WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

BUILDING THE DISTRIBUTED OBJECT VISUALIZATION ENVIRONMENT

by

Michael Kircher

Prepared under the direction of Professor Douglas C. Schmidt

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

, 1998

Saint Louis, Missouri

Building the Distributed Object Visualization Environment

July 18, 1998

Contents

List of Figures

Chapter 1

Introduction

This document describes steps taken in building DOVE, the Distribued Object Visualization Environment. The project served as a milestone in achieving the full functionality of DOVE. Major outcomes are a prototype of DOVE and a tested ACE/TAO environment.

The ACE framework and TAO (The ACE ORB), described below, have been developed in a research, non-commercial environment, therefore the software is not as fully implemented as commercial products and tests have to be performed to assure that the needed functionality is available. Besides assuring functionality tests improve the quality of the ACE framework and TAO.

The structure of this document is as follows. First an overview over the used architectures, like CORBA, ACE, TAO and DOVE, is given. Then the context and objectives of this project are presented. The devolpement of the prototypes is described in phases, which are Analysis, Requirements, Specification and Implementation.

This document also describes an additional project: the implementation of the CORBA Common Object Services LifeCycle Service. It has been added, because it was done within the same independent study class.

Chapter 2

Overview

In the following sections the used architectures and environments are described. Which are: The Common Object Request Architecture (CORBA), the ACE framework, the TAO environment and the DOVE architecture.

2.1 Overview of CORBA

CORBA is a distributed object computing (DOC) middleware standard defined by the Object Management Group (OMG). CORBA is designed to support the development of flexible and reuseable distributed services and applications by (1) separating interfaces from remote implementations and (2) automating many common network programming tasks (such as object registration, location and activation; request demultiplexing; framing and error handling; parameter marshalling and demarshalling; and operation dispatching).

Advances in DOC technology have occured at a time when deregulation and global competition are motivating the need for increased software productivity and quality. Distributed object computing is perceived as a way to control costs through open systems and client/server computing. Likewise, OO design and programming are widely touted as an effective means to reduce cost and improve software quality through reuse and extensibility, and modularity.

The Common Object Request Broker Architecture (CORBA) specifies a system, which provides interoperability between objects in a heterogeneous, distributed environment. CORBA does this in a way transparent to the programmer. The main features are:

- ORB Core
- OMG Interface Definition Language (IDL)
- Language Mappings
- Interface Repository
- Stubs and Skeletons
- Dynamic Invocation and Dispatch

- Object Adapters

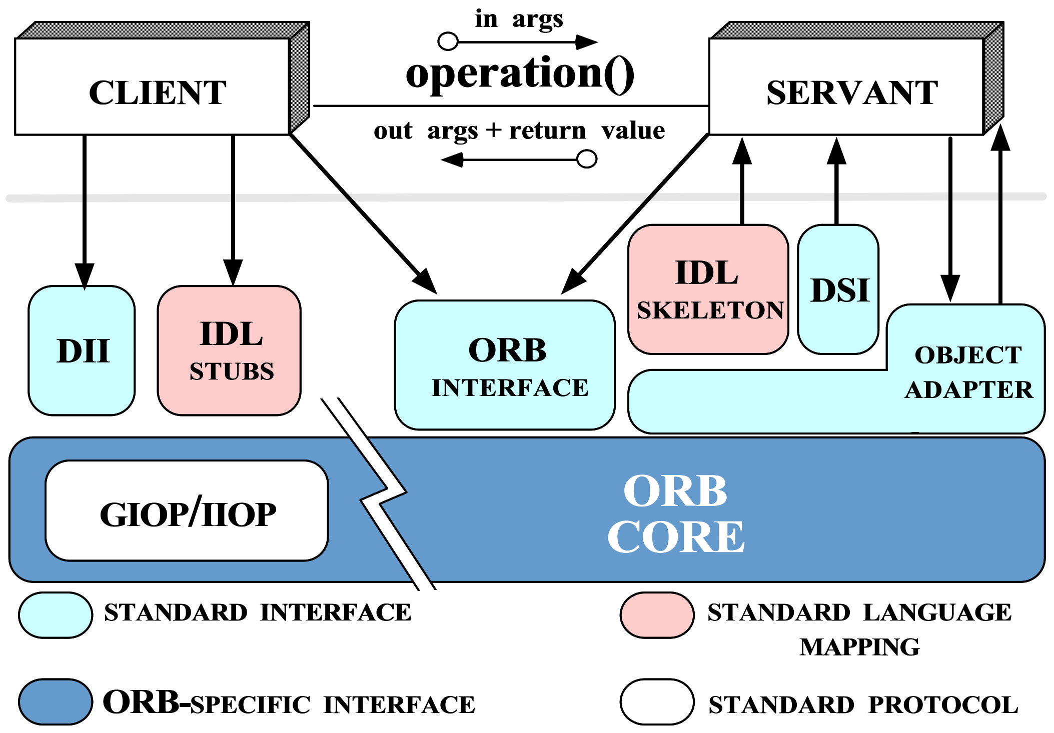

Figure 2.1:

The figure above shows the relationships between the various components.

2.1.1 ORB core

The Object Request Broker (ORB) is the central component, it handles the connection management and delivers data. It is defined by its interfaces and is not required to be a single component. ORB interfaces connect the other components to the ORB core.

The ORB Core is the most crucial part in the CORBA, it is responsible for communication between clients and objects. The communication is done transparently, this is one of the key features of the ORB. The ORB hides the following information from the client:

- Object location: The client does not know where the server object is located. It could be located on the same physical machine, in the same ORB or in a different ORB environment.
- Object Implementation: The server object may be implemented in any language supported by CORBA. The client sees only the interface.
- Object execution state: No information is necessary to the client about the execution state of the server. If necessary the server will be started automatically by the ORB.
- Object communication mechanisms: The client has no information about the underlying communication mechanisms, whether TCP/IP, Novell or any other protocol.

The communication outside the ORB between clients and object implementations is done via OMG IDL or Dynamic Invocation Interface / Dynamic Skeleton Interface (DII/DSI).

2.1.2 Object Creation

A client needs an object reference to make an request to an object. These object references belong only to a single object and are created when the object is created. The client cannot change these object references. They can be obtained in several different ways:

- Object creation: A client can issue a creation request to a so-called `Factory`, a client cannot create an object by its own. The factory object creates a new object and passes the object reference to the client.
- Directory service: This service is responsible for holding object references in association with names and properties. Requests from clients about an object providing a certain service are responded with an object reference belonging to an object, which provides this service. It can be imagined as a DB allowing only two types of requests: object references belonging to names and object references belonging to properties.
- Convert to string and back: Object references can be converted into strings. This is useful for example in the Internet Inter-ORB Protocol to exchange object references.

Like seen above CORBA has no explicit creation service, this fact emerges out of the following rule: "Keep the ORB as simple as possible, and push as much functionality as possible to other OMA components such as Object Services and Common Facilities".

2.1.3 OMG IDL

To be able to access an object, the interface of the object has to be known. The interfaces should have a uniform syntax and semantics, independent from the implementation language. The Interface Definition Language (IDL) does this in specifying the object interfaces in a standardized way. Standardized mapping between implementation language and IDL are provided by OMG. Language-independent interfaces are important within heterogeneous systems, since not all programming languages are supported or available on all platforms. IDL allows defining the methods to access a CORBA object. Some of the supported data types are: long, double, enum, octect and boolean. The data type "any" is a special one. It can contain any data as the name already says. It is enormously important for dynamic environments. Structures and unions are available as well.

2.1.4 OMG IDL - Language Mappings

Language mappings describe how the IDL is mapped to the facilities provided by a certain language. Standardized language mappings exist for C, C++, Smalltalk, Ada95, Cobol and Unix Bourne Shell. A language mapping to Java is announced. These language mappings have to be unambiguous, if not, communication between object implementations in different languages may fail.

2.1.5 Stubs and Skeletons

A stub is the connection between the client and the client ORB. A skeleton is the connection between the object implementation and the server ORB. See Error! Reference source not found.. They are both generated automatically by the IDL compilers and translators. A stub creates and issues requests of a client while a skeleton delivers requests to the CORBA object implementation.

Dispatching through stubs and skeletons is often called static invocation. When a client issues a request, this request is marshaled by the client stub and delivered to the client ORB. Once the request arrives at the target object, the server ORB and the skeleton cooperate to unmarshal the request (convert it from its transmissible form to a programming language form) and dispatch it to the object. When the object has completed the request, the results are sent back the same way, through the server skeleton to the server ORB, ORB connection, client ORB and back through the client stub to the client application.

2.1.6 Interface Repository

Interfaces to objects can be specified in two ways: either in OMG IDL or they can be added to the Interface Repository (IR). Communication via Dynamic Invocation Interface (DII) uses the information in the IR. Object interface definitions, object reference, operation and a list of parameters can be retrieved from the IR. The IR is database, which provides persistent storage of object interface definitions. The IR allows applications to specify their interfaces at run-time and not at compilation-time. This functionality is crucial for gateways between foreign object systems (such as Microsoft Component Object Model applications) and CORBA. Having to recompile and rebuild the gateway every time someone added a new OMG IDL interface type to the system would result in a very difficult management and maintenance problem.

2.1.7 Dynamic Invocation and Dispatch

Beside invocation of objects via OMG IDL another invocation mechanism exists. This consists of:

- Dynamic Invocation Interface (DII) - which supports dynamic client request information.
- Dynamic Skeleton Interface (DSI) - which provides dynamic dispatch to objects.

They can be seen as a generic stub and generic skeleton. Unlike IDL stubs, the Dynamic Invocation and Dispatch mechanism allows clients to make non-blocking deferred synchronous calls which separate send and receive operations. Other forms of calls are: blocking synchronous invocation and oneway invocation.

2.1.8 Object Adapters

Object Adapters serve as glue between the object implementation and the ORB.

An Object Adapter is an interposed object allowing a caller to invoke requests on an object even though the caller does not know that object's true interface. The Object Adapter has the following responsibilities:

- Object registration - OAs supply operations that allow programming language entities to be registered as implementations for CORBA objects.
- Object reference generation - OAs generate object references for CORBA objects.
- Server process activation - If necessary, OAs start up server processes in which objects can be activated.
- Object activation - OAs activate objects if they are not already active when requests arrive for them.
- Request demultiplexing - OAs must cooperate with the ORB to ensure that requests can be received over multiple connections without blocking indefinitely on any single connection.
- Object upcalls - OAs dispatch requests to registered objects

Object Adapters are mostly language dependent the reason is that there are different implementations of object registration. CORBA allows several Object Adapters but it currently only provides one: the Basic Object Adapter (BOA). While an interface is responsible for the type of an implementation, an Object Adapter is responsible for the "style" of object implementation. For instance, activating an object when a request is sent if the object is not already active.

For the BOA four styles of activation are described:

- Per Method
- Shared
- Unshared
- Persistent

Per method activation is when a new server is started every time an object method is invoked. Each method call runs in its own server. Shared activation style servers are those that support multiple objects active at a time. A single server may handle multiple objects all active simultaneously. Unshared activation servers are those that support only one single active object. Multiple method invocations may be handled by the single server as long as they are all on the same object. Persistent servers are those that are always active and do not require activation. Rather these are presumed to be available as long as the system (or machine) is operating.

2.1.9 Inter-ORB Protocol

To allow communication between ORBs of different vendors OMG has released a standardized interoperabilit protocol. General Inter-ORB Protocol (GIOP) was specifically defined to meet the needs of ORB-to-ORB interaction and is designed to work over any transport protocol that meets a minimal set of assumptions. The General Inter-ORB Protocol specifies transfer syntax and a standard set of message formats for ORB interoperation over any connection-oriented transport. GIOP is designed to be simple and easy to implement while still allowing for reasonable scalability and performance.

The Internet Inter-ORB Protocol (IIOP) is a specification how the GIOP has to be implemented using TCP/IP. The support for GIOP and IIOP is mandatory for a CORBA 2.0 ORB. Objects publish their identities and locations in the form of object references. The CORBA 2.0 specification dictates a common format for object references exchanged over IIOP, IOR (Interoperable Object Reference) format. An IOR contains one or more profiles. Each profile describes how a client can contact and send requests to the object using a particular protocol. All legal IORs must have at least one IIOP profile, thus ensuring that wherever that reference goes, any CORBA-compliant ORB will be able to locate the object and send requests to it. The IIOP profile contains the Internet address of the object's server and a key value used by the server to find the specific object described by the reference. Object references can be converted into character strings, which can be published arbitrarily, like URLs, in email messages, files, databases, directories, and so on. Any CORBA-compliant application can convert the string into an IOR and use it to locate and invoke the object.

