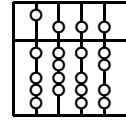


Technische Universität München
Fakultät für Informatik

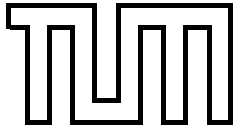


Studienarbeit

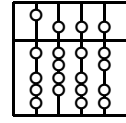
Design and Practical Guideline to a Corba-based Communication Framework

DWARF: Distributed Wearable Augmented Reality Framework

Lothar Richter



Technische Universität München
Fakultät für Informatik



Studienarbeit

Design and Practical Guideline to a Corba-based Communication Framework

DWARF: Distributed Wearable Augmented Reality Framework

Lothar Richter

Aufgabensteller: Prof. Bernd Brügge, Ph.D.

Betreuer: Dipl.-Inf. Asa MacWilliams

Abgabedatum: 16. August 2002

Ich versichere, daß ich diese Studienarbeit selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 16. August 2002

Lothar Richter

Zusammenfassung

Die vorliegende Studienarbeit stellt eine praktische Einführung in den Umgang mit CORBA und DWARF in zwei Abschnitten für Benutzer ohne einschlägige Vorkenntnisse dar. CORBA spezifiziert eine plattformübergreifende Schnittstelle, die eine transparente Kommunikation in verteilten Systemen ermöglicht. DWARF baut auf dieser Technologie auf und stellt ein Softwaregerüst zur Verschaltung verschiedener Komponenten zur Laufzeit zur Verfügung. Diese Eigenschaft prädestiniert DWARF besonders zum Einsatz als Basis von Systemen, die sich mit erweiterter Realität befassen.

Im ersten Teil werden in aufeinander aufbauenden Schritten die wichtigsten theoretischen Grundlagen von CORBA vermittelt. Im Anschluß an diese Ausführungen werden die vorgestellten Konzepte anhand von konkreten Programmierbeispielen ausgeführt. Der Code dieser Beispiele ist bewußt einfach gehalten, um es dem Leser zu erleichtern, die Umsetzung zwischen Konzept und Implementierung zu verstehen, und um den Lernerfolg zu steigern. Anregungen, selbst mit dem Beispielcode zu experimentieren wurden eingesetzt, um bei wichtigen Konzepten ein vertieftes Verständnis durch praktische Erprobung zu vermitteln.

Wurde im ersten Teil mit der Vorstellung von CORBA noch eine Technologie besprochen, die bereits länger bekannt ist, behandelt die Einführung in DWARF – aufbauend auf den im ersten Teil vermittelten Kenntnissen – im zweiten Teil eine Technologie, die es in dieser Form erst seit kurzem gibt. Die in dieser Arbeit beschriebenen Anwenderschnittstellen wurden erst vor kurzem spezifiziert und in der Anwendung bisher nicht umfassend dokumentiert. Somit leistet diese Arbeit einen wesentlichen Teil zur technischen Dokumentation des DWARF-Frameworks und erleichtert damit zukünftigen Anwendern eine Einarbeitung und erfolgreiche Anwendung von DWARF.

Abstract

This work is a practical introduction to the use of CORBA and DWARF in two sections for users without any relevant knowledge required. CORBA specifies a platform independent interface which facilitates transparent communication in distributed systems. DWARF is based upon this technology and provides a software framework for the connection of different components at run-time. These capacity highly recommends DWARF for the use as base for systems dealing with augmented reality.

The first section imparts in consecutive steps the most important theoretical foundations of CORBA. Afterwards of these explanations the presented concepts are demonstrated in implementation examples. The code of these examples was intentionally kept simple to make it easier for the reader to understand the transformation of an idea into code and thereby increase learning success. Playing around with the sample code was suggested to come to a deeper understanding of important ideas through practical experience.

Whereas the discussion of CORBA in the first section dealt with a technology which is already known for some time the introduction to DWARF in section two treats – based on the knowledge gathered in section one – a technology which in this form emerged not long ago. The interfaces described in the work were specified recently and are not comprehensive documented yet. From this point of view this work contributes a substantial part of technical documentation of the DWARF framework and facilitates the training and deployment of DWARF for future users.

Preface

About this work This work was done as Studienarbeit for the Aufbaustudium Informatik at the Technische Universität München under the supervision of Asa MacWilliams and Prof. Bernd Brügge at the Chair for Applied Software Engineering at the Technische Universität München.

Motivation The idea for this work has originated from the TRAMP project conducted in winter 2001/2002 at the Chair for Applied Software Engineering as a course in practical software engineering. At this time, I was a team member of the architecture and the network/middleware team. In this project about fifty students had to develop a “Traveling Repair And Maintenance Platform” – a distributed system which was reconfigurable at runtime. As the communication layer (middleware) between the different components, we employed the DWARF framework which was recently developed at this chair. One of the most timeconsuming steps for the realization of TRAMP was sorting out the few relevant facts from the vast amount of details of CORBA and DWARF one faced when the transition from the UML model to the source code took place. This work now should help future project participants to get a first impression of CORBA and the thereon based DWARF framework.

Acknowledgements I am indebted to Prof. Bernd Brügge and Asa MacWilliams for making this thesis possible and giving helpful advice, guidance and support during the last months.

To the commemoration of the grief and plight of my fellows in the TRAMP course which gave the initial idea to this work. Thanks for your patience.

I thank my family for encouragement, patience and support.

Contents

1	Introduction	1
1.1	Scope and Goals of this Tutorial	1
1.2	Structure of this Work	2
1.3	How to Use this Tutorial	2
2	Some CORBA Essentials	5
2.1	Important CORBA Buzzwords	5
2.1.1	What is a CORBA object at all ?	5
2.1.2	What is a CORBA interface ?	5
2.1.3	Implementation of interfaces	6
2.1.4	Some technical terms	6
2.1.5	CORBA at Work	7
2.1.6	'Magic References'	7
2.1.7	Some words about parameters	7
2.1.8	Synchronous communication by method invocation	9
2.1.9	Asynchronous communication using events	9
2.2	Fundamental CORBA examples	11
2.2.1	Get the Time	12
2.2.2	Parameter Passing	16
2.2.3	Interface Inheritance	17
2.2.4	Using a Callback Interface	17
2.2.5	Using the Notification Service	18
3	Essentials of DWARF	21
3.1	Ideas behind DWARF	21
3.1.1	About DWARF	21
3.1.2	General Overview	21
3.2	Components to which influence the behaviour of a DWARF service	21

Contents

3.2.1	Interfaces, their contribution to service's behaviour and implementation consequences	23
3.3	Dynamic Connection Development	25
3.4	Minimal DWARF service	26
A	Used Libraries	28
B	Installation of the Program	29
B.1	Installation of the Source Code	29
C	Source Code of the Examples	30
C.1	Get the Time	30
C.1.1	Interface Definition	30
C.1.2	Implementations	30
C.2	Parameter Passing	34
C.2.1	Interface Definition	34
C.2.2	Implementations	34
C.3	Interface Inheritance	40
C.3.1	Interface Definition	40
C.3.2	Implementations	40
C.4	Using a Callback Interface	48
C.4.1	Interface Definition	48
C.4.2	Implementations	48
C.5	Using the Notification Service	55
C.5.1	Initialization of the Event Channel	55
C.5.2	Implementation of Supplier and Consumer	57
C.5.3	Driver Code	60
D	DWARF Time Service	65
E	Helper Class Corbainit	74
F	Glossary	79
	Bibliography	81

List of Figures

1.1	Tutorial structure	3
2.1	Living Corba	8
2.2	Event Communication	10
2.3	Generation of Corba	13
2.4	Propagated Calls	15
3.1	ServiceDescription	22

1 Introduction

This introduction chapter contains some considerations about the demands a tutorial have to meet, the resulting internal structure and how to use it to draw benefit from this tutorial. There are only a few typographical convention: Buzz words which useful for the understanding and which are also often encountered in the literature are set in **boldface**. Identifiers of interfaces, typenames and functions are set in `truetype`.

1.1 Scope and Goals of this Tutorial

Scope: First of all I want to figure out what is to be covered in this tutorial and what is not covered. The scope of this tutorial are the presentation of important CORBA principles, mechanisms underlying the DWARF middleware and a basic command of both of them. This tutorial is a starting point into the world of CORBA and DWARF meant for beginners and not for skilled software engineers looking for into the depth coverage. Those people as well as advanced users looking for further information can be refered to some excellent text books [5, 3].

Explicitly not covered are the OMG's software architecture model and the suite of CORBA services. I will also not cover the theory of distributed systems in general or in special except what is inevitably needed. For this tutorial I assume also that the reader has a basic knowledge of object oriented technology and programming experience in C++.

Goals: Beside the design of a helper class to make it easier for AR developers to use the DWARF middleware and further shield them from dealing with nasty middleware details the other major goal is rather a process. The process of enabling users to understand and employ the middleware up from a beginners level what should be done by this tutorial. Writing a tutorial requires an author to have several things in mind. The intended readership are students which are new to CORBA or at least new to DWARF. Because the DWARF middleware is a high complex matter a beginner immediately facing it would be confused and overstrained so that they should be introduced to the theory and taught step by step in conjunction with a hands-on-practise to reach an active command of the subject. So on one hand I had to identify the important ideas and on the other it was necessary to find the sequence for an appropriate presentation. These goals are fulfilled by the beginning with the very basics of CORBA, then move along to the more complicated use of CORBA services and last introduce the DWARF middleware similar to a CORBA service together with a simplifying helper class. This should combine the settlement of already presented material with the aquisition of new material and ensure an optimal training success towards the command of DWARF.

1.2 Structure of this Work

Due to the introductory and explanatory purpose of the Studienarbeit the whole document will be structured into the introduction chapter and to further two main chapters each consisting of two parts – a part what of the theory is going to be covered, and a part how the things applied in code examples.

Because this document follows its very own structure compared to other documents produced during an object oriented software development path like presented in [1] it seemed necessary to me to show up the reasons that convinced me to leave the main stream. Software projects nowadays normally contain requirements analysis, followed by system design, object design, implementation and testing as major development activities. Each of them results in a written document displaying the results of the respective phase which is regarded as part of the deliverables. During development iteration through the different stages still take place, but is only poorly reflected in these documents which describes normally the most recent design. The documents are by-products, nevertheless they are necessary and constitute a major source of technical information. The main product of this process is a working computer system. Because the goal of this work is not only a running system consisting of the example code and the `CorbaInit` helper class but more to supply a user with a command of the DWARF framework, I have chosen this different style for presentation. The global structure of this tutorial is shown in 1.1.

1.3 How to Use this Tutorial

This paragraph deals with the practical application with this tutorial. The examples were developed and tested on a SuSE 7.3 installation and should run well on equivalent systems. The source reference for omniORB and omniNotify is found in appendix A.

- Hardware required: No special requirement, personal computer or equivalent.
- Software required:
 - GNU compiler and utility collections
 - omniORB and omniNotify
 - DWARF framework
- Steps to setup from scratch:
 1. retrieve source of omniORB and omniNotify
 2. compile and install first omniORB, then omniNotify – set the required environment variables
 3. retrieve the DWARF source
 4. run bootstrap and configure as it is described in the documentation
 5. compile and install the DWARF middleware
 6. change to the tutorial root directory

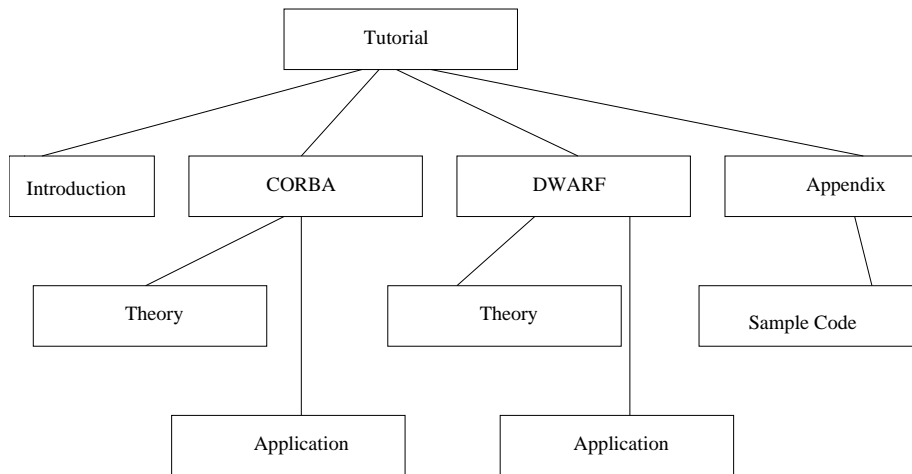


Figure 1.1: Global structure of this tutorial

7. compile the tutorial examples in the subdirectories

- In an installed DWARF environment:
 - change to the tutorial root directory
 - compile the tutorial examples in the subdirectories

If you are new to CORBA and DWARF start with the CORBA section. Read the theoretical section first, then skim through the example code explanation and compile the examples if you have not done this yet and run the examples. Go back now to the example explanation and walk through the explanation step by step with the example code lying in front of you. Try out the suggestions made to modify the code and think about the exercise questions.

If you are already familiar with CORBA and the use of CORBA services skip this section and go directly to the DWARF section. Use the CORBA section for look ups when necessary.

2 Some CORBA Essentials

CORBA was specified by the **Object Management Group (OMG)** to establish a platform independent communication layer for distributed applications. The CORBA middleware supplies a interface for applications to realize distributed applications while releasing the different applications from dealing with platform specific details. The whole system is based on an object oriented architecture model where client objects request services from server objects. The core component of each CORBA run-time implementation is an **ORB**, an object request broker. This is an implementation of the **broker** pattern [2] which does all the platform specific conversions and low-level communication task for application components.

