anAbtAttributeSpec attributeClass asCORBAType;
                                name: 'setParameter';
                                yourself);
                        name: (aSelector asString omitColons,'Set');
                        smalltalkName: aSelector;
                        yourself).
        ].
        ^anOperationCollection
```

OperationDef public methods

```
< anOperationDef
        ^(self name < anOperationDef name)

= anOperationDef
        (anOperationDef respondsTo: #name) ifTrue:[
                ^(self name=anOperationDef name)
        ]
        ifFalse:[
                ^false
        ].

addParameterDef: aParameterDef
        self parameterDefs add: aParameterDef

asIDLString
        "Converts the operation in an IDL source String"
        |aString operationName|
        operationName:=self name.
        aString:=self returnParameter type asIDLString, ' ', operationName.
        aString:=aString, '('.
        self parameterDefs do:[:each|
                aString:=aString, each asIDLString.
                (each~=self parameterDefs last) ifTrue:[aString:=aString,','.].
        ].
        aString:=aString, ');'.
        ^aString

asIDLStringForJava
        "Converts the operation in an IDL source String"
        |aString operationName|
        (JavaDefaults javaReservedWords includesKey: self name) ifTrue:[
                operationName:=OperationDef JavaReservedWords at: self name.
        ]
        ifFalse:[
                operationName:='',self name.
```

```
		].
		aString:='	', self returnParameter type asIDLString, ' ', operationName.
		aString:=aString, '('.
		self parameterDefs do:[:each|
			aString:=aString, each asIDLStringForJava.
			(each~=self parameterDefs last) ifTrue:[aString:=aString,','].
		].
		aString:=aString, ');', LineDelimiter.
		^aString

asJavaOperationString
		|aString operationName aParameterString aParameterNameString|
		aParameterString:=''.
		aParameterNameString:=''.
		self parameterDefs do:[:each|
			aParameterString:=aParameterString, each type asJavaString,' ',each name.
			(each~=self parameterDefs last) ifTrue:[aParameterString:=aParameterString,','].
		].
		self parameterDefs do:[:each|
			aParameterNameString:=aParameterNameString,each name.
			(each~=self parameterDefs last) ifTrue:[aParameterNameString:=aParameterNameString,','].
		].
		^JavaDefaults
			remoteMethod: self name
			returnTypeString: self returnParameter type asJavaString
			parameterString: aParameterString
			parameterNameString: aParameterNameString
			tab: 1

copy
		|anOperationDef|
		anOperationDef:=OperationDef new.
		anOperationDef
			name: self name;
			returnParameter: self returnParameter;
			smalltalkName: self smalltalkName.
		self parameterDescriptions do:[:each|
			anOperationDef addParameterDescription: each copy
		].
		^anOperationDef.

name
		name isNil ifTrue:[self name:''].
		^name

name: aString
		name:=aString

parameterDefs
		parameterDefs isNil ifTrue:[self parameterDefs: OrderedCollection new].
		^parameterDefs

parameterDefs: aCollection
		parameterDefs:=aCollection

returnParameter
		returnParameter isNil ifTrue:[self returnParameter: (ParameterDef new type: (CORBAVoid new);yourself)].
		^returnParameter

returnParameter: aParameterDef
		returnParameter:=aParameterDef

returnParameterIdentifier
		returnParameter isNil ifTrue:[^nil]
		ifFalse:[^self returnParameter type].

smalltalkName
		smalltalkName isNil ifTrue:[self smalltalkName: self name asSymbol].
```

```
        ^smalltalkName.

smalltalkName: aSymbol
        smalltalkName:=aSymbol
```

ParameterDef

```
Object subclass: #ParameterDef
  instanceVariableNames: 'type name '
  classVariableNames: ''
  poolDictionaries: ''

ParameterDef class public class methods

fromString: aString
        |aStringCopy aParameterDef|
        (aString='') ifTrue:[^nil].
        aParameterDef:=self new.
        "Build the parameterDef from an IDL string
        e.g. 'void' or 'in long aName'
        "
        aStringCopy:=aString chopTillSpace.
        ((aStringCopy='in')|(aStringCopy='out')|(aStringCopy='inout')) ifTrue:[
                "we are dealing with a normal parameter"
                aStringCopy:=(aString copyFrom: (aString chopTillSpace size +1) to: aString size) chopBeginEnd.
                aParameterDef type: (CORBAType fromString: aStringCopy chopTillSpace).
                aStringCopy:=(aStringCopy copyFrom: (aStringCopy chopTillSpace size +1) to: aStringCopy size) chopBeginEnd.
                aParameterDef name: aStringCopy.
        ]
        ifFalse:[
                "we are dealing with a returnParameter"
                aParameterDef type: (CORBAType fromString: aString).
        ].
        ^aParameterDef