2.1.10 Common Object Services

Common Object Services (COS) are part of the CORBA standard. Several services, like Naming Service, Trading Service and Event Service have well defined interfaces and functionality.

The Naming Service provides the functionality to bind a name with an object relative to a naming context. A name has to be unique in a given naming context. The user can query the Naming Service for an object by the name of the object. This name has to be the same name, as the one it was registered with.

The Trading Service allows service providers to advertise their services, and service consumers to dynamically discover those services and bind to them. The service provider submits an interface reference to the service and a description of the service's capabilities to the Trading Service. The service consumer queries the Trading Service for all service instances of a given type whose capabilities meet the consumer's requirements.

The Event Service allows the user to send and receive events. Suppliers put events into the Event Channel, which are read by Consumers. Two different models exist for this relationship. One is the PUSH model and the other is the PULL model. The PUSH model is the most popular and most useful. It works in the way, that the Supplier pushes actively events into the Event Channel and the receiver waits passively to get a event pushed to it. The supplier has to register which events it will push and and the consumer has to specify which events it wants to receive. In the PULL model the client pulls actively the Event Channel for events.

Figure 2.2 shows what parts are involved. As can be seen the Event Service supports filtering and correlation.

The Lifecycle service is another service, which is part of the CORBA COSs. This service allows objects to be created through standardized factories, moved to other locations, copied to the same or different locations or deleted. Three kind of interfaces are defined by the CORBA COS specification: the `LifeCycleObject`, which has to be supported by every object conforming to the LifeCycle Service, the `FactoryFinder`, which is an interface for objects capable of finding proper factories and `GenericFactory` which is an interface supported by factories which are forwarding creation requests to concrete factories.

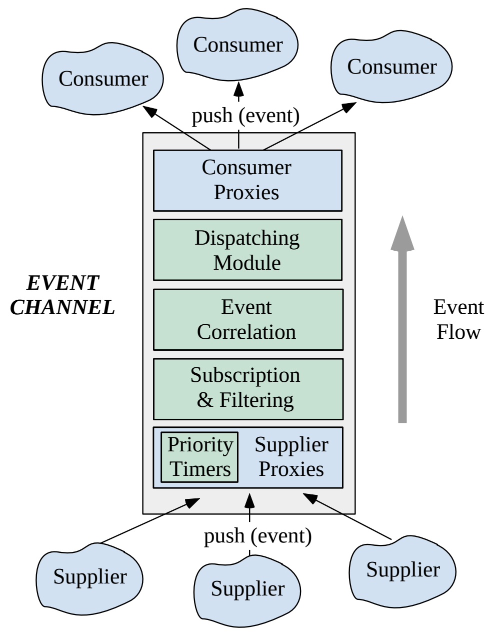

Figure 2.2: Event Channel internals

2.2 Overview of ACE

The following description of ACE was taken from [9]:

The ADAPTIVE Communication Environment (ACE) is an object-oriented (OO) toolkit that implements fundamental design patterns for communication software. ACE is targeted for developers of high-performance communication services and applications on UNIX and Win32 platforms. ACE simplifies the development of OO network applications and services that utilize interprocess communication, event demultiplexing, explicit dynamic linking, and concurrency. ACE automates system configuration and reconfiguration by dynamically linking services into applications at run-time and executing these services in one or more processes or threads.

ACE is freely available and is being used for many commercial projects (such as Ericsson, Bellcore, Siemens, Motorola, Kodak, and Boeing), as well as many academic and industrial research projects. ACE has been ported to a variety of OS platforms including Win32 and most UNIX/POSIX implementations. In addition both C++ and Java versions of ACE are available.

ACE contains a rich set of reusable wrappers, class categories, and frameworks that perform common network programming tasks across a wide range of OS platforms. The tasks provided by ACE include:

- Event demultiplexing and event handler dispatching
- Connection establishment and service initialization
- Interprocess communication and shared memory management.

- Dynamic configuration of distributed communication services.
- Concurrency/parallelism and synchronization.
- Components for higher-level distributed services (such as Name Service, Event Service, Logging Service, Time Service and Token Service).

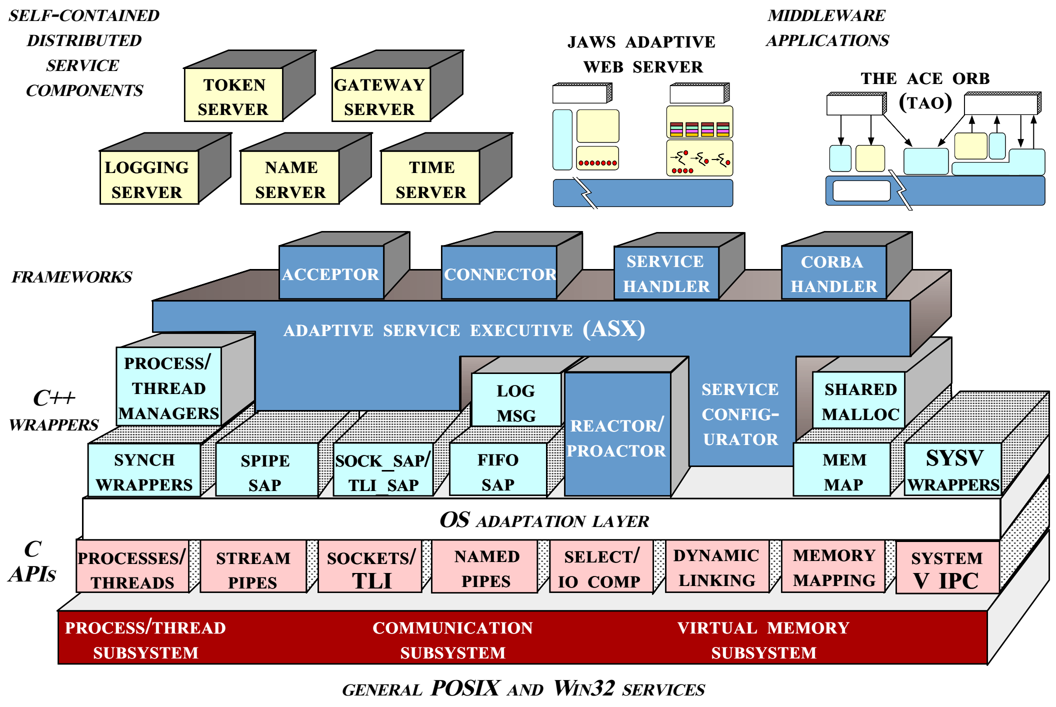

Figure 2.3: ACE layers

The ACE toolkit is designed using a layered architecture. Figure 2.3 illustrates the vertical and horizontal relationship between ACE components. The lower layers of ACE are OO wrappers that encapsulate existing OS network programming mechanisms. The higher layers of ACE extend the wrappers to provide OO frameworks and components that cover a broader range of application-oriented networking tasks and services.

2.2.1 The ACE OS Adaptation Layer

Approximately 10% of the ACE source code are devoted to the OS Adaptation Layer. This layer shields the higher layers of ACE from platform-specific dependencies associated with the following OS mechanisms:

- Multi-threading and synchronization
- Interprocess communication
- Event demultiplexing
- Explicit dynamic linking

- Memory-mapped files and shared memory

2.2.2 The ACE OO Wrappers

Above the OS Adaptation Layer are OO wrappers that encapsulate and enhance the concurrency, interprocess communication (IPC), and virtual memory mechanisms available on modern operating systems like Win32 and UNIX. Applications can combine these components by selectively inheriting, aggregating, and/or instantiating the following ACE wrappers class categories:

- IPC Service Access Point
- Service Initialization
- Concurrency mechanisms
- Memory management mechanisms
- CORBA integration

The use of OO wrappers improves application robustness by encapsulating OS communication, concurrency and virtual memory mechanisms with type-secure OO interfaces. This alleviates the need for applications to directly access the underlying OS libraries, which are written using weakly-typed C interfaces. Therefore, compilers for OO languages like C++ and Java can detect type system violations at compile-time, rather than at run-time. The C++ version of ACE uses inlining extensively to eliminate performance penalties that would otherwise be incurred from the additional type-security and abstraction provided by the wrapper layer.

2.2.3 The ACE Framework

ACE contains a higher layer network programming framework that integrates and enhances the lower layer OS wrappers. This framework supports dynamic configuration of concurrent network daemons composed of application serives. The framework portion of ACE contains the following class categories:

- Reactor - The ACE Reactor provides extensible, object-oriented demultiplexer that dispatches handlers in response to various types of events (e.g. I/O based, timer-based, signal-based and synchronization-based events.)
- Service Configurator - The ACE Service Configurator supports the construction of applications whose services may be configured dynamically at installation time and/or run-time.
- Streams - The ACE Streams components simplify the development of concurrent communication software applications composed of one or more hierarchically-related services.

2.2.4 The ACE Network Service Components

In addition to the wrappers and frameworks, ACE provides a standard library of network service components. These components play two roles in ACE:

1. They illustrate how to utilize the ACE IPC wrappers, Reactor, Service Configurator, Service Initialization, Concurrency, Memory Management and Strams components.
2. They provide reusable components for common distributed system tasks such as logging, naming, locking and time synchronization.

When combined with OO language features (such as classes, inheritance, dynamic binding and parameterized types) and design patterns (such as Abstract Factory, Builder and Service Configurator), the reusable ACE components facilitate the development of communication services and applications, that may be updated and extended without modifying, recompiling, relinking or even restarting running software.

2.3 Overview of TAO

In several areas, like avionics and telecommunication, is a need for real-time CORBA. CORBA alone is not yet suited for performance sensitive, real-time applications for the following reasons:

- **Lack of QoS specification interfaces:** No interfaces are provided by the CORBA 2.0 standard. The clients cannot specify their needs in terms of QoS and servers do not enforce admission control.
- **Lack of QoS enforcement:** Conventional ORBs do not provide end-to-end QoS enforcements. For instance most ORBs use a FIFO strategy for scheduling and dispatching client requests. But FIFO strategies do not obey priorites so a lower priority request could prevent a higher priority request to be served.
- **Lack of performance optimizations:** There exists a significant overhead in most ORBs due to data-copying, inefficient demultiplexing or long chains of intra-ORB virtual method calls. Most ORBs lack of integration with underlying real-time OS and network QoS mechanisms.

TAO (The ACE ORB) is an ORB endsystem architecture for high-performance Real-Time CORBA. TAO complies with the CORBA 2.0 standard and addresses the above mentioned lacks of most ORBs. TAO is developed, implemented and maintained by the Distribued Object Computing group at Washington University.

The Naming Service, Trading Service and Event Service are implemented as part of the CORBA Common Object Services.

Additionaly TAO provides a new Object Service, the TAO real-time Scheduling Service is responsible for allocating scarce system resources to meet application needs. For deterministic real-time systems, the primary function of the Scheduling Service is to guarantee that processing requirements will be met. At the time of writing this document the real-time Scheduling Service was not fully implemented, as a basic functionality the Scheduling Service (briefly Scheduler) is responsible for giving timeouts to registered clients. Real-time information is given to the Scheduler, like period, priority and worst case execution time.

The implementation of the TAO real-time Event Service makes use of the Scheduling Service. Consumers and Suppliers have to register with the Scheduling Service in order to be accepted by the Event Service. The events pushed by the supplier have a struct containing an `CORBA::Any` as event data field. This indicates, that the TAO Event Service is not compatible with other Event Services.

The events pused by the supplier consist out of a struct containing real-time QoS information and a field called EventData. The OMG specification of the Event Service suggests to use *CORBA::Any* in the interface to the *push* for event data.

2.4 Overview of DOVE

The Distributed Object Visualization Environment (DOVE) is meant to be an web-based network management and visualization system using Java and Real-time CORBA

2.4.1 DOVE Objectives

The goal of the Distributed Object Visualization Envirionment is to monitor and control the state of applications distributed throughout a network. DOVE has the following objectives as stated in [10]:

- Create Web-based telecom management application tools using Java component technology (such as JavaBeans) that can automatically generate visualizations of system- and application-level information in a network environment.
- Develop a real-time Agent framework based on TAO (The ACE ORB) to monitor and control applications and network elements in large-scale, hierarchical networks with minimal effort by developers and administrators.
- Develop a platform-independent, persistent, and hierarchical Management Information Base (MIB) that allows end-user applications and management applications to store and retrieve name-value bindings efficiently in a distribued environment. Using components from ACE, applications can store attributes in the DOVE MIB, which makes them automatically eligible for visualization. These attribues may be statistical data, configuration or any other data of interest.