2.1 Important CORBA Buzzwords

Under this topic I will try to give a brief overview about basic CORBA entities and typical mechanisms to handle CORBA objects. This should lead to a basic understanding of the nature and the behaviour of CORBA objects and typical manipulations performed upon them. Of course this cannot be a comprehensive and into the depth coverage of CORBA concepts but that was never intended at all. It is aimed to focus on some simple concepts to lay the ground for later understanding of the DWARF framework where one encounters both basic CORBA actions and overlaid DWARF concepts at the same time. Without basic understanding of CORBA it is hard to distinguish between concepts from these different architecture layers and make a comprehensive understanding difficult.

2.1.1 What is a CORBA object at all ?

A **CORBA object** can be regarded as a simple piece of code somewhere in the working memory of a computer. This group of bytes belongs logically to a function or contains data information which are made available to be called by other functions contained maybe in the same address space, i.e. the same process, or alternatively on the other side of the world. It implements functions specified in a **CORBA interface**. (The access is mediated via an **ORB** and specific connection code harboured in **stub** and **skeleton** files. see below) So a **CORBA object** can be seen as (real) subset of a normal object which implements just one or more **CORBA interfaces**.

2.1.2 What is a CORBA interface ?

and what is generated from an **IDL** interface specification ? The **CORBA interface** is implemented in **IDL** (interface definition language) defined by the **OMG** (Object Management

Group). Any interface has to be transformed in implementation language dependent code. This is done by the **IDL** compiler. The **IDL** compiler transforms the abstract function definitions from the interface to code acting as missing link between objects implementing the concrete actions constituting the function and the **ORB** which resolves the object references, receives the function calls and dispatches it to the implementing objects. The generated code contains all the functionality for necessary type conversions, marshalling of parameters and handling of references to the implementing objects. Depending from the specific **IDL** compiler different files are generated. Either code necessary for clients and code for servants is separated in **stub** and **skeleton** files or both is included in a common file (e.g omniidl produces `interface;SK.cc/h` files).

2.1.3 Implementation of interfaces

How are interfaces implemented and accessed ? As shown above a **IDL** interface results in **stub/skeleton** code. In the **skeleton** part all the methods of the interface can be found as abstract (virtual) methods. The only you have to do now is to crate a separate implementation class derived from a class call `POA_<interface>` and `PortableServer::RefCountServantBase`. In this implementation class you only have to create an implementation of the inherited interface methods. All the connection to the **ORB** and the marshalling stuff is already provided through the super classes. Because the interface methods are declared virtual the **ORB** can make an upcall through the object adapter and can call the implementation call at run-time.

2.1.4 Some technical terms

Now once the basic structures are described there are some technical terms which describe these entities.

Stub: A **stub** is that part of the code generated from an **IDL** interface description which is included by the **client**. It provides access to a **servant** object via a local proxy of this servant on the client side of the connection and contains the code responsible for information passing to the **ORB**.

Skeletons: A **skeleton** contains the base class of interface implementing classes and handover the upcalls coming from an **ORB** to the concrete functional implementation code.

POA: This is an abbreviation for "Portable Object Adapter". This is the connection (adapter) object via which the function calls received by the **ORB** are dispatched to the implementing objects. Every CORBA object upon its creation has to be registered with an **POA** in order to make itself known to the **ORB** and to receive incoming calls. Without connection to an **POA** an object cannot be contacted by an **ORB**.

Servant: A **servant** is a simple instance of CORBA object. It is the interface implementing instance which really carries out the action upon function calls.

Server: A **server** is the process in which address space the **servant** objects are created and stored. A **server** can harbour only a single **servant** as well as multiple **servant** instances of the same or various type and has full control over the servants life cycle.

Client: A **client** is the object or the process which calls a **servant's** method via an **IOR**.

2.1.5 CORBA at Work

On unix and unix like systems tasks are carried out in processes. Each process has its own flow of control and address space. Normally all of the process' objects are kept within this address space and references point to addresses within this range. Overlapping address spaces – even on the same system – require special operating system support and occur mostly as shared memory blocks or I/O buffers for communication with the operating system. Sharing information with processes located on other hosts mostly already needs an explicit communication establishment. CORBA overcome these obstacles by providing a transparent communication model for objects of all kind regardless whether they reside in the same address space or on another host. Communication is directed with the use of worldwide unequivocal object references/addresses (**IOR**, Interoperable Object Reference) and a platform for creating and managing these references, the **ORBs**. A very general sketch which focuses on the aspect of defined interfaces of objects is given in [2.1](#)

2.1.6 'Magic References'

Types of references in C++ and CORBA are completely different. In C++ references or pointers always refer to a specific memory address and can be created and manipulated directly via language means. This is completely different from CORBA references also called interoperable references (**IOR**). An **IOR** is much more than an memory address and rather network wide unambiguous reference to the respective implementing **servant**. The structure of an **IOR** itself consists of a part understandable by all **ORBs** and a more specific part only resolvable by the issuing orb, the orb where the servant was registered. Access to **servant** by a **client** is done by dereferencing the **IOR** through the **orb**.

2.1.7 Some words about parameters

IDL allows to declare different parameter directions. "in" parameters are sent to the client one-directional and the client has no way to control further fate of the transmitted data. This is used, e.g. when you only need the return value of the function and does not care about the parameter. "inout" parameters are often used if parameters are modified during function execution and if the caller want to yontinue which the altered values. It is also very useful when the function should return more than one result value. In such a situation you can use the return value of the function for status indication and inout parameters for data transfer. "out" parameter supply another means of transferring information back from the servant to the client. But, because until the return of the called function these values are often uninitialized unaware use can cause you trouble like any use of uninitialized variables.

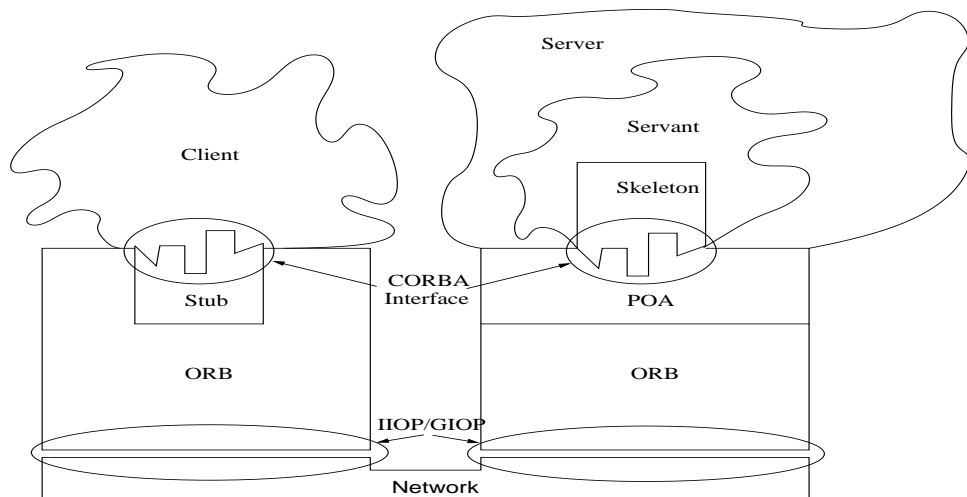


Figure 2.1: This illustration should display the connection over the well defined interface implemented in stubs and skeletons, whereas the cloud-shaped rest of the objects should stand for an undefined interface.

2.1.8 Synchronous communication by method invocation

This communication scenario is the most simple one. A **servant** object exists somewhere inside a **server** process. Once this **servant** publishes its **IOR client** processes can use this **IOR** to access the **servant's** functions, provided they know the **servant/interface** type, i.e. which functions are implemented by the **servant**. The **client** invokes a **servant** function, waits for the return and continues.

2.1.9 Asynchronous communication using events

CORBA services are collection of CORBA objects. Up to now basic features of CORBA technology have been discussed. Things like what is a CORBA object or how CORBA objects interact with each other. But during the design of distributed systems people has also noticed that there is a bunch of regularly repeating problems like demand for an asynchronous communication style, domain-limited unambiguous name resolution, control of object life-cycle or transparent relocation of objects. Some of these demands are specific to a certain application domain and do not affect others. To address these problems the OMG has specified some so-called CORBA services. A service is a collection of functions specified to address such a repeating problem. In other words it is bunch of pre-furnished CORBA objects providing a certain, widely employed functionality and therefore releasing the developer from reinventing the wheel. An example for such a service is the CORBA Notification service. This service provides means for asynchroneous communication between CORBA objects. In the following I will give a short description of how such a service can be used. The implementation used to construct the code example is `omniNotify1.1` from AT&T Research.

Components of Event Communication The core idea of the service is that of an **event channel**. The event channel acts as a central container for messages. Messages can be kept within this container for a certain time span, be can be send to the container and retrieved from the container. So this container acts as like a broker. [2] Messages in this context are now called **events**. These events have to follow some rules regarding their internal structure and can carry information load. Events can be created and funneled into the channel by objects called **suppliers** and can be delivered through the channel to objects called **consumers**. This shows already thatr a channel has two ends, one for suppliers and the other for consumers. A picture displaying the main component and their part in the event propagation is given in [2.1.9](#).

Communication models: There are two communication styles regarding the caller/callee sequence: The **push** and the **pull** model. With the push model the event is created by the supplier send to the channel and delivered to the consumer whereas with the pull model it goes the other way round. The consumer requests the event from the channel and the channel requests the event from the supplier. Which style of communication is used in an application depends from the type and amount of data transferred within an event an the expected frequency of communication. Besides the time decoupling of communication this technique has some additional advantages compared to the earlier method invocation. First supplier and consumer does not have to know each other and a one-to-many communication

is also enabled now since the event channel and the consumer (in the push model) has a publisher - subscriber relation which allows the same event to be sent to many consumers. As well a consumer can receive events from many suppliers at the same time with the same subscription to an event channel. Up to now this is a quite simple picture of the services and of course there are some additional configuration and filtering steps in between to avoid a babylonian event mess.

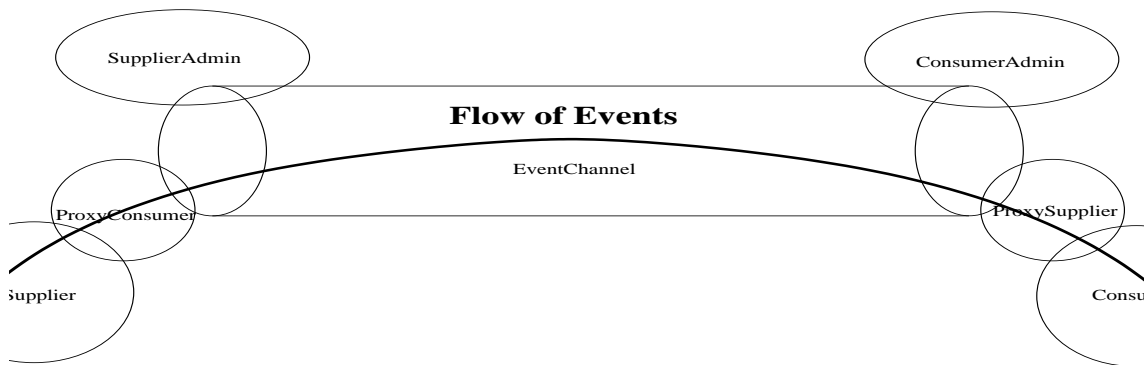


Figure 2.2: This diagram shows the components of an intact event communication line as well as the way of an event propagated after the push model.

Realization of Event Communication The notification run-time environment consists of a daemon process in first place – in the omniNotify implementation this daemon is called

notifd. This process holds all the objects associated with the establishment and administration of the event communication chain except the external objects of supplier and consumer. To establish such a communication line you need an initial object reference (IOR) to a CORBA object residing inside the daemon. Two stringified initial IOR's are supplied with this implementation of the Notification service. The place of these IOR's is stored together with other configuration setting in a special configuration file. This way of bootstrapping the Notification service may differ with other implementations. The two IOR's denote a event channel factory and an event channel created with defaulted settings.

If you don't want to use this prefurnished channel you have to retrieve your own one from the factory. Starting from this channel you can get now the objects needed to connect the original supplier and consumer. To connect the supplier one has to retrieve the object responsible for supplier administration from the channel first and use that to retrieve a proxy consumer object. This proxy consumer object is the real communication partner of the event supplier. Once the proxy consumer is registered with the supplier the supplier is connected to the event channel and ready for communication. On the consumer side the things are similar. First a consumer administration object has to be retrieved from the channel and this delivers a proxy supplier object. Our consumer has to register the proxy supplier and is ready for communication now. The use of specific administration objects of both side of the event channel and special proxy objects makes it possible to apply a sophisticated filtering on a general level (for all suppliers/consumer via the administration object) as well as on single connection level (for a specific supplier/consumer via the proxy objects) and conveys high flexibility to the system. Once you have this whole picture in your mind it should be easier to understand the code example where the specified interface names had to be use. These names are left out here on purpose to avoid that the reader becomes distracted or confused by the often long and maybe confusing interface names.

2.2 Fundamental CORBA examples

Within the next sections I am going to explain now the practical work with CORBA. Though CORBA is principally based on the client/server model, in real CORBA applications most objects act in the client role as well as in the server role depending on the context. This level of complexity is postponed to example four (Using a Callback Interface) with the introduction of a callback interface. The first three example rely on a more simple structure to focus on the respective topics.

The examples one through four are extended step by step to introduce some of the most important steps to develop CORBA applications. Every example is self-contained and complete independent from the others. There are some portions of redundant code in every following example but the code was not refactorized for the sake of completeness of every example and to keep a structure that was easier to understand. That has the nice effect is, that you can modify and compile each example independently from others and explore the effects of your modifications. People who are already familiar with this stuff and want to know how to generate complete refactorized, non redundant application code are referred to the examples which comes with the omniNotify service.

All relevant code is stored in the 'corba'-subdirectories of the source code. Since the stubs and skeletons are generated automatically and rather voluminous they were not included

into the appended sample source code. They can be created easily by compiling the respective example source tree as it is described in ???. Nevertheless code fragments from these files were shown wherever it seemed helpful.