JavaReservedWords
        ^OperationDef JavaReservedWords

ParameterDef public methods

= aParameterDescription
        (aParameterDescription respondsTo: #name) ifTrue:[
                ^(self name=aParameterDescription name)
        ]
        ifFalse:[
                ^false
        ].

asIDLString
        ^'in ',self type asIDLString,' ', self name

asIDLStringForJava
        |parameterName|
        (JavaDefaults javaReservedWords includesKey: self name) ifTrue:[
                parameterName:=JavaDefaults javaReservedWords at: self name.
        ]
        ifFalse:[
                parameterName:=self name.
        ].
        ^'in ',self type asIDLString,' ', parameterName

copy
        |aParameter|
        aParameter:=ParameterDef new.
        aParameter name: self name.
        aParameter type: self type copy.
        ^aParameter
```

```
name
        name isNil ifTrue:[self name: ''].
        ^name

name: aString
        name:=aString

printString
        ^type printString

type
        type isNil ifTrue:[self type: CORBAVoid new].
        ^type

type: aString
        type:=aString
```

Appendix J. Performance measurement results

To measure the performance of an Orbix CORBA Smalltalk server and a VisiBroker CORBA Java client I performed two measurements which are described in paragraph 5.5. The complete measured results are given in this Appendix.

Number of method invocations	Measurement 1	Measurement 2	Measurement 3	Measurement 4	Average (msec)
1	660	660	660	660	660
2	720	710	720	710	715
5	770	770	770	770	770
10	880	930	940	880	908
20	1100	1100	1150	1100	1113
50	1820	1810	1810	1870	1828
100	3030	3020	3070	3030	3038

Table J-1, Time agains method invocations

Number of bytes	Measurement 1	Measurement 2	Measurement 3	Measure 4	Average (msec)
1000	710	720	770	770	743
2000	770	770	770	770	770
5000	770	820	820	830	810
10000	820	940	930	940	908
20000	1040	1100	1100	1100	1085
50000	1600	1650	1540	1650	1610
100000	2420	2410	2370	2410	2403

Table J-2, Time against sent bytes (one method invocation)

Appendix K. References

[OMG 1995a]
Object Management Group, "The Common Object Request Broker - Architecture and Specification". Revision 2.0, July 1995, Updated July 1996

[OMG 1995b]
Object Management Group, "CORBAservices - Common Object Services Specification". Revised Edition March 31, 1995, Updated: March 28, 1996, Updated: July 15, 1996, Updated: November 22, 1996, Updated: March 1997

[OMG 1995c]
Object Management Group, "Common facilities architecture". Revision 4.0, november 1995

[OMG 1997a]
Object Management Group, "A discussion of the Object Management Architecture". January 1997

[OMG 1997b]
Object Management Group, "IDL/Java language mapping". March 19, 1997

[VOG 1997]
Andreas Vogel and Keith Duddy, "Java Programming with CORBA". John Wiley & Sons, 1997

[VIN 1997]
Steve Vinoski, IONA Technologies inc., "CORBA, integrating diverse applications withing distributed hetrogeneous environments". IEEE Communications Magazine, february 1997, page 46

[DOU 1997]
Douglas C. Schmidt, a.o., "A high-performance end system architecture for real-time CORBA". IEEE Communications Magazine, february 1997, page 72

[MON 1997]
John Montgomery, "Distributing Components, for CORBA and DCOM it's time to get practical". Byte magazine, april 1997, page 93

[POM 1997]
John Pompeii, "Programming with CORBA and DCOM, it just isn't as easy as proponents of either side would have you believe". Byte magazine, april 1997, page 103

[TAN 1996]
Andrew S. Tanenbaum, "Computer Networks". Pretice-Hall inc., New Jersey, 1996

[GOL 1980]
Adele Goldberg and David Robson, "The language and its implementation". Addison Wesley Publishing Company, Xerox Palo Alto Research Center, 1980

[VIS 1997a]
Visigenic, "VisiBroker for Java, reference manual release 2.5". Visigenic Software inc., 1997

[VIS 1997b]
Visigenic, "VisiBroker for Java, programmer's manual release 2.5". Visigenic Software inc., 1997

[ION 1997]
IONA Technologies Ltd., "Orbix/Smalltalk programming guide". Release 1.0, January 1997