2.4.2 The DOVE Software Architecture

DOVE consists of five components:

1. The DOVE Application
2. The MIB
3. The DOVE Browser
4. The Visualization Component
5. The DOVE Agent

The role of each component in DOVE is described below:

- The **DOVE Application** publishes state variables to be monitored or configured by the Visualization Component. Using pre-defined and/or automatically generated APIs, the application stores and retrieves values in the DOVE MIB. The high degree of dynamism supported by DOVE is particularly suitable for applications that (1) cannot provide a user interface, (2) must be restarted or recompiled to change their configuration, or (3) for which logging is too costly.
- The **MIB** is the logical repository of information in DOVE. MIBs are separate entities from DOVE agents, which allows the MIB to be persistent and the application to write to the MIB, even when an agent is not currently running. Although the MIB is a logical repository, load or store operations on the MIB will cost no more than a read or write to shared memory since the application and MIB typically share the same endsystem.

- The **DOVE Browser** is a Java applet or stand-alone application that discovers, through Agents the applications on a network offering DOVE services. The Browser is also a visualization builder, allowing end-users to bind graphical or data gathering components to data published by the DOVE Application. The Browser can then package the resulting tool in its own applet or application . Likewise, the Browser can simly save it to a file for later use. This "build once, use frequently" approach is provided by JavaBeans.

  In complex system management environments, such as a central office network operations center, there may be multiple DOVE browsers that interact with multiple Agents. It is the responsiblity of the DOVE framework to provide a transparent and scalable infrastructure for large-scale systems management.

- The **Visualization Component** is the conduit through which information from a DOVE application reaches the user and configuration information from the user reaches the application. Each Visualization Component is a Java Bean, which is registered for updates from Agent or Application Proxy. The Visualization Component makes new information public as it arrives and fires events related to those changes.

  Through the DOVE Browser, the user can connect components to these properties and events to monitor and change the state of the application, a concept illustrated in

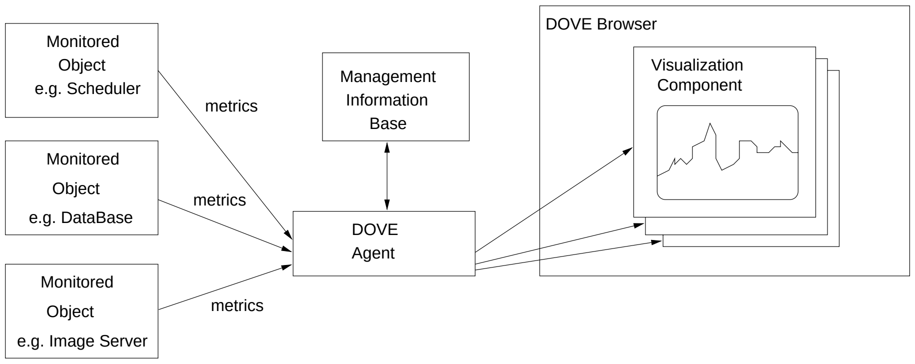
Monitored Object e.g. Scheduler
Monitored Object e.g. DataBase
Monitored Object e.g. Image Server
metrics
metrics
metrics
Management Information Base
DOVE Agent
DOVE Browser
Visualization Component

Figure 2.4: DOVE software architecture

Figure 2.4 explains the relationships between the various DOVE components. For example, suppose an Image Server publishes the outgoing number of megabytes and updates the value every time a client finishes downloading an image. A network management application may want to use the Visualization Component to chart the average outgoing throughput. To achieve this functionality, the Visualization Component would have a property called "outgoing Megabytes," the point data, timestamps indicating the start and finish time of the connection, and an event called"outgoing Megabytes updated." The user would simply connect an averaging graph to the "outgoing throughput" property, which would cause the graph to invisibly attach itself to the timestamp properties and update event, and when the update event occured, the graph would update the current average throughput (in Megabytes/sec) using the new point datum and time elapsed, and plot the next point on the graph.

The standard bridge from JavaBeans to ActiveX makes it possible to exchange information between JavaBeans and COM objects. Thus it is possible to integrate DOVE with Microsoft EXCEL applications running in the Microsoft Office suite. In addition, efforts are underway to integrate JavaBeans with CORBA.

- The **DOVE Agent** runs as a Singleton process on an endsystem containing a DOVE MIB. The Agent performs the following tasks:

  1. Service advertisement
  2. Change notifications
  3. Data reduction and correlation
  4. Visualization configuration

- The optional **Application Proxy** extends the Agent with application-specific functionality. An application proxy is necessary when the application developer wants the Agent and Visualization Component to interact in ways that the generic DOVE framework cannot provide. For example, the application developer may want to send special messages to the Visualization Component when the application sets a flag in the MIB.

Chapter 3

Context

The Distributed Object Computing group is part of the Department of Computer Science at Washington University which is led by Professor Douglas C. Schmidt. Boeing is the primary customer concerning the DOVE project. Boeing is considering the use of DOVE in avionics applications where real-time is needed. This need is satisfied by ACE and TAO, the real-time ORB. Siemens is also interested in the concept of integrating JavaBeans into CORBA applications. Which adds Siemens as a second customer of this project.

Chapter 4

Objectives

In the following the major objectives will be described. As mentioned earlier a major goal of the project is prototyping, this also includes testing for feasability. What means new ways are gone and the outcomes are compared against the expectations about them. Prototyping serves also as a test of the TAO and to collect experience with the various components, e.g. Scheduling Service, in order to improve them.

As an objective introduced by the customer, Boeing, sensor information of an aircraft has to be conveyed from several data sources to several data sinks. The architecture used has to be independent from the number of suppliers(sources) and consumers (sinks). Information filtering and correlation should be supported. As a key architecture the TAO Event Service is introduced to the DOVE project to solve the independence between suppliers and consumers as well as filtering and correlation. The sensor information is encapsulated in so called *events*.

To support location independence of the consumers as well as to have them running in any environment without customization they will be implemented using Java.

As steps towards a full implemenation of DOVE using the Event Service serveral steps have to be taken.

- The interoperability between TAO and an Java ORB (in this case VisiBroker) has to be tested. This test will discover possible flaws in supporting interoperability by TAO. I flaws are detected they have to be debugged and removed.

- The VisiBroker Java ORB has to be tested for the purpose of real-time data visualization.

- The DOVE project makes full use of the Event and Scheduling Service, this will discover possible flaws and restrictions in their implementation.

- Building prototypes of DOVE to support the customer in decision making concerning DOVE. Prototypes are a very helpful media to make customers aware of how things work or will work.

- Integrate the prototypes, especially the DOVE Browser, into the Netscape Browser using the build-in VisiBroker ORB.

Chapter 5

Requirements and Analysis

In this section the two first phases of the software lifecycle of the DOVE project are described.

5.1 Bootstrapping the Naming Service

The prototypes have to communicate with the Scheduling- and Event Service, therefore they should make use of the Naming Service. They find each other through the Naming Service. If the Naming Service were not used, object references to all of them would have to be provided via parameters, which would be tedious.

To be able to use a Common Object Service, the objects, what want to use it must have an object reference to it. There are several ways to obtain one: Feeding in the Interobject Reference (IOR) as a parameter, reading it from an environment variable or asking the Naming Service to provide it.

The last option would work if all Services were properly registered with the Naming Service. It is possible to get object references to all other Common Object Services by just asking the Naming Service with their name and an object reference is replied. Therefore it must be assured that every Common Object Service registers properly with the Naming Service.

One concern remains: Where does the first object get the object reference of the Naming Service?

5.2 The DOVE Browser

A first demo of a DOVE Browser, refered to as “demo” in the following sections, uses Java and Unix sockets. The demo is very restrictive, e.g. it allows only one data source and only one data sink. The data transported consists of weapon status, navigation infromation and statistical data. The demo is a result of some first steps toward DOVE and is used in this project as a basis for a new, more advanced DOVE Browser.

5.2.1 The old DOVE Browser

The demo consists of a Java front end showing the available weapons, an artifical horizon and several statistical data. A statistical data is displayed in its own Java-Canvas and they are all together contained within a Java-Panel. The weapons are listed in a Java-Panel (class `Display_Weapons`) as well as the artifical horizon (class `Display_Art_Horizon`). Class `Display_Weapons` and class `Display_Art_Horizon` inherit from an interface called `Display_Object`. A Factory pattern is used to manage weapon and artifical horizon displays. The factory is called `Display_Factory_Object`.

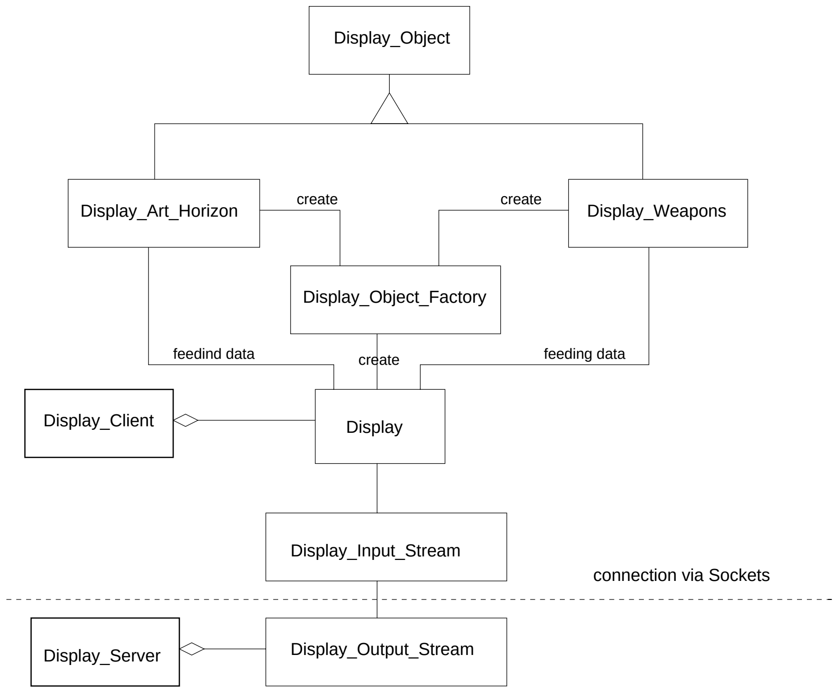

Figure 5.1: Object Diagram of the old DOVE Browser

Figure 5.1 explains the dependencies between the objects (classes). The communication between source and sink (visualization) is done via sockets. No data structure is introduced. The input for this demo is randomly generated by a process, which feeds this data to a socket connected with the display part of the demo.

5.2.2 The new DOVE Browser

The following issues and requirements have been found.

- It is necessary for the DOVE Browser concept to use the Event Service. It has to make use of the PUSH model. The Event Channel, which is part of the Event Service, will play the role of the DOVE Agent.

- The Event Service has to use `CORBA::Any` in the EventData field to be flexible in relation to the format of the data (e.g. sensor data) conveyed by the event channel, the core of the Event Service.
- A well defined event data structure must be introduced to contain the sensor data of an aircraft.
- One of the objectives of the DOVE Browser is to be as generic as possible in Visualization Component relations. One crucial technique is the use of JavaBeans as Visualization Components. It is also important to find a generic way to connect them to Event Service.
- The customer (Boeing) asked for a MIB saving all event data information on persistent storage. Once stored this information could then be used for analysis of the system and for debugging. The stored information should be saved in a way, that is easy to debug.

It has been decided to apply an incremental development approach to build prototypes implementing the above mentioned functionality. This means the old DOVE Browser will be incrementally enhanced until the full functionality, stated here as requirements, is supported.

5.3 TAO's `CORBA::Any` type

As already mentioned the `CORBA::Any` type can be of great use for the scalability of the DOVE Browser using the event channel. The `CORBA::Any` type consists of several fields, the most important being the typecode and the value field. The value field is a pointer to `void`. The CORBA specification allows for filling the `CORBA::Any` with any type, standard CORBA types or user defined types. The value and type of a `CORBA::Any` can be set using the copy-constructor, the copy-operator, the relpace method or the "<<="-operator. The stub (or skeleton) has to ensure that the value(s) and the type(s) are properly marshalled for transmission as the skeleton (or stub) is responsible for proper demarshalling.

The TAO implementation of the `CORBA::Any` type was not fully completed at the time of the beginning of this project. As such, an effort must be made to ensure that the functionality needed for the DOVE Browser is available. Tests must be conducted using `CORBA::Short`, `CORBA::Long`, `CORBA::Sequence`, `CORBA::String` and `CORBA::Double` as types contained in the `CORBA::Any` as they are needed for the DOVE Browser. Inserting the values and types in the `CORBA::Any` using the afore-mentioned operators and extracting them using the ">>="-operator must be tested, as does using the member functions `CORBA::Any::value` and `CORBA::Any::type`. The "inout" parameter passing mechanism is best suited for testing, as it tests for proper implementation of the parameter passing mechanisms.