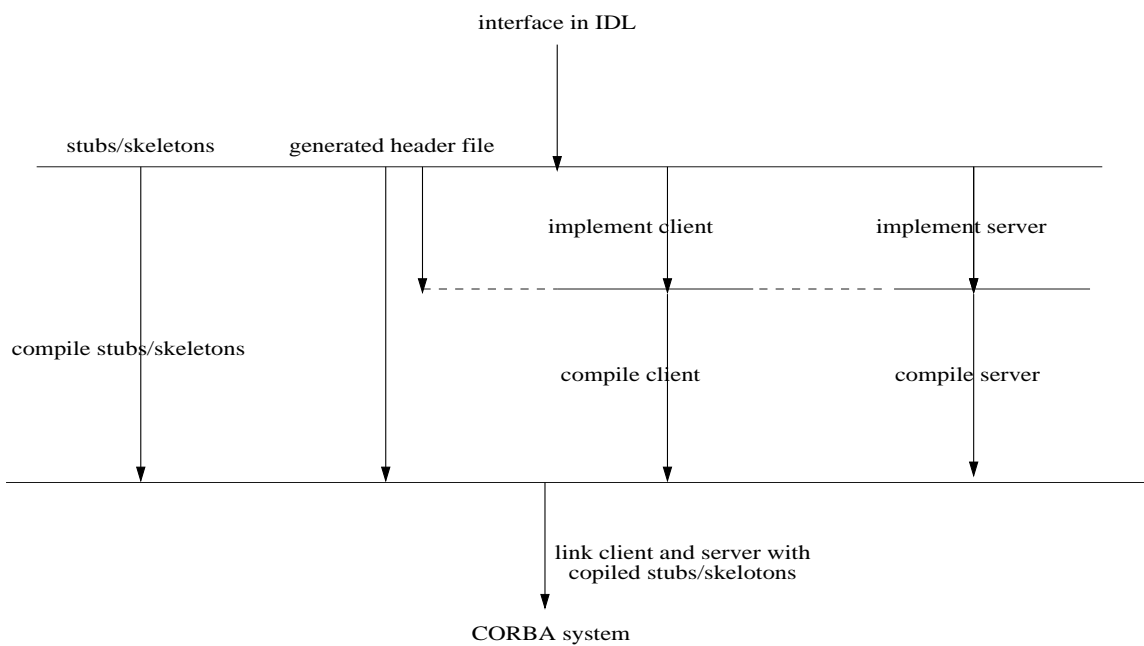
General remarks The first example deals with the basic mechanisms to generate client and server application starting from an IDL interface definition, generate stubs and skeletons and end up with a servant-class implementation.

2.2.1 Get the Time

This paragraph treats the most simple example for a CORBA application.

What is provided through IDL compilation: For this a very simple interface was declared in C.1.1 which described two methods. One method returns the Greenwich time and the other method accepts a string and writes it to standard output. This programming-language independent interface definition is processed by the IDL-compiler `omniidl` into two language specific files `<interface_file>SK.cc` and `<interface_file>.h`. These files contain the beside definitions of some helper classes the definition of stub classes named `class <interface>` and the skeleton classes called `class POA.<interface>`. The identifier `<interface_file>` should denote the file which contains the interface(s) definition(s) in this context. The concise distribution of the several declarations and implementations into two files is implementation specific and may vary with other IDL-compilers. These files are used via `#include` and linking of the compiled `<interface_file>SK.cc` file. A typical way of generating a CORBA application is depicted by the activity diagram shown in 2.3.

Implementing the Server: Because most of the communication structures are already realized in `<interface>SK.cc` files and the libraries representing the ORB there is only the application logic left to be implemented C.1.2. This is done for the server role by creating an implementation class which represents the later servant object by inheriting from `class POA.<interface>` and `PortableServer::RefCountServantBase`. To become a servant object in its own right it is necessary to implement all interface methods which are declared as purely virtual in `class POA.<interface>`. Now we have got the servant object, we need a susceptible server process. This server process, later on referred to as server, is coded in a separate file to demonstrate the difference between specific parts and common server code stuff. It is the server's task to do the initialization of the orb, calling the static method `CORBA::ORB_init(argc, argv)`. This call initializes the orb, and hands over given orb-determined arguments. Until now the ORB is the only given CORBA object reference, but now it is possible to retrieve a small number of additional important object references specified by the standard by invoking the `resolve_initial_references()` method with a specified name. Here it is used to retrieve a reference to `RootPOA`, the initial POA configured with default settings, and for our examples this configuration is far enough. So, in principle possible to run several POA's with different behaviour the selection of the right POA is managed via the `POAManager`. This object reference is retrieved directly from the `Root_POA`. Calling `activate()` on the `POAManager` activates the communication line



Generation of a CORBA System as UML Activity Diagram

Figure 2.3: This UML activity diagram describes the steps involved in the generation of a CORBA system

up to the the POA. But the final recipient of a call is still missing. To fill the gap, a servant object is created by instantiating the implementation class `<interface>_imp` (remember that the name of the concrete implementation class is not specified and free to choose). Now we have got a initialized communication infra structure with the orb and an existing servant object, but still knows about. Therefore the a IOR to the servant object is retrieved by calling `_this()`, stringified by the ORB-method `object_to_string()` and published by writing to standard out. This way of publishing and exchanging of IOR's by stringifying and de-stringifying them is rather laborious and in normal CORBA systems only used during the bootstrapping phase. Lateron, references are normally exchanged by the means of function calls.

Contacting the Server: Now that the server side is completed we can turn to the client's side of the application. The location of the servant object to contact is encoded in the string given as second parameter to `main`. Before the client is able to make use of the stringified IOR it has to establish its own connection to the ORB. This done by the same call to `CORBA::ORB_init(argc, argv)` as in the server process. After this the client can call the ORB's method `string_to_object()` to regenerate an IOR from the stringified parameter. But this conversion yields only the type the most general `CORBA_Object`. Performing the retransformation of a string into an IOR result in a creation of a stub object and return of a pointer to it by the ORB denoted by type `<interface_ptr>`. Using this pointer type would leave the client application with the whole responsibility for the memory management of the stub object. But beside the definition of the `<interface_ptr>` type for the pointer to the stub objects in the `<sample>.hh` files there are also wrapper classes `<interface_var>` defined which take over the burden of memory management and which can be used for the assignment of `<interface_ptr>` variables. This is similar to the use of smart pointers in C++ and hand over the complete control over the stub object to the `<interface_var>` variable. CORBA also defines a specific nil value as returned by `CORBA::_nil()` which is often the return value of a failed action. So the success of every IOR manipulation function can be tested by the call to `CORBA::is_nil()`. After we have got a valid reference to a `CORBA::Object` now we have to **narrow** its type before we can call any other interface specific methods. To do this we use a static method defined within the stub class `<interface>` and called `<interface>::_narrow()`. Successful call of this method gives a working IOR to an object implementing all methods specified in interface. This is checked again by use of `CORBA::is_nil()`. After we have obtained a valid IOR to a `<interface>` object we can invoke methods on this maybe remote object as simple as on local objects denoted by C++ pointers.

Accessing the Server: The call to `get_gmt()` returns a structure object containing the Greenwich which could be as easy accessed as a normal local structure type. The second call of a method on the remote object transfer a string argument to the the server object and demonstrate that a bidirectional information flow is possible. This will be discussed more deeply in the following example. Note that exception handling is rather sparse and that a real-world CORBA application would have to be able to deal with different kinds of `CORBA::Exceptions` rising from physical connection problems up to implementation errors in the server logic. An simplified model of the activation chain is shown in [2.4](#).

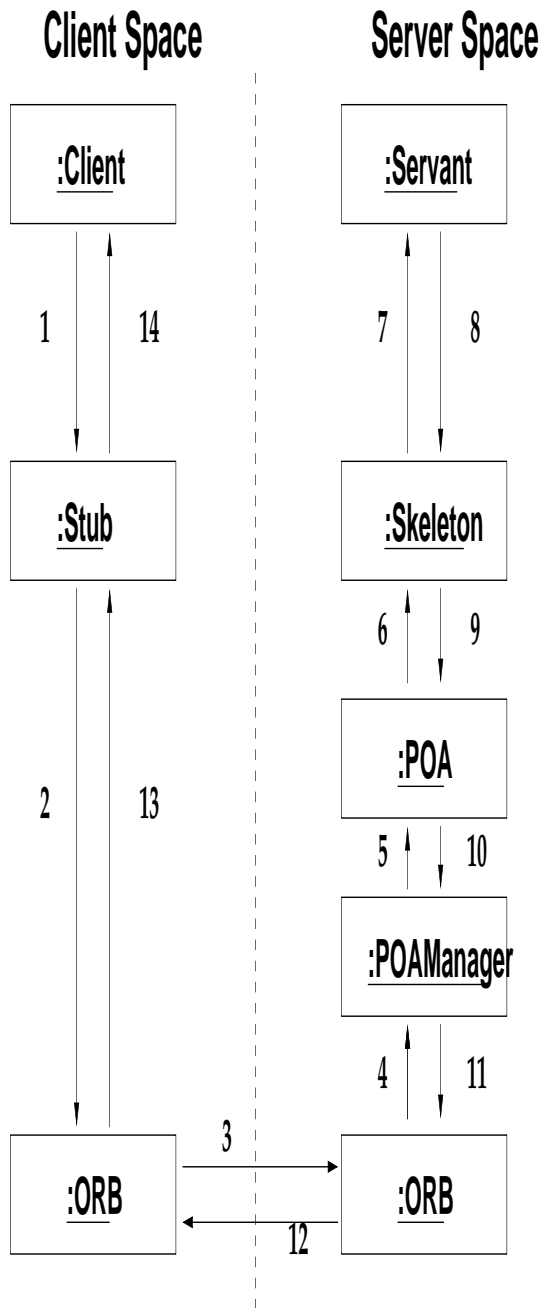


Figure 2.4: This diagram shows the sequence of function call from the requesting client to the servant object and back.

2.2.2 Parameter Passing

The example no. 2 C.2 now focuses more on the way method parameters are used. The IDL declaration of a method requires additional information except the method's name. These are the type of the return value and the types and direction of the parameters. Since the information stored in the parameters has to be transmitted to the servant IDL introduces direction specifier to minimize network traffic. There are three possible specifiers: **in**, **out** and **inout**. Declaring an **in** parameter tells the ORB that no modification of this value is expected respective allowed. In our example this mapped to simple call-by-value parameter. After receiving an **in** parameter and transmit it to the servant the ORB is no longer responsible for it and release all resources allocated for this parameter. The original value in the value stays untouched. The use of **inout** or **out** parameter direction means, that these variables are maybe modified by the servant call and may contain results. This is accomplished by mapping the parameter type to C++ references. With the help of these references it is now possible to modify the original variables in the clients address space upon return of the function call. This procedure requires the ORB to have resources allocated for the buffering of the results until all the respective parameters are passed back to the caller. The following code shows the respective line from the IDL declarations with the appropriate C++ code from the stubs respective skeletons:

- `void myecho(in string input);`
`void myecho(const char* input)`
- `long twofold(in long multiplicand);`
`CORBA::Long twofold(CORBA::Long multiplicand);`
- `void makeHalf(inout long theHalf);`
`void makeHalf(CORBA::Long& theHalf);`
- `void multiply(in long factor1, in long factor2, out long result);`
`void multiply(CORBA::Long factor1, CORBA::Long factor2, CORBA::Long& result);`

The mapping of these definitions shows some interesting points in addition to the above mentioned. First the simple IDL data types are mapped to respective C++ types prefixed with `CORBA::`, these are compatible with the build in C++ types. The IDL type string is mapped to `const char *` in this context where otherwise it is mapped to a simple `char*`. CORBA also provides a wrapper class called `CORBA::string_var` for the convenient string handling, which provides additional built-in memory management. But there are also some pitfalls related with the use of `CORBA::string_var` because of improper initialization. Upon creation from a `char*`, the `CORBA::string_var` object takes over the memory control and releases the memory when its scope is left. If the `CORBA::string_var` is initialized with string literal like "foo" it holds a pointer to the data segment of the program, which must not be modified by the code. This leads to a program failure. So if you want to make use of the advantages of the `CORBA::string_var` you always have to use the `CORBA::string_dup("foo")` call to create a `CORBA::string_var` from the string "foo". When you run the demo program you will also notice that for out parameters no implicit initialization takes place and therefore its value is undefined until return of the function.

2.2.3 Interface Inheritance

This example should demonstrate the use of derived, i.e. inherited interfaces. The IDL file contains a simple hierarchy of interfaces [C.3.1](#) each consists of only one method. Because IDL is a pure declarative language it can provide only interface inheritance. So all of the inherited interface methods are present in the skeletons as virtual functions and each of them has to be implemented in the derived servant class. A client which received an IOR to such an object implementing a complex interface can now invoke any method declared in the interface hierarchy on the remote object. The necessary prerequisite for this is just the conversion of the IOR to appropriate type regarding the interface hierarchy. This because C++ is string typed language and the compiler does not allow calls to methods which are not declared for a certain interface type even when the IOR and servant object would support these methods. Run the example and see the different behaviours of the three servant objects, each representing another level of the interface hierarchy. Modify the code and try to call the 'getSon()' method on the son-objects IOR without narrowing to type Son and see how the compiler reacts.