Chapter 6

Ideas for approaches

6.1 What is a MIB?

A Management Information Base (MIB) is a kind of database containing management information about devices. These devices can be network devices like routers, PCs, workstations or even software running on machines. So the notion of a MIB is used in a broader sense than for example in the "Simple Network Management Protocol" (SNMP). The idea of a MIB is here used for any storage of management information, such as event data or statistics of a service.

6.2 How to build DOVE MIB functionality into the Event Service paradigm.

As mentioned when giving the overview of DOVE, a part of the DOVE concept is the above mentioned MIB (Management Information Base).

1. **Having a MIB connected directly to the Event Service** The Event Service could implement the persistence storage of events in itself. The Common Object Services Description [8] describes on page 4-3 that the event channel may store events for "a specified time, passing it along to any consumer who registers with the channel during that period of time (e.g., it may keep event notifications about changes to engineering specifications for a week)." The advantages would be a simple interface and only a minimum of objects would be involved in a request for old messages, namely the Event Service and the requesting object.

2. A second idea is to have the **MIB as an object external to the Event Service**. The MIB would be a consumer for the event channel. The idea is that this external object listens to all the events, stores them on a persitent storage and retrieves them on demand (time, type and source ID). Retrieving means filtering them from the persistent storage and supplying them to the event channel again. Care has to be taken that other components are bothered/confused by the repetition of the events.

- One way to avoid this would be that the external object, the MIB, pushes the events directly to the consumer who requested the stored events.
- Another way would be that the MIB is contacted without involving the Event Service at all. This would require that the requesting objects have to know whom to ask. The object reference of the MIB could be supplied as a reaction to a request to the Event Service or, this is the better alternative to decouple Event Service and MIB, to the Naming Service.

It is necessary to ensure that the order of the events can be maintained. For example, what would happen if new events arrive while receiving old events from the MIB? In the middle of the project one customer, Boeing, stated the need for a MIB on a persistent storage, as mentioned in the Analysis and Requrirements section of this document. It has been decided to design an external MIB which receives all events from the Event Service which is used as DOVE Agent and stores them as type, member name and value triple on a persistent storage.

6.3 Considerations about the event data

The events in the OMG COS Specification of the Event Service are of type `CORBA::Any`. TAO defines events as a structure containing real-time information and a data field, which is not specified yet. This data field has to be customized to events sent. Having the concrete issue of transporting air fighter data in the Boing project, there are several possibilities:

- Use a `CORBA::Any` type as the data field, this allows the biggest variability. The event data can be changed "on the flight". The drawback is that we sacrifice a lot of performance for this variety.
- A fixed struct/union combination. The event data will be fixed and any change in this data structure will cause the need to recompile and restart the Event Service.

It has been decided to use the Any type, performance will be slightly sacrificed in favor to scalability.

6.4 Considerations about the sensor data structure

Since consumers and suppliers of the event channel are implemented in a different programming language in the case of the DOVE Browser, it is necessary to define the sensor data structure in a portable language. IDL is the obvious choice. The sensor data structure is put into the `CORBA::Any` type member of the events. The requirements are that every sensor data structure should contain statistical scheduling data beside the actual data which are either weapon status or navigation information. It has been decided to put the statistical data together in one structure. This structure is always a member of the sensor data structure. The weapon status and navigation information could be put together into a union because of space optimizations. But this is not necessary because the `CORBA::Any` type already provides the flexibility needed to save space.

Chapter 7

Specifications and Implemenations

The following sections show the incremental developement of the DOVE prototypes. Two phases of the software lifecycle are discussed below: Specification and Implementation. The previous two phases of the software lifecycle, Requirements and Analysis, were mentioned earlier in a more combined way. Documentation of testing has been omitted, not because it is unimportant, but because it is considered as part of the implementation. No implementation of prototypes was done until all parts were running properly. If the components of the produced software are to be used in practice, they will have to be thoroughly tested. The best option would be to rewrite all components to eliminate any bugs created and kept by the incremental development.

7.1 Bootstrapping protocol for the CORBA Common Object Services

7.1.1 Specification

To get the object reference of the Naming Service several possibilities exist. One of them is to give the IOR (Inter Object Reference) of the Naming Service to the application as an input parameter. Another possiblity is to define a so called bootstrap protocol for the Naming Service.

TAO implemented the bootstrap protocol for the Naming Service in the following way:

- The Naming Service listens to a well defined multicast address at a well defined port for requests. The advantage of a multicast instead of a fixed host address is that the Naming Service can reside on any machine in a domain. The other machines in this domain do not have to know on which machine the Naming Service resides, it is transparent to them.
- The Naming Service expects to receive two bytes which define in network byte order the port on which the requesting machine expects the response of the Naming Service. This response consists of the IOR of the Naming Service.
- Any object requesting the Naming Service IOR has to send a UDP packet containing a port number to the multicast address. The port number in the packet is the packet referes to the

port on which the requesting object listens for the response of the Naming Service. It then has to wait until the response arrives or a timeout occurs. The timeout has to be chosen within resonable limits. The received Naming Service IOR can then be resolved by the *string_to_object* command specified by CORBA.

By having the the Naming Service object the other COSes can be resolved easily using the *resolve* method of the Naming Service.

7.1.2 Implementation of the bootstrap protocol for the Naming Service

The implementation was done in a straight forward manner. Thre seconds was chosen as the timeout, because the network environment was small. Java supports multicast datagrams so they were used. To have the port number for replies in network byte order a little hacking had to be done because Java does not support different byte orders. After testing which order worked, the mapping from Java to network byte order was hardcoded. The results of this were:

- Name Service crashes if you give a name not available in its DB.
- The Resolve Initial Reference method has to use af fixed buffer when receiving the Name Service IOR, this could be dangerous if the IOR size is very large. This was solved immediately to: Use the maximum datagram size defined in ACE.

7.2 Testing the `CORBA::Any` type in TAO

7.2.1 Specification: How to test the `CORBA::Any` type

To test the Any type a server and a client, complying to an Any intensive interface, are built. The object implementation managed by the server provides one method, this method accepts one `inout` parameter of type Any. The method checks for the type code and selects the appropriate commands to modify the value of the `CORBA::Any`. The manipulations are very simple, so that it is easy to control on the client side if the correct value is returned by the server. The client does several requests, each time using a different type in the Any. Tests using the sensor data structure ensure that they will work when used by the DOVE Browser.

7.2.2 Results: Needed `CORBA::Any` type customizations

The tests showed that sequences and arrays did not work properly. A bug in the stub code could be found as well as a bug in the marshalling code.

- When using the any type alone as an argument, the any was not given as an argument to the `do_call`!
- When using the any type in a struct an `Any_var` was generated, which cannot be handled by the marshalling code.
- The copy operator released memory, which did not belong to the ORB. The `Release` statement in `any.cpp (operator=)` had to be checked.

The bugs found were fixed immediately by the responsible person. After the corrections, the `CORBA::Any` type is now ready to use. The sensor data structure cannot contain sequences because of the above mentioned problems. A tradeoff must be made, which is: Using a fixed number of members with names indicating the position in the sequence instead of a sequence type itself.

The Any type could have been tested more completely by also using simple in or out parameter passing mechanisms and return types, but this was left for other projects.

7.3 Testing the interoperability between Orbix, Visibroker, TAO

7.3.1 Specification

As mentioned earlier TAO is built to be used with C++, which means TAO provides only an IDL compiler to compile from IDL to C++. Interoperability with Java implementations should be possible by definition of GIOP or better IIOP, but there is no guarantee that everything works. Therefore it had been decided to test all cross compatibilities. To achieve this the Any test mentioned earlier is implemented with all three ORBs, including both the client and server sides. The port of the already existing C++ Any test to Visibroker for Java will of course be more complicated than the port to Orbix also using C++. It is expected that, using the same ORB to communicate between the client and server, no problems should occur. Therefore the x to x connections are only used to proof the correct implementation of client and server. The server implementation prints the IOR (Inter Object Reference) and the client accepts an IOR of a server. In this way it is possible to connect any client to any server.

7.3.2 Results about the interoperability

The communication between TAO and Orbix works both ways without any problems. (TAO server and Orbix client, as well as TAO client and Orbix server). Also the communication between Orbix and Visibroker for Java show no problems. The communication between Visibroker for Java Version 2.5 and TAO has problems, though. After switching over to Visibroker for Java Version 3.1 the communication works without any problems. The reason for this flaw is not known, it is assumed that it is an incompatibility in the implemented IIOP protocols. In March 1998 the new version of Visibroker for Java came out, Version 3.2. A bug has been found a bug in this version when using more than one child struct inside a parent struct. The marshalling code does not marshall these kind of structs properly, perhaps optimizations might have been improperly implemented.

7.4 Testing the access to the TAO Event Service using Visibroker

7.4.1 Specification

The key to using the TAO Event Service is the IDL specifications of the various objects involved in the communication. The involved objects are the Naming Service, Scheduling Service and Event Service. The IDL specifcations have to be compiled to Java code in order to allow the DOVE Browser to connect to the three services. It has been decided to use the already mentioned PUSH model of the Event Service to achieve event delivery between suppliers and consumers.

7.4.2 Result

Initialization is tedious, a helper class is needed and/or initialization of the "data_" field should be separated. The dependency information supplied by a supplier or a consumer of the Event Service contains also events. This fact makes some problems in situations where no event is wanted to be sent by the supplier. So in this case the event information has to be initialized, though actually no event data is available. A redesign of the dependency information could resolve these kind of situations.

7.5 Testing the sensor data structure as content of an `CORBA::Any` type

7.5.1 Specification

Because the sensor data structure is not just a simple type it must be proven that TAO and Visibroker are able to exchange this structure in an `CORBA::Any`. The already mentioned Any test will be enhanced to not only test simple types and arbitrary structs, but also to test the two concrete structures: Navigation information structure and Weapons status structure.

7.5.2 Implementation and Results

The implementation is no big effort. Some dummy values are assigned to the members of the sensor data structure. The Any test client then calls the Any test server with the sensor data structure inside the Any type. On the TAO client side the `replace` method is used because it is assumed that it is implemented properly. The Visibroker server side extracted the sensor data structure with the help of the Helper classes, which are automatically supported by the Visibroker Java to IDL compiler.

7.6 Enhancing the DOVE Browser with the Event Service

7.6.1 Specification

As mentioned in the Analysis and Requirements the DOVE Browser has to be enhanced to use the Event Service and the events must transport the sensor data in their `CORBA::Any` type member. After completing the various tests to assure that the needed functionality, `CORBA::Any`s and interoperability, is available the DOVE Browser can now be redesigned. The old DOVE Browser, the so called demo, uses sockets to communicate between a data generator and a viewer. Raw bytes are sent between them. The old viewer consists of several visualization components: a graph component, a list-panel to display the weapon status and an artifical horizon. The way the viewer is build makes it possible to have several of these components running at the same time connected to different data sources. Thus it is possible to bring up graphs dynamically, but the set of viewers is hard coded. One goal is to have the components running as JavaBeans. Doing so allows to be more dynamic and to facilitate the connection between visualization components and specific metrics supplied by a event channel consumer, which is part of the viewer. This will be a topic of further enhancement.

The whole idea with the Event Service is to have several suppliers (e.g. sensors), several consumers (viewers) and the possibility to filter and correlate events (e.g. sensor information).

The monitored object has to be a supplier of events in this concept, either directly or indirectly with a proxy in between. In the case of this DOVE Browser, the monitored object is a dummy supplier supplying only dummy statistical data, weapon status and navigation information.

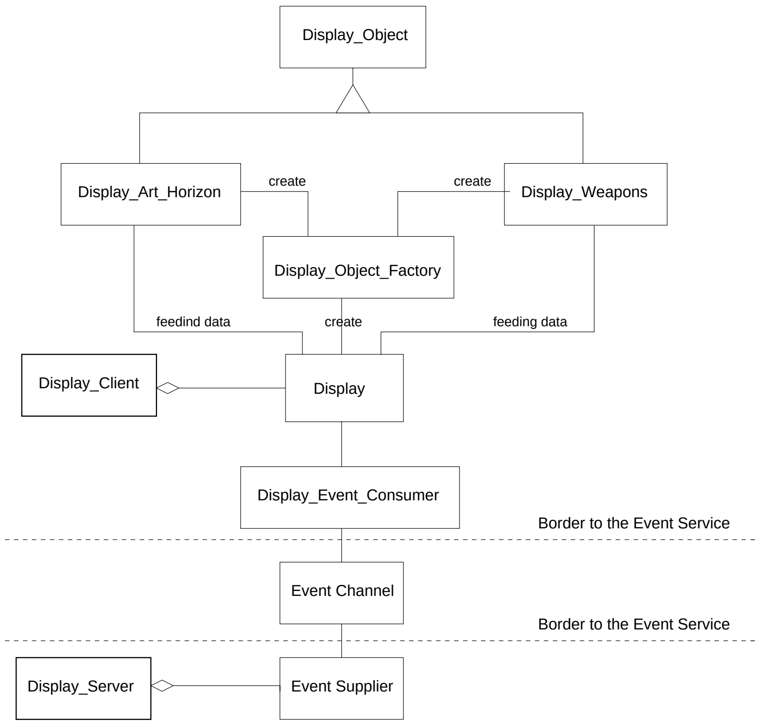

Figure 7.1: Object Diagram of the old DOVE Browser using the Event Service

Figure 7.1 shows the relationships of the Event Service parts to the rest of the DOVE Browser. The Event Channel pushes events to the Event Consumer which then updates information inside the DOVE Browser.

7.6.2 Implementation

The implementation of the enhancement of the DOVE Browser using the Event Service requires translating the IDL definitions of the Naming Service, Scheduling Service and Event Service to Java. The objects are now accessed by Java as it was done by TAO objects. The main problem now is to figure out the mapping.

7.7 Putting the DOVE Browser into a web browser as an applet

7.7.1 Specification

As mentioned in the analysis section, a main concern is to be able to have the DOVE Browser running in a web browser with the event channel being on any machine in the internet and the data source, the event channel supplier on a third location or on one of the before mentioned locations. Since Netscape has provided an ORB built-in in their Netscape Communicator for quite some time, the vision of having an applet using CORBA running in a web browser is a reality. The ORB provided by Netscape is the Visigenic ORB Visibroker for Java version 2.5 assuming Netscape Communicator 4.04. Since the DOVE Browser is written using Java 1.1 and Visibroker 3.1, the applet can not run in the regular Netscape Communicator 4.04, which does not support Java 1.1 at this time, only Java 1.0 (assuming February 1998). It is very important for the DOC group to be able to show the customers, e.g. Boeing, the DOVE Browser running in a web browser. Therefore several tests are made to find out the part not working in the communication between TAO and Visibroker 2.5. The final result is, that there must be a flaw in the marshalling code of Visibroker 2.5. At the time of the tests (February 1998) an extension of Netscape Communicator was available supporting Java 1.1 and Visibroker 3.1. This could solve the problems, but the way to install the extension is very complicated and therefore not reasonable for the use by the customers. It has been decided to wait for an already properly set up Netscape Communicator instead of spending time trying to fix the not provided functionality of a commercial product.

7.8 Enhancing the DOVE Browser with JavaBeans functionality

7.8.1 Specification

One basic idea of DOVE is to be very generic in displaying information. This requirement lead to the idea of using JavaBeans as DOVE Visualization Components, as mentioned in the overview of DOVE. To create a Java Bean no special classes are needed. The only thing that has to be obeyed is that the Java Bean should depend on as few as possible assumptions about the environment it will use. The method names should also obey a certain naming convention so that reuse and configuration are as easy as possible. The naming convention is quite simple: A method that sets any parameter of an Java Bean should be named as `set<parametername> (<parametername>)` and a method that gets some parameter of a Java Bean should be named as `<parametername> get<parametername>`. For further information about JavaBeans see [16].

Having a generic "front end" will not be enough when not being generic inside the DOVE Browser, too. Therefore some considerations have been made and the final idea is to use an Observer pattern. The two main components of an Observer pattern are Observables and Observers. Observers register with one or more Observables, which then inform one or more registered Observers about changes as soon as its own state changes. So when this pattern is mapped to the DOVE Browser the metrics supplied by the event data become Observables, which means that each metric has a corresponding Observable. The visualization components become Observers. Now two things have to be done to have this concept working. One is to demultiplex the event data into Observables. This means several event data structure members could form one metric and therefore the metric has to be computed first.

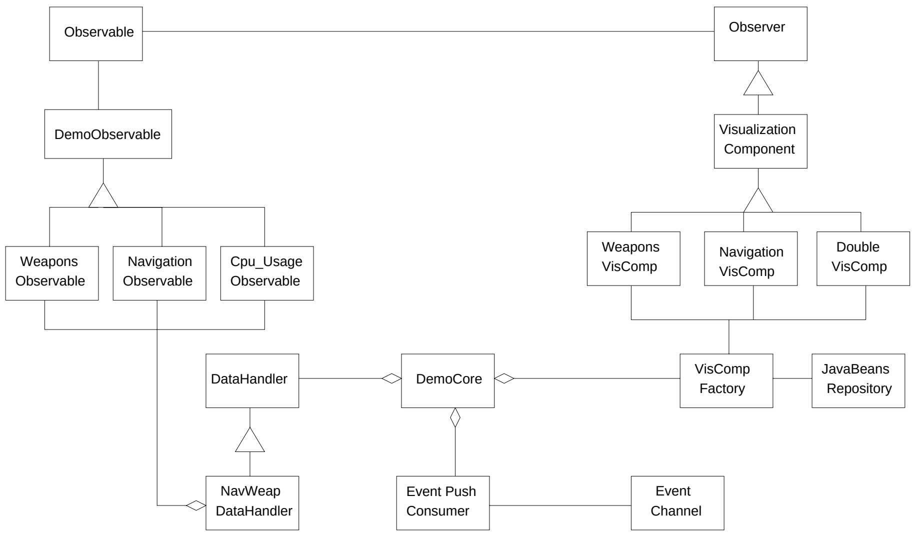

Figure 7.2: Object Diagram of the DOVE Browser

Figure 7.2 shows the architecture of new DOVE Browser using the generic JavaBeans. As it can be seen all concrete Observables inherit from the Observable object and all concrete Visualization Components (JavaBeans) inherit from the Observer object. Because communication between Observables and Observers is done assuming the base object (Observable object, and Observer object) functionalities only, indepencence between concrete Observables and Observers is guaranteed.

The NavWeapDataHandler object inherits from a neutral DataHandler object. This way it can be assured that the PushConsumer object is independent from any concrete event data structure. So the DataHandler object defines the basic interface.

The functional model of the new DOVE Browser is simple, Figure 7.3. The `DemoCore` object tells the `VisCompFactory` to create a new Visualization Component with the properties

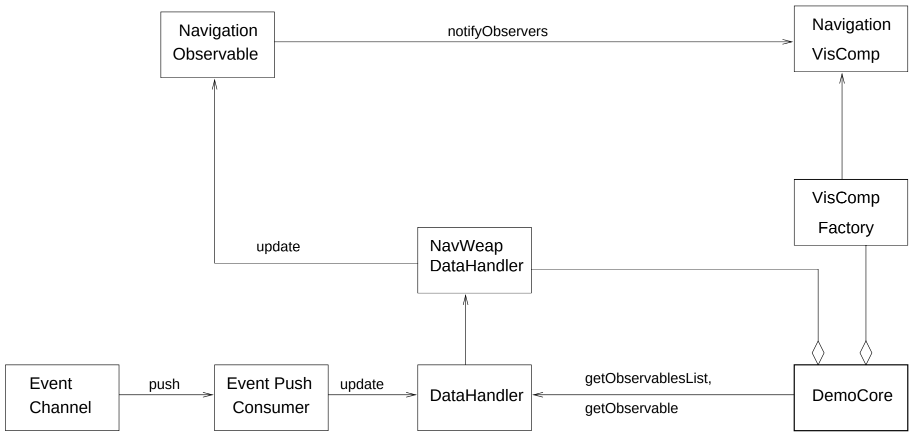

Figure 7.3: Functional Model of the DOVE Browser

necessary to display the chosen `Observable`. After that the `DemoCore` is responsible for connecting `Observables` with `Observers` (Visualization Components). Connecting `Observables` is done through the `DataHandler` which generic interface allows the `DemoCore` to ask for a list of Observables and then to pick one specifically to connect it with an appropriate Visualization Component.

Updates of the Visualization Components take the following flow of control: The Event Channel pushes events to the `PushConsumer`. The `PushConsumer` forwards the event data field to the Data Handler, which is actually an concrete Data Handler (e.g. `NavWeapDataHandler`). This Data Handler then demultiplexes the event data structure into several metrics. The metrics influenced by this recognize that the value they contain has been changed and trigger a notification event to their Observers. This event contains then an object with the changed data. The observers read the data field of the notification event and update their graphical front end.

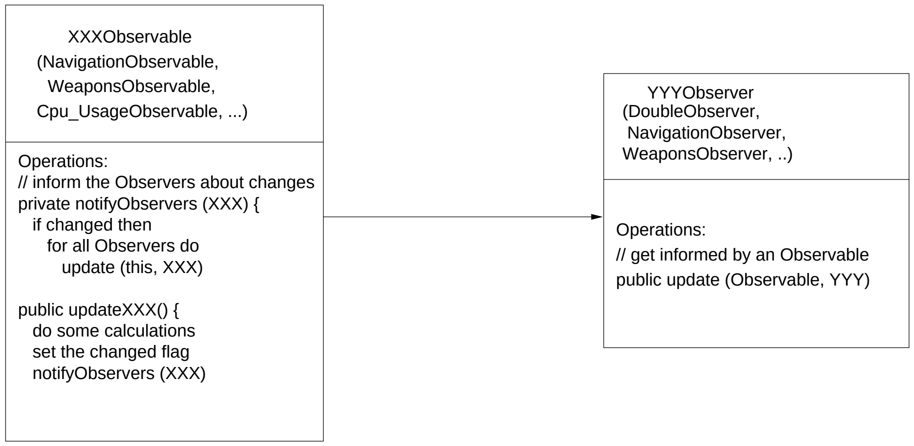

Figure 7.4: Interface description between Observables and Observers

Figure 7.4 shows the interface between Observables and Observers. The Observables get called by their update method. The method does some calculations, sets the changed flag and calls then the method notifyObservers with a reference to an object. The object is in this use-case the changed metric, e.g. a double, a long or a Navigation struct. The method notifyObservers checks the changed flag and calls the update method on all registered Observers. The called Observers read then the object out of the parameter and update their display.

Observers, Observables pairs can be categoried into two classes depending on the data they are responsible for. The first category uses only build in Java types like `Double` or `Int`. The second category uses structs containing more sophisticated information like Weapon status or Navigaton information. It is obvious that the first category is more generic than the second one. The concept of JavaBeans is in the second category slightly lost because there is a heavy dependence between Observables and Observers through the Weapon status structure (short only Weapons structure) or the Navigation information structure (short only Navigation structure). The following should give an example: `Cpu_UsageObservables, JitterObservables` and `OverheadObservables` are all displayable by a `Double Visualization Component`. Whereas the `NavigationObservable` can only be connected to the `NavigationObserver`.

7.8.2 Implementation

As already mentioned in the specifiation, JavaBeans have been used as generic DOVE Visualization Components. It was mentioned in the specification, though it is clearly an implementation decision. But because the concept of JavaBeans is very basic for this project it has been decided to mention it anyway.

Java provides the Observer pattern already. This means two classes called "java. util. Observer" and "java. util. Observable" are part of the Java Development Kit. These classes are used to implement the afore mentioned Observer pattern.

The implementation has been done straightforward to the specification. The objects are directly mapped to Java classes. The part of the DOVE Browser connecting to the Event Service could be taken without many changes from the previous prototype. The Visualization Components have to be customized in order to fulfill the JavaBeans requirements.

7.9 Building a Management Information Base for DOVE

7.9.1 Specification

The Analysis and Requirements phases stated already the need for a Management Information Base, refered to as MIB from now on. The Ideas section of this document discussed several system architectures of such a MIB. After a specific request for a MIB by Boeing at a meeting it has been decided to choose the external object approach to implement the MIB. Which means an external object is connected via Event Consumer to the Event Channel, which is serving as the DOVE Agent. The MIB listens to all events sent on the Event Channel and stores the event data, in a format similar to the format used for declarations in C++, on persistent storage.

The event data is contained in an `CORBA::Any` and no assumptions can be made about the data inside. The MIB has to be able to handle every kind data inside the event data. Therefore the MIB has to analyse the data contained in the `CORBA::Any` carefully. The `CORBA::Any` could contain a struct, a double, long and so on. Several layers of types could be contained, e.g. a struct, containing a struct, containing a string. Having several layers requires recursive analysis of the

`CORBA::Any`. It has been decided to use the notion of a tree for this purpose. This means the nodes contain the type and value information and links represent a parent-child relationship (a struct as parent and and the members of the struct as children).