2.2.4 Using a Callback Interface

Until now we have covered the most basic ideas you are confronted with using CORBA. In this example now we will go one step closer to real CORBA application. In such an application there are a lot of interactions between objects making each object behave sometimes as client and sometimes as server. The establishment of the connections between these objects is normally done by exchanging IOR and not like in the first examples shown via the exchange of stringified IOR. This suitable for getting a few initial reference during bootstrapping of the system but it is not sufficient for a larger number of CORBA objects because it is too laborious as well as time and resource wasting with all its type conversions. The example defines two simple interfaces to demonstrate this idea [C.4.1](#). One is the interface 'Reminder' which is implemented by initial servant object. This interface offers three methods:

```
interface Reminder {
    boolean registerCallbackInterface(in Subscription subscriber);
    string getServantName();
    boolean remindMe(in long seconds);
};
```

Begin as usual: The methods 'getServantName()' and 'remindMe()' are rather administration or working methods – 'getServantName()' returns a string showing the identity of referenced object and 'remindMe()' reminding somebody or something after a certain number of seconds which you can specify. The third method 'registerCallbackInterface()' is the most interesting method for us now. It takes a Subscription-type argument which contains besides a name string the IOR of a 'CallbackInterface' object, a CORBA object which implements the 'CallbackInterface' interface. Because the servant object stores the transmitted IOR it knows now about another CORBA object. To make use of this a client has to implement the 'CallbackInterface' interface itself, instantiate and retrieve an IOR to its own 'CallbackInterface'

object, because IOR's can only be retrieved from CORBA objects. This IOR can now be encapsulated into a Subscription structure and passed to the 'Reminder' object, the servant up to now.

Change the roles: Once the client has registered with the servant it can call the servant's 'remindMe()' method together with specified delay for the callback. An important change occurs: Client and server change their roles, the 'Reminder' object – the former server – becomes a client and calls a method of the 'CallbackInterface' object – the former client – which now becomes the server. The sample code reflects this in several points. First we have to implement the separate class 'CallbackInterface.impl' and instantiate it in the client process which is realized by method main in sample_client.cc. This is because only CORBA objects can be accessed by the ORB – note that in the other direction CORBA objects and the ORB can be accessed by non-CORBA objects very well. Until the registration with the 'Reminder' object and invoking the 'remindMe()' method the client process behaved purely like a client. Now the client has to turn into a server to activate its 'CallbackInterface' object and become susceptible for incoming calls. This is done by invoking the 'run()' method telling the ORB to listen for requests for the implemented interface.

Take Care ! This is a point where things start to become complicated. Because control is now passed over to the servant object waiting for incoming requests, the process cannot act any longer as a client and do anything else as waiting for requests in this thread. You can try this out rather easily by exchanging the respective code line in sample_client.cc and observe the blocking of the client. You can solve this problem by splitting the control into separate threads and thus allowing the servant to listen for requests as well as continue to do some other work in another thread. This solution is also used by the DWARF framework and is demonstrated in a simple way in the DWARF TimeService.

2.2.5 Using the Notification Service

After the introduction of basic techniques dealing with CORBA objects I want to cover the application of the CORBA Notification Service as a means to achieve asynchronous communication. While synchronous communication can simply be realized by invoking methods on other CORBA objects asynchronous communication is more difficult.

The use of functions provided by CORBA services is rather formalized because of the high degree of specification of these services and maybe seen to be laborious. But on the other hand this also has some advantages. Once you have set up the basic structures reuse of these components is rather easy and can save further development time. This idea led to the splitting of the code of the example which shows the use of the CORBA Notification service. The complete code is distributed in different portions each include a certain part of the whole functionality.

A working event communication line consists of at least five CORBA objects: An event supplier, an event consumer, an event channel and a ProxyConsumer and a ProxySupplier object. These objects are generated during a cascade of several function calls. Because all of these objects are CORBA objects the communication endpoints, the event supplier and the

consumer have to be implemented by the application developer by inherit from respective skeleton classes whereas the other components are provided by the implementation of the Notification Service.

Implementing the Supplier In our example we are going to use structured events and the push model for the communication. That means the supplier implementation has to inherit from `virtual POA_CosNotifyComm::StructuredPushSupplier` the skeleton class generated from the interface `StructuredPushSupplier` within the modul `CosNotifyComm`. Like all other types of event supplier this interface is derived from interface `NotifyPublish` in the same module and so we end up with two methods we have two implement in our supplier class:

- `void disconnect_structured_push_supplier()`
- `void subscription_change(const CosNotification::EventTypeSeq& added, const CosNotification::EventTypeSeq& removed)`

Implementing the Consumer The respective implementation of the consumer object has then to inherit from `virtual POA_CosNotifyComm::StructuredPushConsumer` and has to implement the interfaces `StructuredPushConsumer` and `NotifyPublish` which are also declared in modul `CosNotifyComm`. Thus our supplier class has to implement the methods:

- `void disconnect_structured_push_consumer()`
- `void push_structured_event(const CosNotification::StructuredEvent& se)`
- `void offer_change(const CosNotification::EventTypeSeq& added, const CosNotification::EventTypeSeq& removed)`

Implementing the EventChannel Now we have defined our communication endpoint with supplier and consumer implementations but we still have no defined access to the communication infrastructure that means an event channel. Because this is rather general task the code is summarized in a separate class `EventChannel`. This class provides the connection of our application to (performs the bootstrapping of) the Notification Service and can retrieve an initial object reference to an event channel factory from various sources. We use the stringified factory reference stored by `notifd`, the Notification Service daemon, in a file given by the daemon's configuration. Upon retrieval of the event factory reference an event channel is generated by the call to `myChannelFactory->create_channel(initial_qos, initial_admin, id)` using only default settings and returned upon the call to `EventChannel::getEventChannel()`.

Glueing Things together – the Proxy Objects We have defined now the both communication endpoints and the event channel in the center of communication chain. In between these parts some components are still missing, even when they are all supplied with the service implementation: the `StructuredProxyPushConsumer` object and the `StructuredProxyPushSupplier` object. To obtain these objects we have to get special administration objects from the event channel first. A `SupplierAdmin` object is returned by `new_for_suppliers()` which in turn creates the `StructuredProxyPushConsumer` object upon call to `obtain_notification_push_consumer()` on the event supplier side of the communication line. On the event consumer side the things are very similar. You have to create a `ConsumerAdmin` object by invoking `new_for_consumers()` on the event channel and this object delivers the desired `StructuredProxyPushSupplier` object as result of call to `obtain_notification_push_supplier()`. On each of these steps you have to provide some mandatory parameters to configure administration, quality of service and filtering settings. I haven't mentioned them here in the discussion of the several steps of assembling the communication line and for the beginning use of default values like is the example is far enough. The tuning of these parameters is presumable one the most demanding topics when one is using the CORBA notification service and should not tried unless you are familiar with the basics. The different interfaces mentioned here in the last paragraph are all defined within module `CosNotifyChannelAdmin` whereas the type information for quality of service and administrative objects can mostly be found in modules `CosNotification` and `CosNotifyFilter`. A complete description can be found in the OMG's specification document.

Final Execution – the Drivers All these necessary steps are performed by the two driver programs. The code in `supplierDriver.cc` initializes the event channel and creates the supplier's side of the application. It generates as well the structured event which are sent out via the event channel. As the name says, a structured event has a fixed internal structure. The concise structure is described extensively within module `CosNotification`. The information in the sections header and filterable data is meant to be used together with the different administration and filter policies to ensure effective distribution of the events. Data in which only the final receiver is interested are normally stored in the section `remanider_of_body`. The code in `consumerDriver.cc` retrieves the load information from the received events and completes the application on the consumer's side. The drivers for the first time in the code examples make also use of the new developed class `CorbaInit`, which carries out the initialization of the ORB and provides some important object references in addition.

3 Essentials of DWARF

3.1 Ideas behind DWARF

3.1.1 About DWARF

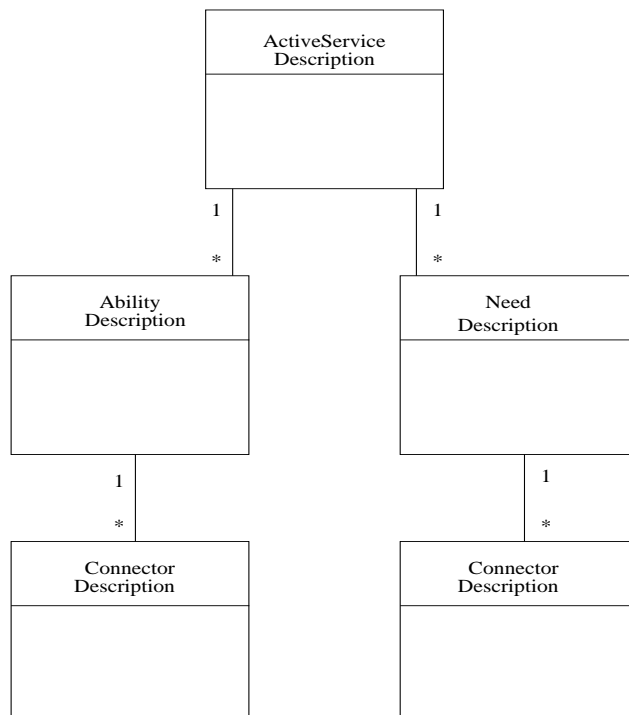
After becoming familiar with CORBA basics and getting a first idea of CORBA services it is time to introduce the DWARF framework. DWARF is a framework to support the development of augmented reality systems. Since augmentation of reality poses some special problems on software systems like handling multimodal input, spatial distribution and a high degree of flexibility in respect to reconfiguration at run-time a dedicated middleware layer is needed. This middleware layer is realized by the DWARF framework. DWARF was developed at the Chair of Applied Software Engineering as a coordinated effort.

3.1.2 General Overview

In the former chapter CORBA services have been introduced and now I am going to introduce DWARF in similar terms. From this point of DWARF can be regarded as CORBA service also. This service is specialized on run-time reconfiguration of distributed systems consisting of corba CORBA objects. Though an active system consists of objects belonging to the application part – the DWARF services – and objects belonging to the middleware, both parts realized by CORBA objects. The DWARFservices contain all the application and can be realized as separate processes distributed over different address spaces or hardware platforms and written in different programming languages. These services implement certain interfaces which allow them to describe themselves in terms of **needs** and **abilities** with their favorite style of communication and the type or format of data they use. The connection itself is established by the middleware represented through the service manager, the central process where all the services register with. Using the descriptions provided by the services the service manager find matching pairs and exchange the respective object references between the communication partners. A class diagram showing the structure of an `ActiveServiceDescription` is shown in [3.1](#).

3.2 Components to which influence the behaviour of a DWARF service

Most of the above given information is still very general and only suitable to get a big overview. Now I want to go into some details by introducing some of the more important



UML Class Diagram for a Completed ActiveServiceDescription

Figure 3.1: This illustration display the components of a completed ActiveServiceDescription.

DWARF interfaces and implementation consequences of use of certain interfaces. The interfaces mentioned below are always declared within module `DWARF` if not stated otherwise and can be found in the `Services.idl` file of the distribution.

3.2.1 Interfaces, their contribution to service's behaviour and implementation consequences

The interfaces involved in the connection establishment can be grouped in regard to two perspectives: To which phase of connection establishment belongs the interface and who implements this interface – service or service manager.

Communication Protocols: A thing that is common for all services is that a service inherently has to provide an implementation of a communication interface. If the communication is done directly with the corresponding service via method call, the services has to implement the interfaces `SvcProtObjrefImporter` for the information sink respective `SvcProtObjrefExporter` for the information source – the later interface is an empty one and realized by returning `_this`. Service which want an event based communication have to implement the `SvcProtPushConsumer` for the information sink respective `SvcProtPushSupplier` for the information source. These separation of one communication style into two interfaces comes from the still underlying client/server model which is now reflected as a pair of need and ability. Another mandatory requirement is the implementation of the `Service` interface with its only method `getStatus()`.

After discussing the question how the communication style itself comes to the service I am now going to describe the several interfaces as a service is introduced and connected to a DWARF system. A new feature which is common now for all interfaces is the inheritance from interface `Attributes`. These attributes supports the use of key/value string pairs to supply interfaces with further information for fine tuning demands.

Service Initialization: Before a service can be integrated it must initialize its contact to the middleware and it must introduce itself to the service manager. The initialization is done by the retrieval of a `CorbaInit` object which also holds an initial service manager reference. The introduction is done by configuring a `ServiceDescription` object which is issued by the service manager upon call to `newServiceDescription()`. This method needs an unique (is this right ?) parameter string which then becomes a identifier of that `ServiceDescription` object. This method is defined in the `SMgrDescribeServices` interface which offers beside some additional administrative methods the alternative description method `newServiceDescriptionFromXML()` which uses a persistent service description stored in XML format and thereby releases the service from this task. Just put a XML file in the right directory. When the description object is configured is has to be activated by calling `activateServiceDescription()` on the service manager.

What needs to be done to configure a `ServiceDescription` object ? Because the description of a service is crucial for the later behaviour I will explain these steps in more detail. A service description can be roughly separated into two sections. There is a section

which defines mainly the start and stop behaviour and a section describing all the different qualities of support a service needs or a service can provide.

StartOnDemand: At run-time a service runs at least in two different threads. One thread – the server thread – listens to the ORB for incoming requests for implemented functions. The other thread – the worker thread – comprises the rest of the work. This can be a dialog with a user, background work like image analysis or the query of another service, too. Because of the needed resources sometimes execution of the worker thread is not appreciated without any concrete reason. It is now possible to hand over the responsibility for start of the worker thread to the service manager.

This feature is controlled with `setStartOnDemand()` and the respective boolean parameter. If start on demand is desired the service has also to implement the `SvcStartup` interface. The service manager call then the `startService` method to start the worker thread.

Need/AbilityDescriptions: All the configuration methods I am relying on in the next lines are declared with the interfaces `NeedDescription` and `AbilityDescription`. The specific features in respect to information exchange are described in terms of **needs** and **abilities** as already mentioned above. In this context an **ability** stand for a certain type of information a service can offer to others whereas a **need** stand for a certain type of information a service depends on. A service must have at least one need or ability. A new `NeedDescription` respective `AbilityDescription` object is retrieved from a `ServiceDescription` object with the help of `newNeed()` respective `newAbility` and a name string as parameter. Besides an individual name `NeedDescription` and `AbilityDescription` objects have also a type. This type is implemented as a string and that type is what the service manager looks for when it is searching for matching Need/Ability pairs. `NeedDescription` as well as `AbilityDescription` objects can also hold a list with the various `ConnectorDescriptions` they want to use. For every `ConnectorDescription` mentioned here a service has to implement the respective method defined in the corresponding `SvcProt...` interface mentioned above and variables to store the references to the communication partners because correct connection handling is the in service's responsibility.