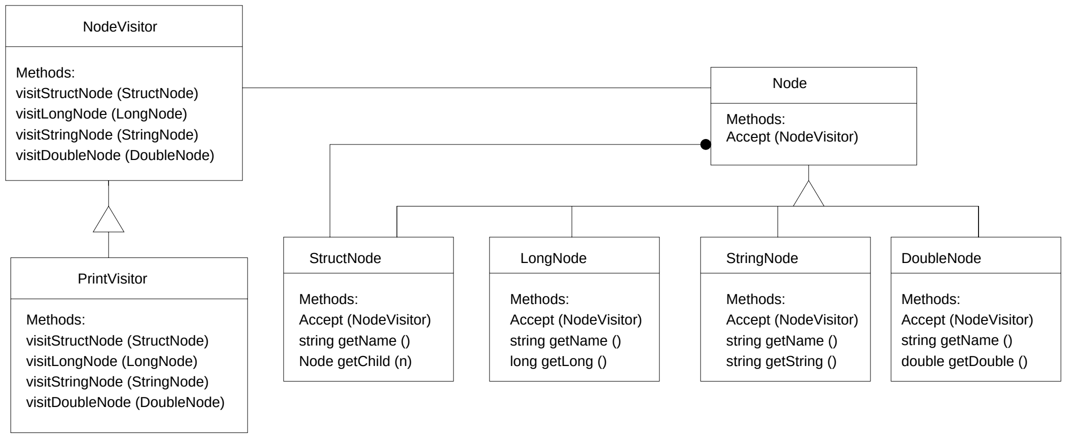

Figure 7.5: Visitor and tree representing the content of a CORBA::Any

See Figure 7.5 for an object diagram of the the above mentioned nodes. So all types of nodes (`StructNode, DoubleNode, LongNode, ULongNode, StringNode..`) inherit from `Node`. It has been decided to use a Visitor pattern [3] to traverse the tree.

All Visitors inherit from `NodeVisitor`. which provides the basic functionality to traverse a tree. The visitor `PrintVisitor` accepts a file name and a tree as input. It prints the value of the tree in a format similar to the declaration format in C++.

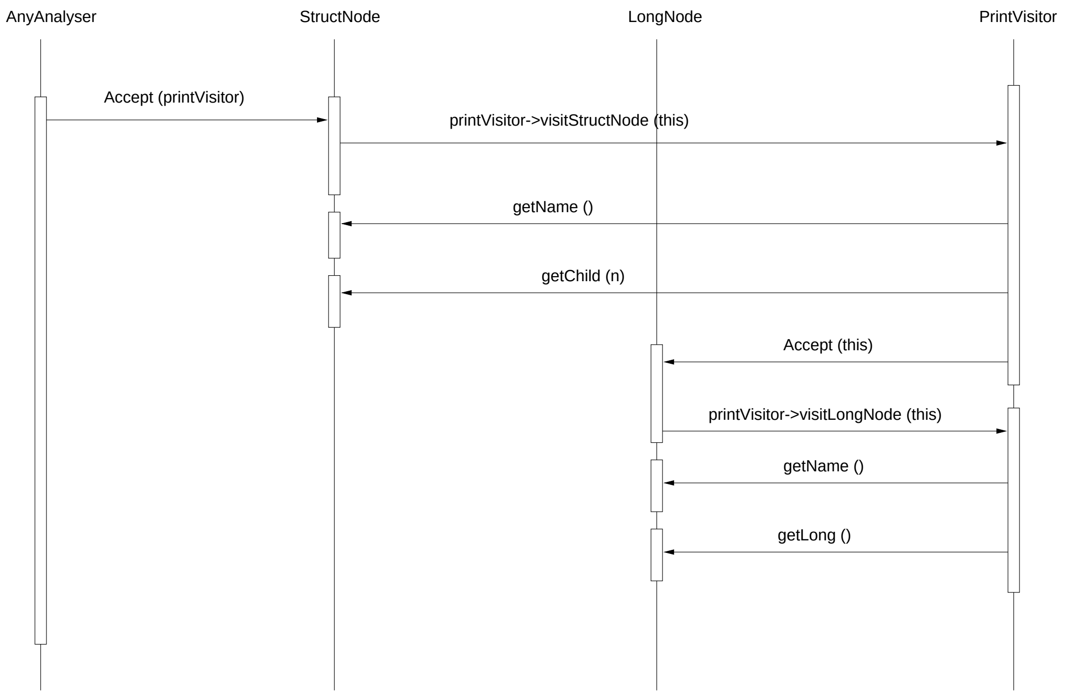

Figure 7.6: Functional Model of the Visitor Pattern

Figure 7.6 shows the relation between the objects and methods. The `AnyAnalyser`, creates a tree out of the type code and value information of the `CORBA::Any`. It instantiates the `PrintVisitor` and calls the root of the tree with a reference to the `PrintVisitor`. The node, in this case the `StructNode`, calls then the `PrintVisitor`, which in turn calls the StructNode again to obtain the properties of it. In this case the properties are the name and references to children. Now, the advantage of the Visitor pattern is obvious: The nodes themselves do not depend on what is done with the properties, if they are used for further computation, just printed or further traversed. This independence assures that new Visitors can be added without any change in the node objects. To go on in the explanation of the example, the `PrintVisitor` uses the references to the children of the `StructNode` to further traverse the tree. The first child is a `LongNode`, it gets called by the `PrintVisitor` on its `Accept` method and replies with the `visitLongNode` method call of the `PrintVisitor`. In turn the `PrintVisitor` gets the properties of the `LongNode` and prints them. The LongNode does not have to care about what is done with its properties.

A wrapper called `AnyAnalyser` wraps the building of the trees with the `PrintVisitor` together, so that the `AnyAnalyser` can be seen as one object to which `CORBA::Any`s can be given. The content of the `CORBA::Any` will be written to a file.

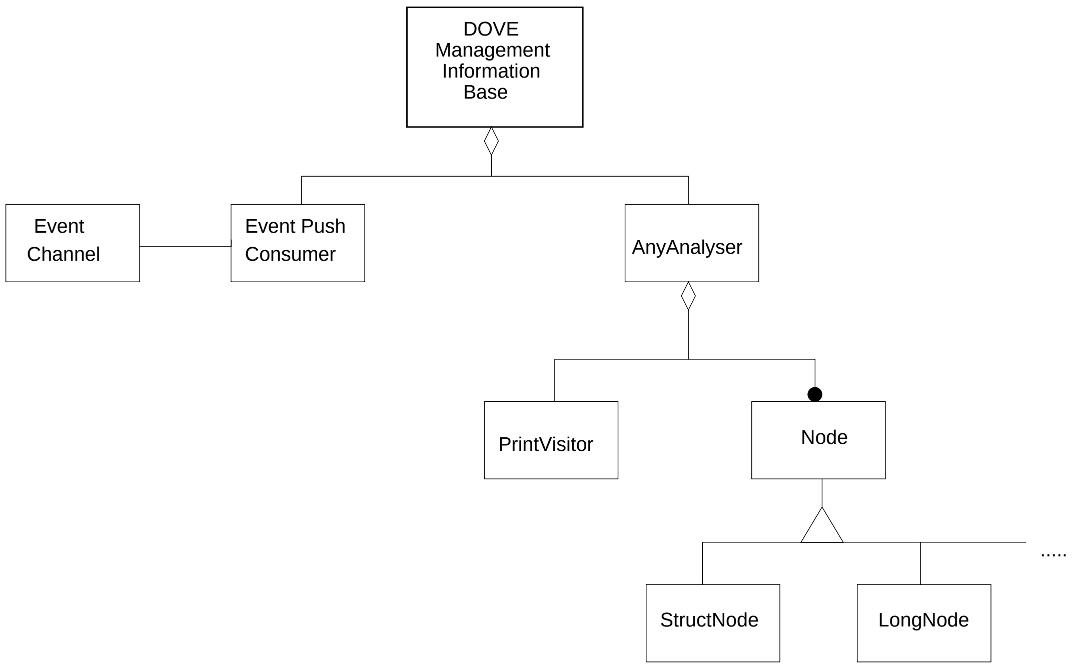

Figure 7.7: Object Diagram of the DOVE Management Information Base

The MIB application itself, see Figure 7.7, consists out of a `Event Push Consumer` and an `AnyAnalyser`. This way the two basic functionalities are separated as much as possible. Which means a change in the `Event Push Consumer` does not influence the `AnyAnalyser` and vice versa. Figure 7.7 shows the relations between the objects.

7.9.2 Implementation

Because the MIB does not need a user interface it has been decided to implement it using C++ and TAO. The Event Push Consumer part can be taken from the previously written prototypes for the Event Service functionality. The afore mentioned objects like `Node`, {`Struct, Double, Long` .. }`Node, NodeVisitor, PrintVisitor` and `AnyAnalyser` are mapped directly to C++ classes. The MIB accepts two parameters: a name for the file on a persistent storage and a number of events. The number of events tells the MIB how many events have to be monitored until it shuts down itself. This restriction has been made because limited space on the persitent storage has to be assumed.

The `AnyAnalyser` has detailed knowledge about the internals of a `CORBA::Any`. The `CORBA::Any` in TAO holds the value in two diffent ways, depending on if the ORB owns the data or not. If the ORB does own the data, a simple pointer to an allocated memory area can be retrieved. If the ORB does not own the data, it has to be copied into a newly allocated memory area and decoded. Explaining decoding is beyond the scope of this document.

Analysing the `CORBA::Any` involves checking the type code information and creating a tree node with a proper pointer to the actual memory location. Then skipping a number of bytes defined by the type code size and repeatedly applying this sceme.

Chapter 8

Common Object Service: LifeCycle Service

8.1 Overview

The COS specification [8] describes the LifeCycle Service in the following way:

Lifecycle services define services and conventions for creating, deleting, copying and moving objects. Because CORBA-based environments support distributed objects, lifecycle services define services and conventions that allow clients to perform lifecycle operations on objects in different locations.

Clients are not allowed to create objects themselves by specification. The objects have to be created by factories. Factories are dependent on the objects they create. A factory has to know how to initialize and how to support resources. The COS specification introduces the concept of `Generic Factory` in order to introduce a generic way of creating objects. The `Generic Factory` interace can be either supported directly by a concrete and object implementation dependent factory or by a separate factory implementing only the interface. The separate object then needs to have a way to find concrete factories to create the requested object. Factories implenting the `Generic Factory` interface must implement a generic operation, named `create_object` Instead of invoking an object specific operation on a factory with statically defined parameters, the client invokes the `create_object` operation whose parameters can include information about resource filters, state initialization, etc.

In many environments resources are scarce or certain contraints have to be fulfilled by a factory. A constraint could concern the architecture like intel, sun, etc. As a standardized interface for factories the Generic Factory interface was introduced by OMG as part of the LifeCycle specification.

Figure 8.1 shows how the `Generic Factory` interface can be used to implement a “creation service” which distributes creation requests using the “Criteria” to a set of concrete factories.

Objects who want to comply to the lifecycle specification have to support the `LifeCycleObject` interface. The `LifCycleObject` interface is shown in Figure 8.2.

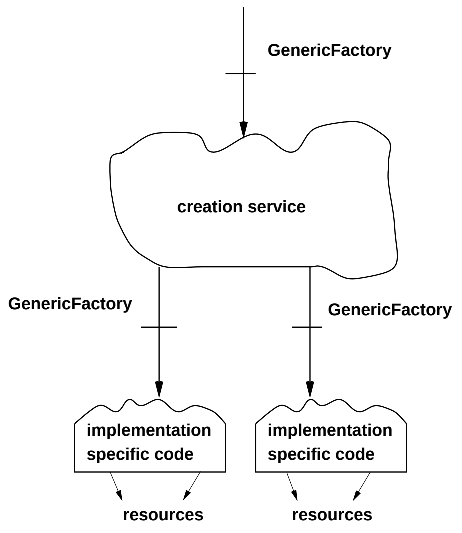

Figure 8.1: GenericFactory interface

```
interface LifeCycleObject {
   LifeCycleObject copy (in FactoryFinder there,
                         in Criteria the_criteria)
                         raises (NoFactory, NotCopyable,
                                 InvalidCriteria,
                                 CannotMeetCriteria);
   void move (in FactoryFinder there,
              in Criteria the_criteria)
              raises (NoFactory, NotMovable, InvalidCriteria,
                      CannotMeetCriteria);
   void remove ()
                raises (NotRemovable);
};
```

Figure 8.2: IDL definition of LifeCycleObject

A client that wishes to remove an object, invokes the objects `remove` operation. A client that wishes to move or copy an objects issues a `move` or `copy` request on an object. The user of the lifecycle service must be aware of the consequences for other clients of the moved object.