Simple Partner Selection: In addition to this a `NeedDescription` object has some extra fields which can be used to gain more influence on the partner selection. First of all there may be more than one suitable ability offered to fulfill a service's need. With the means of the methods `setMinInstances()` and `setMaxInstances()` a service itself can specify a suitable amount of communication partners. Again the correct handling of the set of partners has to be provided by the service. Second, there may be different matching abilities of the desired type, but the service has some special demands regarding e.g. quality of service or information content. The abilities can describe all these aspects of their information content by using functions from the inherited `Attributes` interface. The `NeedDescription` objects can now make use of these ability descriptions by setting a predicate using the `setPredicate()` method. This predicate is then used by the service manager to evaluate a suitable partner.

Advanced Partner Selection: Because the predicate has only a rather simple structure there is a further way to extend a services selection means by implementing the `SvcSelection` interface. After the evaluation of the predicates by the service manager all matching abilities are offered to the need using the `foundPartners()` method to the service. Now the need's service can do a much more comprehensive evaluation of the ability's attributes and can select a suitable partner calling `selectPartner()` on the service manager's `AsdSelection` interface.

How to use Sessions: Up to now I have described all the means how services can modify their start-up and connection behaviour. The last remaining important feature of services is the ability to support sessions. Sessions enable the service to keep track of their communication partners and allow the attachment of states to connections. If a service want make use of this feature it has to implement the `SvcSession` interface with its two methods `newSession()` and `endSession()`. It is also up to the service to implemented suited session objects. Because of the delegation of the actual connection handling to the session object this has now to implement the respective communication style depended method defined in the `SvcProt...` interfaces.

At run-time the service manager checks for implemented interfaces of its registered services and it recognizes a service's trait for session support – the implementation of the `SvcSession` interface. This makes the service manager to call `newSession()` on the service to retrieve the object on which is subsequently called the respective connection interface to set the connection partner. The special session object holds then all the accessory information that is regarded to worth to be kept with the connection. A demonstration implementation can be found in the `StatusReporter` service. This demonstration service can be found in the `src/services` subdirectory of the DWARF distribution.

At least, it has become clear that the complete behaviour of a service is defined by the set of the implemented interfaces. Due to the fact that CORBA restricts an objects to inherit only from one skeleton class (`POA_DWARF: :...` there is section three of the central definition file `services.idl`. In this section special aggregation interfaces are defined for every possible combination of DWARF interfaces. This leads then to the inclusion of all needed interfaces into one stub respective skeleton class suitable for further service implementation.

3.3 Dynamic Connetion Development

In the former section I describe the relation between a service's behaviour and implemented interfaces and methods. Now I want focus on the dynamic aspect. The first contact between a service and the service manager is when the service calls the service manager's `newServiceDescription()` method to describe itself (description phase). The following steps of self description a kept within the description object. With the call to `activateServiceDescription()` followed by `registerService()` the description become active and the service manage can act upon it (negotiatin phase). The service manager now looks for abilities which could satisfy this service's needs or needs of other services' which could be satisfied by this service's abilities. When all the needs of a service are satisfied (the predicates are already evaluated) the service manager proceeds with the next step

Phase	Implemented by the Service	Implemented by the Service Manager
Description Phase	ServiceDescription	SMgrDescribeServices
	AbilityDescription	
	NeedDescription	SMgrRegisterServices
Negotiation Phase	SvcSelection	AsdSelection
	SvcSession	
Connection Phase	SvcProt...	
	SvcProtPush...	
	SvcStartup	

Table 3.1: Interface of DWARF and their use.

in connecting two services. First a `SvcSelection` interface would be activated if available. Afterwards either a `SvcSession` interface is activated and a new session started or the services are immediately connected (connection phase). If the affected services were not started yet because of their `StartOnDemand` property they are started now and the communication begins.

This correlation is also depicted in table 3.1.

3.4 Minimal DWARF service

D This section discuss which implementation requirements a minimal featured service has to meet. The code for the here discussed example is part of the DWARF source code tree unlike the other example code. This means it is automatically compiled and installed during the DWARF installation.

The interface `TimeService` of this service is assembled from two parts. It inherits from the `Service` interface which is mandatory for every DWARF service and from the `Clock` interface which provides the application logic. This structure was chosen to demonstrate the separation of internal (DWARF) logic and application logic. The result in aggregation interface was then implemented by the `TimeService` service. To do this the `TimeService` has only to implement the method `getStatus()` from `Service` interface and the method `getDayTime` from the `Clock` interface. Because we have chosen communication via object references the corresponding interface for `TimeService` is empty and we need not to implement a further method. The `TimeClient` that should act a client to our `TimeService` has also to be a DWARF service, i.e. it has also to implement `getStatus()`. Beside this it need not to implement additional application logic method. But because of the communication style it has to implement `importObjref` inherited from `SvcProtObjrefImporter` and a variable to store the received IOR. To run this demonstration you have to start the service manager and `TimeService` and `TimeClient`. As soon as the connection is established the client retrieves the local time every three seconds and print it to the standard output. If you go through to code you can also see very clear, that the main thread is reserved for the server role to satisfy incoming requests whereas the application logic is run in a separate thread.

Appendix

A Used Libraries

For simplicity reasons we use the same platform for all the examples, CORBA-examples as well as DWARF-examples.

We used the following libraries:

omniORB. omniORB is a robust, high-performance ORB developed by AT&T Laboratories in Cambridge. It can easily be used with C++ applications and has been the base of all the CORBA communication described in this thesis. It is now available at <http://omniorb.sourceforge.net/>

omniNotify. omniNotify is a robust, high-performance Notification service developed by AT&T Laboratories in Cambridge. It can easily be used with C++ applications and has been the base of all the event based communication described in this work. It is still available at <http://www.research.att.com/~ready/omniNotify/>

B Installation of the Program

Text for installing the programs

B.1 Installation of the Source Code

Installing the CORBA examples

The source code of every tutorial example is grouped in its own source tree. Each tree is rooted with the tutorial number and two sub-trees:

- **bin:** The bin sub-tree contains the compiled executables of the demonstration examples.
- **corba:** The corba sub-tree contains all of the source files in three further sub-directories:
 1. **idl:** contains the interface definition written in idl and a makefile.
 2. **stubs:** contains the generated c++ stubs and a makefile.
 3. **cxx:** contains the implementations for server, client and driver components and the main makefile. Run 'make all' in this directory to compile the whole example.

1. Install all libraries mentioned in appendix [A](#) into the directory `D:\Developing`
2. Set the library and include paths according to the installation instructions.
3. Compile the various tutorial files by running running make in the respective root directory:
 - a) Get the Time
 - b) Parameter Passing
 - c) Interface Inheritance
 - d) Using a Callback Interface
 - e) Using the notification service
4. Enjoy the programs!

C Source Code of the Examples

C.1 Get the Time

C.1.1 Interface Definition

```
// sample.idl
// sample interface

struct TimeOfDay {
    short hour;
    short minute;
    short second;
};

interface Sample
{
    TimeOfDay get_gmt();
    void myecho(in string input);
};

// end sample.idl
```

C.1.2 Implementations

This directory contains the code for the client and for the server.

```
// sample_client.cc

#include <iostream>
#include <iomanip>
#include "sample.hh"

int main (int argc, char* argv[])
{
    try {
        // check arguments
        if (argc != 2) {
            cerr << "Usage: client IOR_string" << endl;
        }
    }
}
```

```

    throw 0;
}

// initialize and attach to orb
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// destringify argv[1]
//tell orb to restore IOR back from parameter string
CORBA::Object_var obj = orb->string_to_object(argv[1]);

if (CORBA::is_nil(obj)){ // check for valid IOR
    cerr << "Nil Time reference" << endl;
    throw 0;
}

// narrow
// cast IOR to the desired (right) interface type
Sample_var tm = Sample::_narrow(obj);

// check whether IOR was of the assumed type
if (CORBA::is_nil(tm)){
    cerr << "argument is not a Time reference" << endl;
    throw 0;
}
// start of the variable application code
// get time as response from the servant object
TimeOfDay tod = tm->get_gmt();
cout << "Time in Greenwich is "
<< setw(2) << setfill('0') << tod.hour << ":"
<< setw(2) << setfill('0') << tod.minute << ":"
<< setw(2) << setfill('0') << tod.second << endl;

// send information to the servant
// call myecho()
tm->myecho("Here comes CORBA !");
}

// simple standard error handling
catch (const CORBA::Exception &) {
    cerr << "Uncaught CORBA exception" << endl;
    return 1;
}
catch (...){
    return 1;
}

```



```

    return 0;
}

// end sample_client.cc

// sample_impl.hh
// contains the declaration parts of my derived class Sample_impl
#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh"

class Sample_impl : public POA_Sample,
                   public PortableServer::RefCountServantBase
{
public:
    inline Sample_impl(){}
    // virtual ~Sample_i(){}

    // routine: redeclare all interface methods from sampleSK.hh
    // use copy and paste to avoid typos
    virtual TimeOfDay get_gmt() throw (CORBA::SystemException);
    virtual void myecho(const char* input) throw (CORBA::SystemException);
};

// end sample_impl.hh

// sample_impl.cc
// contains the implementation parts of my derived class Sample_impl
// implementation of the above declared methods

#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh"
#include "sample_impl.hh"

TimeOfDay Sample_impl::get_gmt() throw (CORBA::SystemException)
{
    time_t time_now = time(0);
    struct tm* time_p = gmtime(&time_now);

    TimeOfDay tod;
    tod.hour = time_p -> tm_hour;
    tod.minute = time_p -> tm_min;
    tod.second = time_p -> tm_sec;
}

```

```
    return tod;
}

void Sample_impl::myecho(const char* input) throw (CORBA::SystemException)
{
    cout << input << endl;
}

// end sample_impl.cc
```

C.2 Parameter Passing

C.2.1 Interface Definition

```
// sample.idl

struct TimeOfDay {
    short hour;
    short minute;
    short second;
};

interface Sample {
    TimeOfDay get_gmt();
    void myecho(in string input);
    long twofold(in long multiplicand);
    void makeHalf(inout long theHalf);
    void multiply(in long factor1, in long factor2, out long result);
};

// end sample.idl
```

C.2.2 Implementations

This directory contains the code for the client and for the server.

```
// sample_client.cc

#include <iostream>
#include <iomanip>
#include "sample.hh"

int main (int argc, char* argv[])
{
    try {
        // check arguments
        if (argc != 2) {
            cerr << "Usage: client IOR_string" << endl;
            throw 0;
        }

        // initialize and attach to orb
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // destringify argv[1]
        //tell orb to restore IOR back from parameter string
```

```

CORBA::Object_var obj = orb->string_to_object(argv[1]);

if (CORBA::is_nil(obj)){ // check for valid IOR
    cerr << "Nil Sample reference" << endl;
    throw 0;
}

// narrow
// cast IOR to the desired (right) interface type
Sample_var tm = Sample::_narrow(obj);

// check whether IOR was of the assumed type
if (CORBA::is_nil(tm)){
    cerr << "argument is not a Time reference" << endl;
    throw 0;
}

// start of the variable application code
// get time as response from the servant object
TimeOfDay tod = tm->get_gmt();
cout << "Time in Greenwich is "
<< setw(2) << setfill('0') << tod.hour << ":"
<< setw(2) << setfill('0') << tod.minute << ":"
<< setw(2) << setfill('0') << tod.second << endl;

// send information to the servant
// call myecho()
tm->myecho("Here comes CORBA !");
cout << " Let's do some work:" << endl;
long aNumber = 7;
long theDouble = tm->twofold(aNumber);
cout << "Dublication of " << aNumber << " gives " << theDouble << endl;
cout << "value of theDouble before division by two is: " << theDouble << endl;
tm->makeHalf(theDouble);
cout << "theDouble has now the value: " << theDouble << endl;
cout << "Let's multiply" << endl;
long factor1 = 3;
long factor2 = 4;
long result; // only declared, not initialized yet !

cout << " factor1: " << factor1 << endl;
cout << " factor2: " << factor2 << endl;
cout << " result: " << result << "not yet initialized !" << endl;

tm->multiply(factor1, factor2, result);
cout << " factor1: " << factor1 << endl;
cout << " factor2: " << factor2 << endl;

```

```

    cout << " result: " << result << endl;
}

// simple standard error handling
catch (const CORBA::Exception &) {
    cerr << "Uncaught CORBA exception" << endl;
    return 1;
}
catch (...){
    return 1;
}
return 0;
}

// end sample_client.cc

// sample_impl.hh
// contains the declaration parts of my derived class Sample_impl
#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh"
#include <string>

#ifdef SAMPLE_IMPL
#define SAMPLE_IMPL
class Sample_impl : public POA_Sample,
    public PortableServer::RefCountServantBase
{
    // store the reference address of the communication partner
private:
    string myID;

public:
    inline Sample_impl(){myID = string("objectWithoutName");}
    virtual ~Sample_impl(){cout << myID << " says goodbye" << endl;}
    inline Sample_impl(string aName){ myID = aName;}

    // routine: redeclare all interface methods from sampleSK.hh
    // use copy and paste to avoid typos

    // first interface function
    virtual TimeOfDay get_gmt() throw (CORBA::SystemException);

    // second interface function
    virtual void myecho(const char* input) throw (CORBA::SystemException);

```

```
// third interface function
virtual long twofold(CORBA::Long multiplicand)
    throw (CORBA::SystemException);

// fourth interface function
virtual void makeHalf(CORBA::Long& theHalf)    throw (CORBA::SystemException);

// fifth interface function
virtual void multiply(CORBA::Long factor1, CORBA::Long factor2, CORBA::Long result)
};
#endif

// end sample_impl.hh

// sample_impl.cc
// contains the implementation parts of my derived class Sample_impl
// implementation of the above declared methods

#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh"
#include "sample_impl.hh"

// legacy
// first interface method
TimeOfDay Sample_impl::get_gmt() throw (CORBA::SystemException)
{
    time_t time_now = time(0);
    struct tm* time_p = gmtime(&time_now);

    TimeOfDay tod;
    tod.hour = time_p -> tm_hour;
    tod.minute = time_p -> tm_min;
    tod.second = time_p -> tm_sec;

    return tod;
}

// legacy
// second interface method
void Sample_impl::myecho(const char* input) throw (CORBA::SystemException)
{
    cout << myID << " has received the message: " << input << endl;
}

// demonstrate in parameter
```

```

// third interface funtion
long Sample_impl::twofold(CORBA::Long multiplicand)
    throw (CORBA::SystemException)
{
    return multiplicand * 2;
}

// demonstrate out parameter
// fourth interface function
void Sample_impl::makeHalf(CORBA::Long& theHalf) throw (CORBA::SystemException)
{
    theHalf /= 2;
}

// demonstrate out parameter
// fifth interface function
void Sample_impl::multiply(CORBA::Long factor1, CORBA::Long factor2, CORBA::Long& result)
    throw (CORBA::SystemException)
{
    result = factor1 * factor2;
}

// end sample_impl.cc

// server_driver.cc
// This file serves as driver for the sample objects
// It contain main() and conducts also the general ORB
// initialization, which is not object specific

#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh"           /* interface and helper declarartions*/
#include "sample_impl.hh"     /* my derived class' declarartions*/

// standard start-up code
// unmodified initialization code from Henning et al.
int main (int argc, char * argv[])
{
    try {
        // Initialize ORB
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // Get reference to root POA
        // necessary for server role
        CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
        PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
    }
}

```

```
// Activate POA manager
PortableServer::POAManager_var mgr = poa->the_POAManager();
mgr->activate();

// Create desired object
Sample_impl sample_servant_Andi("ANDI") ;
Sample_impl sample_servant_Bertha;

// Write its stringified reference to stdout
Sample_var sampleIOR_Andi = sample_servant_Andi._this(); // retrieve the se
CORBA::String_var str_Andi = orb->object_to_string(sampleIOR_Andi); // stri
cout << "Andi's address: " << endl << str_Andi << endl; // publish IOR

// Write its stringified reference to stdout
Sample_var sampleIOR_Bertha = sample_servant_Bertha._this(); // retrieve th
CORBA::String_var str_Bertha = orb->object_to_string(sampleIOR_Bertha); //
cout <<"Bertha's address:" << endl << str_Bertha << endl; // publish IOR

// Accept requests for servant functions
orb->run();
}
catch (const CORBA::Exception &) {
    cerr << "Uncaught CORBA exception" << endl;
    return 1;
}
return 0;
}

// end server_driver.cc
```


C.3 Interface Inheritance

C.3.1 Interface Definition

```
// sample.idl

// interface Grandfather
interface Grandfather
{
    string getGrandfather();
};

// interface Father
interface Father : Grandfather
{
    string getFather();
};

// interface Son
interface Son : Father
{
    string getSon();
};

// end sample.idl
```

C.3.2 Implementations

This directory contains the code for the client and for the server.

```
// sample_client.cc

#include <iostream>
#include <iomanip>
#include "sample.hh"

int main (int argc, char* argv[])
{
    try {
        // check arguments
        if (argc != 2) {
            cerr << "Usage: client IOR_string" << endl;
            throw 0;
        }

        // initialize and attach to orb
```

```
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// destringify argv[1]
//tell orb to restore IOR back from parameter string
CORBA::Object_var obj = orb->string_to_object(argv[1]);

if (CORBA::is_nil(obj)){ // check for valid IOR
    cerr << "Nil reference" << endl;
    throw 0;
}

// narrow to grandfather
// cast IOR to the desired (right) interface type
Grandfather_var gf = Grandfather::_narrow(obj);

// check whether IOR was of the assumed type
if (CORBA::is_nil(obj)){
    cerr << "argument is not a Grandfather reference" << endl;
    throw 0;
}else{
    cout << "object behaves now like a Grandfather" << endl;
}
cout << "Name of Grandfather: " <<gf->getGrandfather() << endl;

// narrow to Father
// cast IOR to the desired (right) interface type
Father_var fa = Father::_narrow(gf);

// check whether IOR was of the assumed type
if (CORBA::is_nil(fa)){
    cerr << "argument is not a Father reference" << endl;
    throw 0;
}else{
    cout << "object behaves now like a Father" << endl;
}
cout << "Name of Grandfather: " << fa->getGrandfather() << endl;
cout << "Name of Father: " << fa->getFather() << endl;

// narrow to Son
// cast IOR to the desired (right) interface type
Son_var so = Son::_narrow(fa);

// check whether IOR was of the assumed type
if (CORBA::is_nil(so)){
    cerr << "argument is not a Son reference" << endl;
    throw 0;
}else{
```

```
        cout << "object behaves now like a Son" << endl;
    }

    cout << "Name of Grandfather: " << so->getGrandfather() << endl;
    cout << "Name of Father: " << so->getFather() << endl;
    cout << "Name of Son: " << so->getSon() << endl;

}

// simple standard error handling
catch (const CORBA::Exception &) {
    cerr << "Uncaught CORBA exception" << endl;
    return 1;
}
catch (...){
    return 1;
}
return 0;
}

// end sample_client.cc

// grandfather_impl.hh
// contains the declaration parts of my derived class Sample_impl
#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh" // contains Grandfather, Father and Son definitions

class Grandfather_impl : public POA_Grandfather,
    public PortableServer::RefCountServantBase
{
private:
    const char* nameOfGrandfather;

public:
    inline Grandfather_impl(const char* firstName){nameOfGrandfather = firstName;
    // virtual ~Sample_i(){}

    // schema F: Deklarartion aller Interface Methoden, wie in ...SK.cc
    virtual char* getGrandfather();
};

// end grandfather_impl.hh

// grandfather_impl.cc
// contains the implementation parts of my derived class Sample_impl
```

```
// implementation of the above declared methods

#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh"
#include "grandfather_impl.hh"

char* Grandfather_impl::getGrandfather()
{
    return const_cast<char*>(nameOfGrandfather);
}

// end grandfather_impl.cc

// father_impl.hh
// contains the declaration parts of my derived class Sample_impl
#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh" // contains Grandfather, Father and Son definitions

class Father_impl : public POA_Father,
                  public PortableServer::RefCountServantBase
{
private:
    const char* nameOfGrandfather;
    const char* nameOfFather;

public:
    inline Father_impl(const char* firstName, const char* secondName)
    {
        nameOfGrandfather = firstName;
        nameOfFather = secondName;
    };
    // virtual ~Sample_i(){}

    // schema F: Deklaration aller Interface Methoden, wie in ...SK.cc
    virtual char* getGrandfather();
    virtual char* getFather();

};

// end father_impl.hh
```

```
// father_impl.cc
// contains the implementation parts of my derived class Sample_impl
// implementation of the above declared methods

#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh"
#include "father_impl.hh"

char* Father_impl::getGrandfather()
{
    return const_cast<char*>(nameOfGrandfather);
}

char* Father_impl::getFather()
{
    return const_cast<char*>(nameOfFather);
}

// end father_impl.cc

// contains the declaration parts of my derived class Sample_impl
#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh" // contains Grandfather, Father and Son definitions

class Son_impl : public POA_Son,
                public PortableServer::RefCountServantBase
{
private:
    const char* nameOfGrandfather;
    const char* nameOfFather;
    const char* nameOfSon;

public:
    inline Son_impl(const char* firstName, const char* secondName, const char* th
    {
        nameOfGrandfather = firstName;
        nameOfFather = secondName;
        nameOfSon = thirdName;
    };
    // virtual ~Sample_i(){}

    // schema F: Deklarartion aller Interface Methoden, wie in ...SK.cc
    // routine: declare methods mentioned in ...SK.cc
```

```

    virtual char* getGrandfather();
    virtual char* getFather();
    virtual char* getSon();

};

// contains the implementation parts of my derived class Sample_impl
// implementation of the above declared methods

#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh"
#include "son_impl.hh"

char* Son_impl::getGrandfather()
{
    return const_cast<char*>(nameOfGrandfather);
}

char* Son_impl::getFather()
{
    return const_cast<char*>(nameOfFather);
}

char* Son_impl::getSon()
{
    return const_cast<char*>(nameOfSon);
}

// server_driver.cc
// This file serves as driver for the sample objects
// It contain main() and conducts also the general ORB
// initializtion, which is not object specific

#include <stdio.h>
#include <string>
#include <iostream.h>
#include <cstdlib>
#include "sample.hh"           /* interface and helper declarartions*/
#include "grandfather_impl.hh" /* my derived class' declarartions*/
#include "father_impl.hh"     /* my derived class' declarartions*/
#include "son_impl.hh"

// standard start-up code
// unmodified initializatin code from Henning et al.
int main (int argc, char * argv[])

```

```
{
  try {
    // Initialize ORB
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    // construction of string as workaound to avoid const char * in servant obj
    const char* nameOfGrandfather = "Old John";
    const char* nameOfFather = "John";
    const char* nameOfSon = "Little John";

    // Get reference to root POA
    CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);

    // Activate POA manager
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Create desired objects
    Grandfather_impl myGran (nameOfGrandfather);
    Father_impl myDad(nameOfGrandfather, nameOfFather);
    Son_impl mySelf(nameOfGrandfather, nameOfFather, nameOfSon);

    // Write its stringified reference to stdout
    Grandfather_var gf = myGran._this(); // retrieve the servant's IOR
    CORBA::String_var strG = orb->object_to_string(gf); // stringify IOR
    cout << "IOR of Grandfather: " << endl;
    cout << strG << endl; // publish IOR

    Grandfather_var mf = myDad._this(); // retrieve the servant's IOR
    CORBA::String_var strF = orb->object_to_string(mf); // stringify IOR
    cout << "IOR of Father: " << endl;
    cout << strF << endl; // publish IOR

    Grandfather_var son = mySelf._this(); // retrieve the servant's IOR
    CORBA::String_var strI = orb->object_to_string(son); // stringify IOR
    cout << "IOR of Son: " << endl;
    cout << strI << endl; // publish IOR

    // Accept requestsfor servant functions
    orb->run(); // Servant aktivieren
  }
  catch (const CORBA::Exception &) {
    cerr << "Uncaught CORBA exception" << endl;
    return 1;
  }
  return 0;
}
```

```
// end server_driver.cc
```


C.4 Using a Callback Interface

C.4.1 Interface Definition

```
sample.idl

interface CallbackInterface;

struct Subscription {
    string subscriberName;
    CallbackInterface subscriberIOR;
};

interface Reminder {
    boolean registerCallbackInterface(in Subscription subscriber);
    string getServantName();
    boolean remindMe(in long seconds);
};

interface CallbackInterface {
    void remember( in string wakeUpMessage);
};

// end sample.idl
```

C.4.2 Implementations

This directory contains the code for the client and for the server.

```
// client_impl.hh
// contains the declaration parts of my derived class Sample_impl
#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh"
#include <string>

#ifndef CLIENT_IMPL
#define CLIENT_IMPL

class CallbackInterface_impl : public POA_CallbackInterface,
    public PortableServer::RefCountServantBase
{
    // store the reference address of the communication partner
private:
    string myID;
```

```
public:

    inline CallbackInterface_impl(){myID = string("objectWithoutName");}
    inline CallbackInterface_impl(string aName){ myID = aName;}

    // routine: redeclare all interface methods from sampleSK.hh
    // use copy and paste to avoid typos

    virtual void remember(const char* wakeUpMessage)
        throw (CORBA::SystemException);
    const char* getID();
};

#endif

// end client_impl.hh

// client_impl.cc
// contains the declaration parts of my derived class Sample_impl
#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh"
#include <string>
#include "client_impl.hh"

void CallbackInterface_impl::remember(const char* wakeUpMessage)
    throw (CORBA::SystemException)
{
    cout << wakeUpMessage << endl;
};

const char* CallbackInterface_impl::getID()
{
    return myID.c_str();
};

// end client_impl.cc

// sample_client.cc

#include <iostream>
#include <iomanip>
/* sample.hh contains the code base for interface reminder as well
 * as interface CallbackInterface*/
#include "sample.hh"
```

```
#include "client_impl.hh"
int main (int argc, char* argv[])
{
    try {
        // check arguments
        if (argc != 2) {
            cerr << "Usage: client IOR_string" << endl;
            throw 0;
        }

        // initialize and attach to orb
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // destringify argv[1]
        //tell orb to restore IOR back from parameter string
        CORBA::Object_var servObj = orb->string_to_object(argv[1]);

        if (CORBA::is_nil(servObj)){ // check for valid IOR
            cerr << "Nil Sample reference" << endl;
            throw 0;
        }

        // narrow, check for right type
        // cast IOR to the desired (right) interface type
        Reminder_var myReminder = Reminder::_narrow(servObj);

        // check whether IOR was of the assumed type
        if (CORBA::is_nil(myReminder)){
            cerr << "argument is not a Time reference" << endl;
            throw 0;
        }

        // prepare the client to act as callback recipient (server)
        // to receive calls from the orb you need a connection to a POA
        CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
        PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);

        // Activate POA manager
        PortableServer::POAManager_var mgr = poa->the_POAManager();
        mgr->activate();

        // we have to supply a Callback (interface) object
        // Create the servant object
        CallbackInterface_impl subscriber_Juli("JULI") ;
    }
}
```

```
    // make Juli susceptible for incoming calls
    /// orb->run(); causes block of further control flow;
cout << "control message" << endl;
    // retrieve IOR of Juli
    CallbackInterface_var clientIOR_Juli = subscriber_Juli._this();

    // assembly subscription
    Subscription subOfJuli;
    subOfJuli.subscriberName = subscriber_Juli.getID();
    subOfJuli.subscriberIOR = clientIOR_Juli;

    myReminder->registerCallbackInterface(subOfJuli);
    myReminder->remindMe(15);

    orb->run();

    // send information to the servant

}

// simple standard error handling
catch (const CORBA::Exception &) {
    cerr << "Uncaught CORBA exception" << endl;
    return 1;
}
catch (...){
    return 1;
}
return 0;
}

// end sample_client.cc

// sample_impl.hh

// contains the declaration parts of my derived class Sample_impl
#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh"
#include <string>
```

```

#ifndef REMINDER_IMPL
#define REMINDER_IMPL
class Reminder_impl : public POA_Reminder,
    public PortableServer::RefCountServantBase
{
    // store the reference address of the communication partner
private:
    Subscription mySubscriber;
    string myID;
    string mySubsName;
public:

    inline Reminder_impl(){myID = string("objectWithoutName");}

    // inline Sample_impl(string myName){myOwnName = myName;}
    virtual ~Reminder_impl(){cout << myID << " says goodbye" << endl;}
    inline Reminder_impl(string aName){ myID = aName;}
    // schema F: Deklarartion aller Interface Methoden, wie in ...SK.cc
    virtual bool registerCallbackInterface(const Subscription& subscriber)
        throw (CORBA::SystemException); // second interface function
    virtual bool remindMe(CORBA::Long seconds)
        throw (CORBA::SystemException);
    virtual char* getServantName()
        throw (CORBA::SystemException);
};
#endif

// end sample_impl.hh

// sample_impl.cc

// contains the implementation parts of my derived class Reminder_impl
// implementation of the above declared methods

#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh"
#include "sample_impl.hh"
#include <unistd.h>
// legacy
// first interface method

bool Reminder_impl::registerCallbackInterface(const Subscription& subscriber)