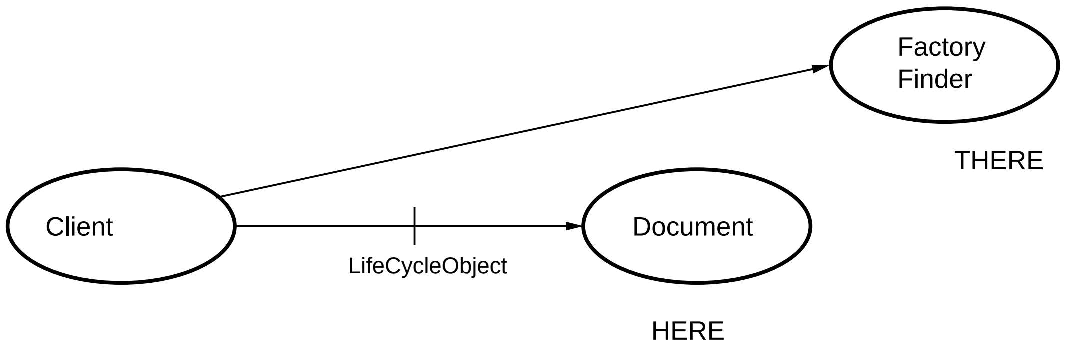

Figure 8.3: Lifecycle services define how a client can move or copy an object from here to there

In the example of Figure 8.3, client code would simply issue a `copy` request on the document and pass it an object supporting the *FactoryFinder* interface as an argument.

Factory Finders are used to find factories in environments. They support an operation, `find_factories`, which returns a sequence of factories. Clients pass factory finders to the `move` and `copy` operations, which typically invoke the `find_factories` operation to find a factory to interact with.

The client of the `GenericFactory` interface invokes the `create_object` operation and can express criteria for creation. The `GenericFactory` then forwards the creation request to a more concrete factory. There exists also the possiblity that several `GenericFactories` are invoked until finally a conrete factory creates the requested object. However, all of this is transparent to the client.

Figure 8.4 explains the `create_object` operation, which accepts two parameters. The first is the key of the factory to be used and the second is a constraint expression.

8.2 Context

The outcome of this project will serve as research object and as an example of the LifeCycle Service as part of the TAO package.

The main goals of this project are:

- Providing an example for the use of the COS LifeCycle Service.

```
Object create_object (in Key key,
                      in Criteria the_criteria)
                      raises (NoFactory,
                              InvalidCriteria,
                              CannotMeetCriteria);
```

Figure 8.4: IDL definition of the create_object operation of the `GenericFactory` interface

- Get first some experience and later ideas for approaches and optimizations.
- Provide, if possible, support for Generic Factories and Factory Finders.
- Build a generic creation service, named “lifecycle service” and try to factor out its code to have it as separate component available for all applications.

8.3 Starting point: The Quoter example - Version 1.0

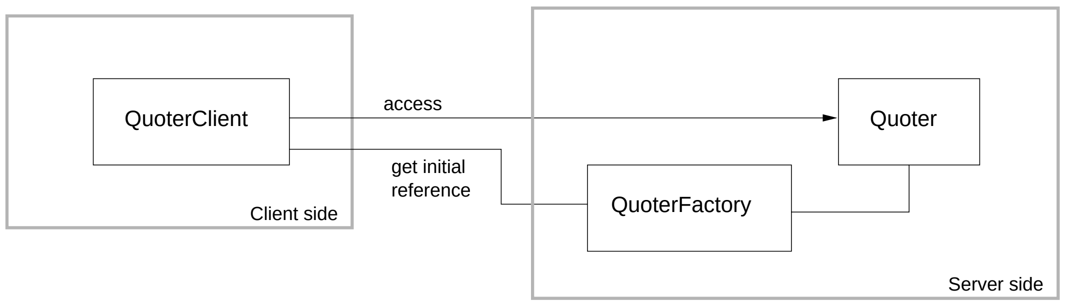

Figure 8.5: Object Diagram of the first Quoter example

Figure 8.5 shows the first version of the Quoter example. It consists out of a Quoter client, a Quoter factory and an actual Quoter object. The Quoter object itself is not very sophisticated, it just offers a method called `get_quote`, which gives the quote of the stock specified. So the focus will not be on the Quoter object itself but on the lifecycle of it.

8.4 Analysis and Requirements

The Quoter example version 1.0 does not support any LifeCycle operations beside the `create_quoter` operation of the Quoter Factory to get a reference of the actual Quoter. In order to achieve the above mentioned goals the following schedule has been devised:

- The Quoter object should support the lifecycle operations.
- A Factory Finder should be provided for finding Quoter Factories.
- Support for Generic Factories and Factory Finders should be provided.
- Support the “Lifecycle Service”, a very generic creation service, which will be explained in detail later.

As in the DOVE project, this project will use incremental developement, too. This means, that the prototypes are incrementally enhanced to finally achieve the full functionality.

8.4.1 Analysis for a Generic Factory and a LifeCycle Service

The implementation of a Factory Finder is pretty straightforward. One way to find factories in a certain environment is to use the Naming Service. This is the easiest way to find a object reference to a suitable factory.

In the case of a Generic Factory, which is supposed to forward creation requests to more specific and concrete factories this would work too. But the disadvantage using this sceme is that the `Criteria` are worthless in this case. The Naming Service cannot use the `Criteria` to select any specific factory. Remembering other COSs, there is the idea of a Trading Service. After looking in detail over the `Criteria` specification [8] it was found out that the criteria language is a subset of the Trading service constraint language. The trading service provides everything needed to build a generic creation service. Also the Trading Service supports registering objects with describing properties and searching for objects, which fulfill the specified constraints. Though this is only part of the Trading Service functionality, it is exactly what is needed by a Generic Factory to make use of the `Criteria` specified. The idea is to force all concrete factories to comply to the Generic Factory interface and make them registering with the creation service, our lifecyle service. So all factories must know how to handle the generic `create_object` call and how to extract the information specified in the `Criteria`.

Two kind of factories, implementing the `GenericFactory` interface, are conceivable: A simple Generic Factory using no registering of concrete factories using simple Naming Service lookup or a sophisticated creation service like the LifeCycle Service shown in Figure 8.6. The LifeCycle Service is supporting an interface to register factories implementing the `Generic Factory` interface.

Figure 8.6 also introduces the idea of having a concrete factory which forwards the creation request to the LifeCycle Service because it cannot handle the create request. The reason for that might be scarce resources for example. The `create_object` request from a client, or a forwarded request from a concrete factory, then issues a query on the best matching concrete factory. For more details see the specification [8].

8.5 Specifications and Implementations

8.5.1 Specification: Adding a Factory Finder

The addition of the COS LifeCycle operations was done by an other team member. The Quoter must inherit from the *LifeCycleObject* and implement the lifecycle operations. So the starting point is a Quoter object, which inherits from the LifCycleObject, and a concrete Quoter Factory, specialized to create only Quoter objects. Both objects are registered with Naming Service, so that the Naming Service can be used to find either one of them.

Figure 8.7 shows the relations between the Quoter Factory Finder, the Quoter Factory and the actual Quoter. It can be seen that the Factory Finder introduces an additional level of indirection. The additional level of indirection cannot be prohibited following the COS specification.

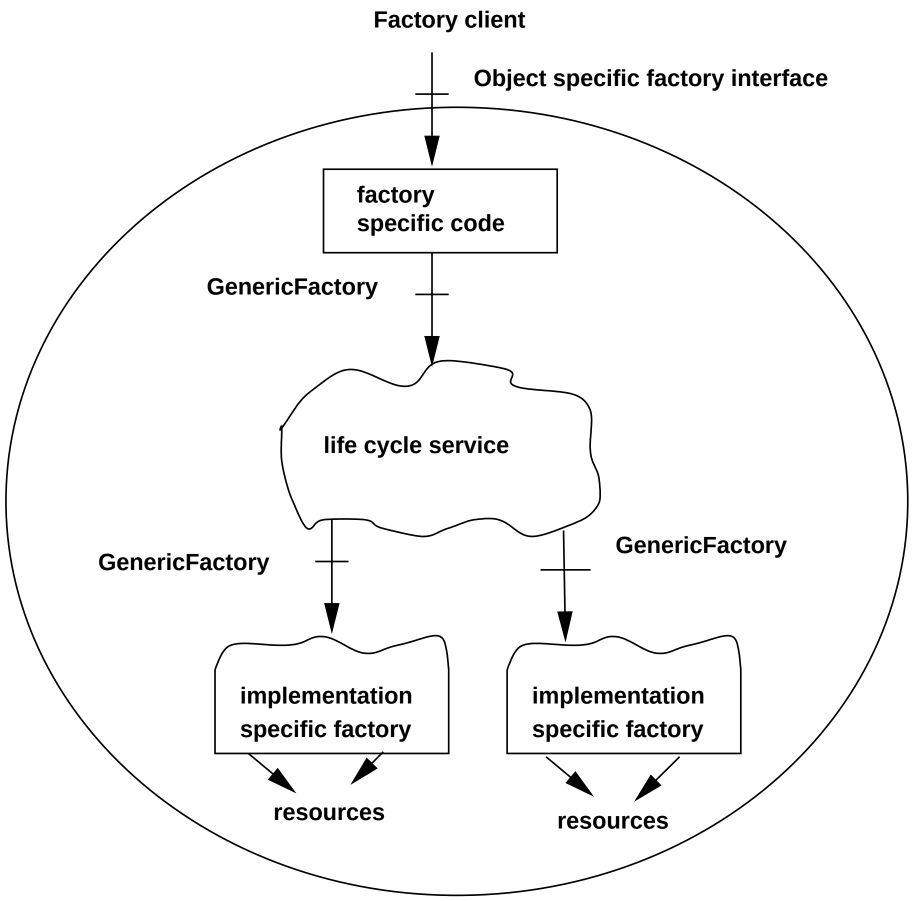

Figure 8.6: Generic Factories - lifecycle service

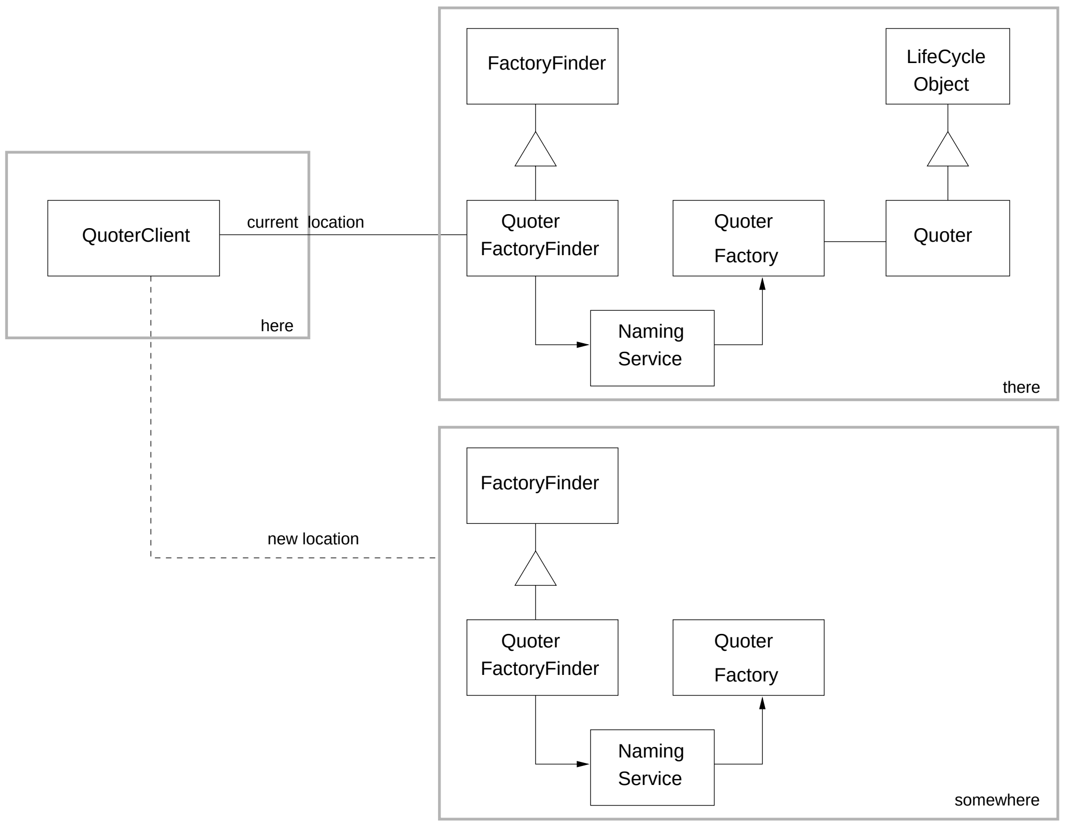

Figure 8.7: Object Diagram of the Quoter example with a Quoter Factory Finder

8.5.2 Specification: Adding a Generic Factory

As already stated in the analysis and requirements part, one way to implement a Generic Factory is to use the Naming Service to find a proper concrete factory. This is the first step towards the full functionality. In more detail this means, the first prototype of a Generic Factory for the Quoter example will inherit from the LifeCycle Service `GenericFactory`. The `create_object` method uses the Naming Service to find a factory with the proper name, like `Quoter_Factory`. As it can be seen, this approach is not very generic and not very dynamic. The `Criteria` is not used. The implementation should be pretty straightforward. How to access the Naming Service is a well known issue.

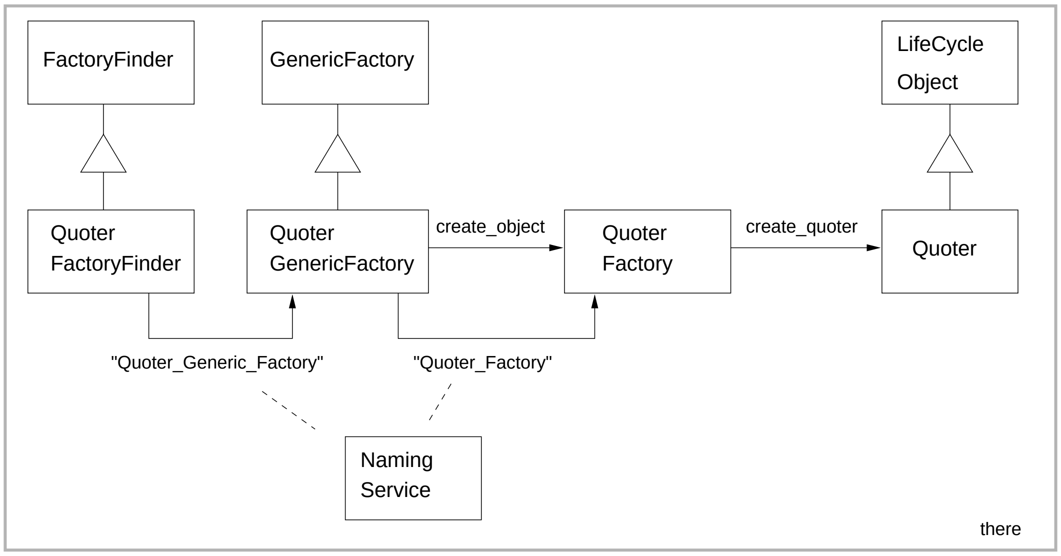

Figure 8.8: Object Diagram of the Quoter example with a Generic Factory

Figure 8.8 shows how the Generic Factory fits into the Quoter example.

8.5.3 Specification: Adding a LifeCycle Service

The LifeCycle Service implements the `Generic Factory` interface and uses the Trading Service as lookup engine. As already stated, the Trading Service provides all functionality needed:

- An interface to create a new "type". "Types" in Trading Service terminology are descriptions of objects, which are traded.
- An interface to register concrete factories with the lifecycle service, which is in Trading Service terminology "exporting".
- An interface to query object, depending on constraints/criteria.

The Trading Service is used as a colocated object, which means, no networking overhead is introduced. Trading Service properties have to be specified in order to establish criteria for selection. For the first version of such the Lifecycle Service a fixed set of criteria is assumed. The properties

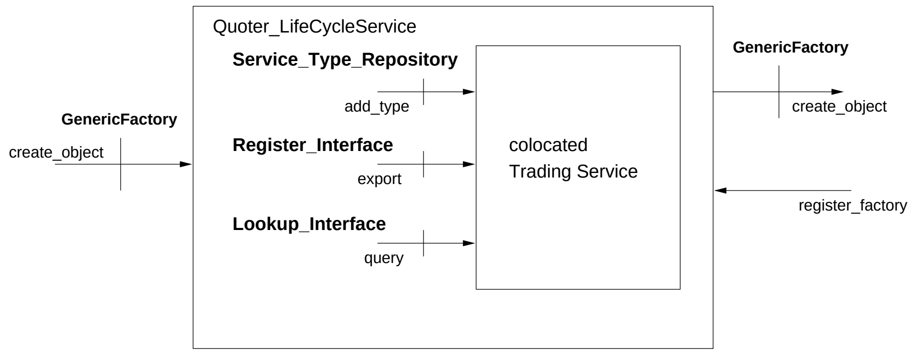

Figure 8.9: Object Diagram of the Quoter LifeCycle Service

are: name, description and location. These properties can be easily exchanged for more sophisticated ones, as soon as the Lifecycle Service is properly running.

Figure 8.9 shows the LifeCycle Service in more detail with the Trading Service as a colocated object.

8.5.4 Implementation

This is part of the documentation provided with the implemenation of the Quoter example. Several processes exist, each holding one or two objects.

The following processes exist:

- server
- Factory_Finder
- Generic_Factory
- Life_Cycle_Service
- client

and the following objects exist:

- `Quoter` (inherits from `CosLifeCycle::LifeCycleObject`)
- `Quoter_Factory` (dependent on Quoter)
- `Quoter_Factory_Finder` (inherits from `CosLifeCycle::FactoryFinder`)
- `Quoter_Generic_Factory` (inherits from `CosLifeCycle::GenericFactory`)
- `Quoter_Life_Cycle_Service` (inherits from `CosLifeCycle::GenericFactory`)

Server

The server process contains two kind of objects: `Quoter` and `Quoter_Factory`. A `Quoter` is a very simple Object, supporting only one method `get_quote`. The focus is not on building a sophisticated object, but on showing how lifecycle policies work. The object `Quoter_Factory` serves as a factory for `Quoter`s.

Factory Finder

The COS specification introduces the concept of a Factory Finder, which is capable to find proper factories. The Naming Service is used as lookup-mechanism. A reference to the `Factory_Finder` is passed as parameter to every `copy` or `move` request of an object supporting the `LifeCycleObject`, like the `Quoter` does for example.

Generic Factory

This process holds the object `Quoter_Generic_Factory`. The `Quoter_Generic_Factory` implements the `GenericFactory` interface introduced by the COS specification. It forwards `create_object` requests to more concrete factories, e.g. the `Quoter_Factory`. The concrete factories are found via Naming Service.

LifeCycle Service

This process is very similar to the Generic Factory process. It also supports an Object, which conforms to the `GenericFactory` interface. The `Quoter_Life_Cycle_Service` conforms to the idea of a Lifecycle Service" as it is introduced by the COS specification. The `Quoter_Life_Cycle_Service` is neutral against the `Quoter` example. It is not dependent on the interfaces defined in the Quoter IDL file. It uses only interfaces defined by the *CosLifeCycle* module. The implementation uses the COS Trading Service to manage registered Generic Factories, as the `Quoter_Generic_Factory` for example. A lookup on the Trading Service is performed when a `create_object` request is invoked on it.

Client

Creates a `Quoter` using the `Quoter_Factory_Finder` to find a suitable factory. After that, the copy method of `Quoter` is invoked to copy the `Quoter` to an other location, which is in this example the same location, but that does not matter so much. The concept is important in this example.

The objects are invoked in the following order: client → `Quoter` → `Quoter_Factory_Finder` → `Quoter_Life_Cycle_Service` → `Quoter_Generic_Factory` → `Quoter_Factory`

Chapter 9

Conclusion

The concept of DOVE proved to be very flexible and generic. It has been easy to extend and implement the concept. CORBA technology and especially TAO have proven to be the right frameworks to do such a task. The CORBA::Any type allowed to have very generic interfaces, which are still type-safe.

Many design lessons have been learned and a lot of experience gained. Implementing a COS service helped to understand the absolute need for exact and complete specifications, as it is good software engineering practice.

9.1 Future work

9.1.1 Future of the DOVE demo

The DOVE demo will serve as a showcase and starting point for many new projects. One these projects will be the monitoring of the Audio/Video streaming service, which is part of TAO. An other project will be to display debugging information of the Event Service used by Boeing.

9.1.2 Future of the LifeCycle Service

The move operation has been implemented using the location forwarding technique.

After this document was closed, TAO has been enhanced to support location forwarding. TAO provides location in two different ways, one is through replacement of the object with an object, that causes a GIOP Location Forward reply, the other approach is to use a so called "servant locator" (part of the Portable Object Adapter Specification) and have it to send GIOP Location Forward reply. For details, please refer to the TAO documentation at

http://www.cs.wustl.edu/ schmidt/ACE_wrappers/TAO/docs/releasenotes/index.html.

Chapter 10

Appendix

10.1 Presentations and talks

- March 17 1998: Presentation of DOVE to people of Boeing Research Laboratory at Boeing, Saint Louis.
- April 20 1998: Presentation of DOVE to people of Siemens Research Laboratory at the DOC laboratory, Washington University, Saint Louis.
- April 21 1998: Presentation of the LifeCycle Service to people of Siemens Research Laboratory at the DOC laboratory, Washington University, Saint Louis.
- June 15,16 1998: Presentation of the DOVE demo to people of Motorola Research Laboratory at Schaumburg, near Chicago, Illinois. The DOVE demo is going to be modified and will serve as a prototype of a feasability study.

10.1.1 Learned design lessons

- Patterns are a valuable tool for software development. They make resusing and extending code easier. Though they should be applied carefully [1], to extensive use of one design pattern - “Golden Hammer” - can lead to poor software quality again.
- Exception handling is one of the most valuable concepts. Doing the job proper shortens the time of debugging dramatically.
- Memory management is still one of the main problems in dynamic applications. Design Patterns should also support proper memory management.
- Incremental development has been proven to be successful for developing prototypes.
- Proper exception handling pays off many times later during the use and while testing. Points of failure can quickly be tracked down to the source.

- There are many ways to do a specific task, but there is always a proper and efficient (real-time constraints) way of doing things. Using macros, using inlining, using as little as possible virtual functions and much more, see ACE and TAO for hundreds of examples.
- When handling complex tasks, solve first the simple subtasks, then combine them to a complex solution.

Acknowledgments

Thanks to the members of the DOC group for their great support. Thanks especially to Professor Douglas C. Schmidt giving me this great opportunity to learn and experience so much about object-oriented and CORBA technology.

Michael Kircher

Washington University in Saint Louis
1998

References

[1] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *Antipatterns - Refactoring Software, Architecture and Projects in Crisis.* Jon Wiley and Sons, Inc., 1998.

[2] David Flanagan. *JAVA in a Nutshell 2nd ed.* O'Reilly, 1997.

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

[4] Object Management Group. *The Common Object Request Broker: Architecture and Specification, 2.0 ed.* OMG, 1995.

[5] Object Management Group. *Specification of the Portable Object Adapter.* OMG, 1997.

[6] Timothy H. Harrison, David Levine, and Douglas C. Schmidt. *The Design and Performance of a Real-Time CORBA Event Service.* DOC group, Washington University, 1997.

[7] IONA. *Orbix 2 Programming Guide.* IONA, 1997.

[8] OMG. *CORBA services: Common Object Services Specification.* OMG, 1997.

[9] Douglas C. Schmidt. *The ADAPTIVE Communicaton Environment.* DOC group, Washington University, 1994.

[10] Douglas C. Schmidt. *The Distributed Object Visualization Environment.* DOC group, Washington University, 1997.

[11] Douglas C. Schmidt. *Experiences Converting a C++ Communication Software Framework to Java.* DOC group, Washington University, 1997.

[12] Douglas C. Schmidt. *ACE Software Development Guidelines.* DOC group, Washington University, 1998.

[13] Douglas C. Schmidt, David Levine, and Sumedh Mungee. *The Design of the TAO Real-Time Object Request Broker.* DOC group, Washington University, 1997.

[14] Douglas C. Schmidt, Sumedh Mungee, and Aniruddha Gokhale. *Alleviating Priority Inversion and Non-determinism in Real-Time CORBA Core Architectures.* DOC group, Washington University, 1997.

[15] Bjarne Stroustrup. *The C++ Programming Language 2nd ed.* Addison-Wesley, New Jersey, 1995.

[16] Sun. *http://java.sun.com/beans/spec.html.* Sun Microsystems, Inc, 1997.

[17] Visigenic. *VisiBroker for JAVA 3.1 documentation.* Visigenic, 1998.

[18] Andreas Vogel and Keith Duddy. *Java Programming with CORBA*. Wiley & Sons, 1997.