```

```

        throw (CORBA::SystemException)
    { mySubscriber = subscriber; mySubsName = mySubscriber.subscriberName; return true; }

bool Reminder_impl::remindMe(long seconds)
throw (CORBA::SystemException)
{
    sleep (seconds);
    string myMessage = string ("Hello ") + mySubsName + " it's time to wake up";
    mySubscriber.subscriberIOR->remember(myMessage.c_str()) ;
    return true;
}
char* Reminder_impl::getServantName()
    throw (CORBA::SystemException)
{
    return const_cast<char*> (myID.c_str());
}

// end sample_impl.cc

// server_driver.cc

// This file serves as driver for the sample objects
// It contain main() and conducts also the general ORB
// initializtion, which is not object specific

#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include "sample.hh"           /* interface and helper declarartions*/
#include "sample_impl.hh"     /* my derived class' declarartions*/
// standard start-up code
// unmodified initialization code from Henning et al.
int main (int argc, char * argv[])
{
    try {
        // Initialize ORB
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // Get reference to root POA
        // necessary for server role
        CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
        PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);

        // Activate POA manager
        PortableServer::POAManager_var mgr = poa->the_POAManager();
        mgr->activate();
    }
}

```

```
// gewünschtes Servant-Objekt erzeugen
// Create the servant objects
Reminder_impl sample_servant_Andi("Servant ANDI") ;

// Write its stringified reference to stdout
Reminder_var sampleIOR_Andi = sample_servant_Andi._this(); // IOR auf Servant
CORBA::String_var str_Andi = orb->object_to_string(sampleIOR_Andi); // IOR
cout << "Andi's address: " << endl << str_Andi << endl; // str. IOR bekannt m

// Write its stringified reference to stdout
//Sample_var sampleIOR_Bertha = sample_servant_Bertha._this(); // IOR auf S
//CORBA::String_var str_Bertha = orb->object_to_string(sampleIOR_Bertha); /
//cout <<"Bertha's address:" << endl << str_Bertha << endl; // str. IOR beka

// Accept requests

orb->run(); // Servant aktivieren

// worker code goes here

}
catch (const CORBA::Exception &) {
    cerr << "Uncaught CORBA exception" << endl;
    return 1;
}
return 0;
}

// end server_driver.cc
```

C.5 Using the Notification Service

This directory contains the code for the client and for the server.

C.5.1 Initialization of the Event Channel

```
// eventChannel.hh

#ifndef MY_EVENTCHANNEL
#define MY_EVENTCHANNEL

class EventChannel{
private:
    CorbaInit* myOrbConnection; // holds usefule object pointers
    CosNotifyChannelAdmin::EventChannelFactory_var myChannelFactory;
    CosNotifyChannelAdmin::EventChannel_var myEventChannel;
    void createMyChannel();
public:
    EventChannel(CorbaInit* intialRef);
    EventChannel(CorbaInit* intialRef, CosNotifyChannelAdmin::EventChannelFactory
    EventChannel(CorbaInit* intialRef, const char* channelFactRefFile);

    CosNotifyChannelAdmin::EventChannel_var getEventChannel();
};

#endif

// end eventChannel.hh

// eventChannel.cc

#include <string>
#include <iostream>
#include <stdio.h>
#include <omniORB3/CORBA.h>
#include <COS/CosNotifyChannelAdmin.hh>
#include "corbainit.hh"
#include "eventChannel.hh"

void EventChannel::createMyChannel()
{
    // create event channel
    // see also interface EventChannelFactory defined in CosNotifyChannelAdmin.idl
    CosNotification::QoSProperties initial_qos;
    CosNotification::AdminProperties initial_admin;
    typedef long ChannelID;
```



```

ChannelID id;
myEventChannel = myChannelFactory->create_channel(initial_qos, initial_admin,

};

// only default values
EventChannel::EventChannel(CorbaInit* intialRef)
{
    // initialize orb
    myOrbConnection = intialRef;
    // get event channel factory
    myChannelFactory = CosNotifyChannelAdmin::EventChannelFactory::_narrow(myOrbCo
    if (CORBA::is_nil(myChannelFactory)){
        cerr << "argument is not a ChannelFactory reference" << endl;
        throw 0;
    }

    createMyChannel();

};

// use customer's channel factory
EventChannel::EventChannel(CorbaInit* intialRef, CosNotifyChannelAdmin::EventCh
{
    myOrbConnection = intialRef;

    if (CORBA::is_nil(channelFactoryToUse)){
        cerr << "Your argument is not a valid ChannelFactory reference" << endl;
        throw 0;
    }
};

myChannelFactory = CosNotifyChannelAdmin::EventChannelFactory::_duplicate(char

    createMyChannel();

};

// use channel factory specified with IOR stored in file
EventChannel::EventChannel(CorbaInit* intialRef, const char* channelFactRefFile
{
    myOrbConnection = intialRef;

    myChannelFactory = CosNotifyChannelAdmin::EventChannelFactory::_narrow(myOrbCo
    if (CORBA::is_nil(myChannelFactory)){
        cerr << "File contains not a valid stringified ChannelFactory reference" <<
        throw 0;
    }
};

```

```

};

createMyChannel();
};

CosNotifyChannelAdmin::EventChannel_var EventChannel::getEventChannel()
{
    return myEventChannel;
};

// end eventChannel.cc

```

C.5.2 Implementation of Supplier and Consumer

```

// pushSupplier.hh

// this file contains the implementation of the StructuredPushSupplier-interface

// idl definition:
// interface StructuredPushSupplier : NotifySubscribe {
// void disconnect_structured_push_supplier();
// }; // StructuredPushSupplier

#ifndef MY_PUSHSUPPLIER
#define MY_PUSHSUPPLIER

class StructuredPushSupplier_impl : public virtual POA_CosNotifyComm::StructuredPushSupplier,
    public PortableServer::RefCountServantBase
{

private:
    bool connectionStatus;
    CosNotifyChannelAdmin::StructuredProxyPushConsumer_var prx_cons;
public:
    inline StructuredPushSupplier_impl() {connectionStatus = true;}
    // inherited from interface CosNotifyComm::StructuredPullSupplier : CosNotifyComm::NotifySubscribe
    void disconnect_structured_push_supplier();
    // inherited from interface CosNotifyComm::NotifySubscribe
    void subscription_change(const CosNotification::EventTypeSeq& added,
        const CosNotification::EventTypeSeq& removed);

    bool getConnectionStatus();
    void setConnectionStatus(bool newStatus);

```

```

    void setProxyConsumer(CosNotifyChannelAdmin::StructuredProxyPushConsumer_var
    void pushNow(CosNotification::StructuredEvent& se);
};
#endif

// end pushSupplier.hh

// pushConsumer.cc

#include <iostream>
#include <omniORB3/CORBA.h>
#include <COS/CosNotifyChannelAdmin.hh>
#include <unistd.h>
#include "pushSupplier.hh"

void StructuredPushSupplier_impl::disconnect_structured_push_supplier()
{
    connectionStatus = false;
}

void StructuredPushSupplier_impl::subscription_change(const CosNotification::Event
const CosNotification::EventTypeSeq& removed)
{ CORBA::ULong ix;
  for (ix=0; ix < added.length(); ix++) {
    cout << "+ " << added[ix].domain_name <<
      "::<" << added[ix].type_name << endl;
  }
  for (ix=0; ix < removed.length(); ix++) {
    cout << "- " << removed[ix].domain_name <<
      "::<" << removed[ix].type_name << endl;
  }
}

bool StructuredPushSupplier_impl::getConnectionStatus()
{
    return connectionStatus;
}

void StructuredPushSupplier_impl::setConnectionStatus(bool newStatus)
{
    connectionStatus = newStatus;
}

void StructuredPushSupplier_impl::setProxyConsumer(CosNotifyChannelAdmin::Struc
{

```

```

    prx_cons = CosNotifyChannelAdmin::StructuredProxyPushConsumer::_duplicate(the
}

// probably the change of connectionStatus is blocked because we have only one
void StructuredPushSupplier_impl::pushNow(CosNotification::StructuredEvent& se)
{

    prx_cons->push_structured_event(se);
        //      sleep(5);

}

// end pushConsumer.cc

// pushConsumer.hh

#ifndef MY_PUSHCONSUMER
#define MY_PUSHCONSUMER

class StructuredPushConsumer_impl : public virtual POA_CosNotifyComm::Structure
    public PortableServer::RefCountServantBase
    {

public:
    inline StructuredPushConsumer_impl() : num_events(0)    {;}
    void disconnect_structured_push_consumer(); //    {;}
    void offer_change(const CosNotification::EventTypeSeq& added,
        const CosNotification::EventTypeSeq& removed);
    void push_structured_event(const CosNotification::StructuredEvent& se);
    CORBA::Long get_num_events() const; //    { return num_events; }

private:
    // The following is used to keep track of the number of events pushed by t
    CORBA::Long num_events;
};

#endif

// end pushConsumer.hh

// pushConsumer.cc

// implementing the event consumer
#include <iostream.h>
#include "COS/CosNotifyComm.hh"
#include "COS/CosNotifyChannelAdmin.hh"

```

```

#include "pushConsumer.hh"

void StructuredPushConsumer_impl::disconnect_structured_push_consumer()
{;}

void StructuredPushConsumer_impl::offer_change(const CosNotification::EventType
        const CosNotification::EventTypeSeq& removed)
{ CORBA::ULong ix;
  for (ix=0; ix < added.length(); ix++) {
    cout << "+ " << added[ix].domain_name <<
      "::<" << added[ix].type_name << endl;
  }
  for (ix=0; ix < removed.length(); ix++) {
    cout << "- " << removed[ix].domain_name <<
      "::<" << removed[ix].type_name << endl;
  }
}

void StructuredPushConsumer_impl::push_structured_event(const CosNotification:
{
  CORBA::Long val;
  se.reminder_of_body >>= val;
  cout << se.header.fixed_header.event_type.domain_name << "::<"
    << se.header.fixed_header.event_type.type_name << " : "
    << val << endl;
  num_events += 1;
}

CORBA::Long StructuredPushConsumer_impl::get_num_events() const
{
  return num_events;
}

// end pushConsumer.cc

```

C.5.3 Driver Code

```

// supplierDriver.cc
// driver code for structured event supplier
// the channel initialization is done by the supplier driver

#include <iostream>
//#include <fstream>
//#include <iomanip>
#include <unistd.h>
#include <cstdlib>

```

```

/* #include <stdio.h>
 * #include <iostream.h> */
#include <omniORB3/CORBA.h>
#include <COS/CosNotifyChannelAdmin.hh>
#include "corbainit.hh"
#include "eventChannel.hh"
#include "pushSupplier.hh"

int main(int argc, char* argv[])
{
    CorbaInit* myOrbConnection = CorbaInit::initializeOrb(argc, argv);
    EventChannel myChannelInit(myOrbConnection); // c++ object which holds the ch
    CosNotifyChannelAdmin::EventChannel_var myChannelIOR = myChannelInit.getEvent
        // check whether IOR was of the assumed type
        if (CORBA::is_nil(myChannelIOR)){
            cerr << "no event channel available" << endl;
            throw 0;
        }

        // depose event channel ior for consumer side
        //
const char* myChannelfile = "/tmp/eventChannel.ior";
const char* eventChannelString = (myOrbConnection->returnOrb())->object_to_stri
    cout <<eventChannelString << endl;

    // original code:
const char* result = myOrbConnection->storeIORinFile(myChannelfile, myChannelIOR
    // out commented after type change _var->_ptr in corbainit
    // const char* result = myOrbConnection->storeIORinFile("/tmp/eventChannel.ior

        cout << "channel IOR written to " << result << endl;

    StructuredPushSupplier_impl* mySupplier = new StructuredPushSupplier_impl();

    // get the right admin interface
    // using default values for configuration of the admin interface
    CosNotifyChannelAdmin::AdminID saID; // ID's used to distinguish objects

    CosNotifyChannelAdmin::SupplierAdmin_var sadminIOR = myChannelIOR->new_for_su
        // check whether IOR was of the assumed type
        if (CORBA::is_nil(sadminIOR)){
            cerr << "no admin interface available" << endl;
            throw 0;
        }else{
            cout << "Supplier admin interface available" << endl;
        }
}

```

```

// creating the proxy consumer object
// specific type is set via parameters and subsequent narrowing
CosNotifyChannelAdmin::ProxyID pxID; // ID's used to distinguish objects
CosNotifyChannelAdmin::ProxyConsumer_var tmp_consIOR = sadminIOR->
obtain_notification_push_consumer(CosNotifyChannelAdmin::STRUCTURED_EVENT, pxID);
// check whether IOR was of the assumed type
if (CORBA::is_nil(tmp_consIOR)){
    cerr << "no proxy interface available" << endl;
    throw 0;
}else{
    cout << "obtain the proxy consumer IOR" << endl;
}

// converting type of proxy consumer into structured ...
CosNotifyChannelAdmin::StructuredProxyPushConsumer_var prx_consIOR =
CosNotifyChannelAdmin::StructuredProxyPushConsumer::_narrow(tmp_consIOR);
// check whether IOR was of the assumed type
if (CORBA::is_nil(prx_consIOR)){
    cerr << "no proxy interface available" << endl;
    throw 0;
}else{
    cout << "narrow to StructuredProxyPushConsumer successful" << endl;
}

// connecting source to proxy consumer
prx_consIOR->connect_structured_push_supplier(mySupplier->_this());
mySupplier->setProxyConsumer(prx_consIOR);

// create the structured event and
CosNotification::StructuredEvent se;
int numEvents = 100;
se.header.fixed_header.event_type.domain_name = CORBA::string_dup("READY");
se.header.fixed_header.event_type.type_name = CORBA::string_dup("Long");
se.header.fixed_header.event_name = CORBA::string_dup("");
se.header.variable_header.length(0);
se.filterable_data.length(0);

// for (CORBA::Long i = 0; i < numEvents; i++) {
long i =0;
while(true){
    se.remainder_of_body <=< i;
    mySupplier->pushNow(se);
    cout << "event number " << i << endl;
    sleep(3); // wait for 3 seconds before pushing the next event
}

```

```

        i++;
    }
}

// end supplierDriver.cc

// consumerDriver.cc
// creates StructuredPushConsumer servant, necessary interface objects and connections
#include <iostream>
#include <unistd.h>
#include <cstdlib>
#include "omniORB3/CORBA.h"
#include "COS/CosNotifyComm.hh"
#include "COS/CosNotifyChannelAdmin.hh"
#include "corbainit.hh"
#include "pushConsumer.hh"

int main (int argc, char* argv[])
{
    CorbaInit* myOrbConnection = CorbaInit::initializeOrb(argc, argv);
    CORBA::Object_var tmpChannelIOR = (myOrbConnection->returnOrb())->string_to_object("t");
    // CORBA::Object_var tmpChannelIOR = myOrbConnection->restoreIORfromFile("/tmp/channelIOR");
    CosNotifyChannelAdmin::EventChannel_var myChannelIOR = CosNotifyChannelAdmin::EventChannel::new_event_channel(myOrbConnection);
    // check whether IOR was of the assumed type
    if (CORBA::is_nil(myChannelIOR)){
        cerr << "no event channel available" << endl;
        throw 0;
    }

    StructuredPushConsumer_impl myConsumer;

    CosNotifyComm::StructuredPushConsumer_var myConsumerIOR = myConsumer._this();

    CosNotifyChannelAdmin::AdminID saID;
    CosNotifyChannelAdmin::ConsumerAdmin_var consumerAdminIOR =
        myChannelIOR->new_for_consumers(CosNotifyChannelAdmin::AND_OP, saID);

    if ( CORBA::is_nil(consumerAdminIOR)) {
        cerr << "Failed to create new ConsumerAdmin object" << endl;
        return -1;
    }
}

```



```
CosNotifyChannelAdmin::ProxyID pxID;
CosNotifyChannelAdmin::ProxySupplier_var tmp_suplIOR =
    consumerAdminIOR->obtain_notification_push_supplier(CosNotifyChannelAdmin::

CosNotifyChannelAdmin::StructuredProxyPushSupplier_var prx_suplIOR =
    CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow(tmp_suplIOR)
if ( CORBA::is_nil(prx_suplIOR) ) {
    cerr << "Failed to create PushSupplier Proxy object" << endl;
    return -1;
}
prx_suplIOR->connect_structured_push_consumer(myConsumerIOR);

//      while ( consumer->get_num_events() < numEvents )
long receivedEvents = 0;
while(true){
    omni_thread::sleep(5);
    receivedEvents = myConsumer.get_num_events(); // this is a local c++ object
    // No more events to be consumed. Disconnect the consumer and destroy the

}

}

// end consumerDriver.cc
```

D DWARF Time Service

```
// TimeService.idl

#ifndef __TIMESERVICE_IDL
#define __TIMESERVICE_IDL

#pragma prefix "in.tum.de"
#include <DWARF/Service.idl>

module DWARF{

    struct TimeOfDay {
        short hour;
        short minute;
        short second;
    };

    interface Clock {
        TimeOfDay getDayTime();
    };

    interface TimeService : Service, Clock {
    };

};
#endif

// end TimeService.idl

// TimeService.cpp

/*
 * TimeService.cpp
 * Lothar Richter, July 2002
 * Distributed Wearable Augmented Reality Framework - www.augmentedreality.de
 *
 * This file contains an example DWARF service that provides a DayTime
```

```
*/

#include <config.h>
#include <debug.h>
#include <cmdlineprops.h>

#include <string>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

#include <DWARF/Service.h>
#include <corbainit_cpp/corbainit.h>
#include <DWARF/TimeService.h>

using namespace DWARF;
using namespace CORBA;

class TimeService_impl :
public POA_DWARF::TimeService
{
string myName;

public:
    TimeService_impl(const char* name);

    virtual ~TimeService_impl(){;}

    //Service interface
    char* getStatus();

    // Clock interface
    DWARF::TimeOfDay getDayTime();

    static char* getClassName() { return "TimeService_impl"; }
    string getClassDesc() { return myName; }

    //the work
    void run();
    static void* threadStartRoutine(void* data);
};
```

```
//constructor
TimeService_impl::TimeService_impl(const char* name):
    myName(name)//,
{
    DEBUGSTREAM(1, "constructor");
    DEBUGSTREAM(1, "done");
}

//Service interface
char* TimeService_impl::getStatus()
{
    DEBUGSTREAM(1, "getStatus");
    char status[100];
    sprintf(status, "test status");
    return string_dup(status);
}

// Clock interface
DWARF::TimeOfDay TimeService_impl::getDayTime()
{
    time_t time_now = time(0);
    struct tm* time_p = localtime(&time_now);

    DWARF::TimeOfDay newDayTime;
    newDayTime.hour = time_p -> tm_hour;
    newDayTime.minute = time_p -> tm_min;
    newDayTime.second = time_p -> tm_sec;

    return newDayTime;
}

//main loop
void TimeService_impl::run()
{
    char myhostname[200];
    gethostname(myhostname, 200);
    char hello[400];
    sprintf(hello, "Hello, World, from %s, running on %s!", myName.c_str(), myhos
    while (true)
    {
        cout << " run is activated " << endl;
        sleep(3);
    }
}
```

```
    DEBUGSTREAM(1, "done");
}

//static thread starter
void* TimeService_impl::threadStartRoutine(void* data)
{
    TimeService_impl* ts=(TimeService_impl*)data;
    ts->run();
    return 0;
}

////////////////////////////////////
int main(int argc, char** argv)
{
    CorbaInit* myOrbConnection;
    cmdLineProps.init(argc, argv);
    try
    {

        DEBUGSTREAM(1, "Initializing CORBA");
        myOrbConnection = CorbaInit::initializeOrb(argc, argv);

        ServiceManager_var smgr = myOrbConnection->getServiceManager();

        ServiceDescription_var sd;
        NeedDescription_var nd;
        AbilityDescription_var ad1;
        AbilityDescription_var ad2;
        ConnectorDescription_var pd1;
        ConnectorDescription_var pd2;

        DEBUGSTREAM(1, "Describing Service");
        sd=smgr->newServiceDescription("TimeService");

        ad1=sd->newAbility("status");
        ad1->setType("Status");
        pd1=ad1->newConnector("ObjrefExporter");

        ad2=sd->newAbility("giveTime");

        ad2->setType("TimeOfDay");

        pd2=ad2->newConnector("ObjrefExporter");
        smgr->activateServiceDescription("TimeService");
    }
}
```

```
    DEBUGSTREAM(1, "Creating Service");
    bool fast=false;

    TimeService_impl* myTimeService=new TimeService_impl("TimeService");

    DEBUGSTREAM(1, "registering Service");
    myOrbConnection->registerService("TimeService", myTimeService->_this());

    DEBUGSTREAM(1, "Starting ORB");
    myOrbConnection->run();
    myOrbConnection->destroy();
}
catch(CORBA::Exception&) {
    DEBUGSTREAM(10, "Caught CORBA::Exception.");
}
catch(...) {
    DEBUGSTREAM(10, "Caught unknown exception.");
}
return 0;
}

// end TimeService.idl

// TimeService.cpp

/*
 * TimeService.cpp
 * Lothar Richter, July 2002
 * Distributed Wearable Augmented Reality Framework - www.augmentedreality.de
 *
 * This file contains an example DWARF service that provides a DayTime
 *
 */

#include <config.h>
#include <debug.h>
#include <cmdlineprops.h>

#include <string>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include <iomanip>
```

```
#include <DWARF/Service.h>
#include <corbainit_cpp/corbainit.h>
#include <DWARF/TimeService.h>

using namespace DWARF;
using namespace CORBA;

class TimeClient_impl :
public POA_DWARF::Service_ObjrefImporter
{
    string myName;
    DWARF::Clock_var myTimeService;

public:
    TimeClient_impl(const char* name);

    virtual ~TimeClient_impl(){;}

    //Service interface
    char* getStatus();

    // interface SvcProtObjrefImporter
    // void DWARF::_impl_SvcProtObjrefImporter::importObjref(CORBA::Object * obj)
    void importObjref(CORBA::Object * objref);

    // // Clock interface
    // DWARF::TimeOfDay getDayTime();

    static char* getClassName() { return "TimeClient_impl"; }
    string getClassDesc() { return myName; }

    //the work
    void run();
    static void* threadStartRoutine(void* data);
};

//constructor
TimeClient_impl::TimeClient_impl(const char* name):
    myName(name)//,
{
    DEBUGSTREAM(1, "constructor");
    DEBUGSTREAM(1, "done");
}
```

```
myTimeService = 0;
}

//Service interface
char* TimeClient_impl::getStatus()
{
    DEBUGSTREAM(1, "getStatus");
    char status[100];
    sprintf(status, "test status");
    return string_dup(status);
}

//void DWARF::_impl_SvcProtObjrefImporter::importObjref(CORBA::Object * objref)
void TimeClient_impl::importObjref(CORBA::Object * objref)
{
    myTimeService = DWARF::Clock::_narrow(objref);
    // check whether IOR was of the assumed type
    if (CORBA::is_nil(myTimeService)){
        cerr << "argument is not a TimeService reference" << endl;

        DEBUGSTREAM(1, "the given objectref was not a TimeService-IOR");
        exit(25);
        //      throw 0;
    }
}

;

//main loop
void TimeClient_impl::run()
{
    while (true)
    {
        cout << " run is activated " << endl;
        if (myTimeService == 0)
        { cout << " no time service found yet" << endl;
        continue; // prevents immature termination because connection needs some time
        };

        DWARF::TimeOfDay tod = myTimeService->getDayTime();
        cout << "actual time is: "
        << setw(2) << setfill('0') << tod.hour << ":"
        << setw(2) << setfill('0') << tod.minute << ":"
        << setw(2) << setfill('0') << tod.second << endl;
    }
}
```



```
        sleep(3);
    }

    DEBUGSTREAM(1, "done");
}

//static thread starter
void* TimeClient_impl::threadStartRoutine(void* data)
{
    TimeClient_impl* ts=(TimeClient_impl*)data;
    ts->run();
    return 0;
}

////////////////////////////////////
int main(int argc, char** argv)
{
    CorbaInit* myOrbConnection;
    cmdLineProps.init(argc, argv);
    try
    {

        DEBUGSTREAM(1, "Initializing CORBA");
        myOrbConnection = CorbaInit::initializeOrb(argc, argv);

        ServiceManager_var smgr = myOrbConnection->getServiceManager();

        ServiceDescription_var sd;
        NeedDescription_var nd1;
        AbilityDescription_var ad1;
        ConnectorDescription_var pd1;
        ConnectorDescription_var pd2;

        DEBUGSTREAM(1, "Describing Service");

        sd=smgr->newServiceDescription("TimeClient");

        ad1=sd->newAbility("status");
        DEBUGSTREAM(1, "got newAbility ad1");
        ad1->setType("Status");
        pd1=ad1->newConnector("ObjrefExporter");
        DEBUGSTREAM(1, "geot newConnector pd1");
```

```
    nd1 = sd->newNeed("giveTime");
    DEBUGSTREAM(1, "got newNeed nd1");
    nd1->setType("TimeOfDay");
    pd2=nd1->newConnector("ObjrefImporter");
    DEBUGSTREAM(1, "got newConnector pd2");
    smgr->activateServiceDescription("TimeClient");

    DEBUGSTREAM(1, "Creating Service");
    bool fast=false;

    TimeClient_impl* myTimeClient = new TimeClient_impl("TimeClient");

    DEBUGSTREAM(1, "registering Service");
    myOrbConnection->registerService("TimeClient", myTimeClient->_this());

    DEBUGSTREAM(1, "startService");
    (new omni_thread(&TimeClient_impl::threadStartRoutine, myTimeClient))->start();
    DEBUGSTREAM(1, "thread started");

    DEBUGSTREAM(1, "Starting ORB");

    myOrbConnection->run();
    myOrbConnection->destroy();

}
catch(CORBA::Exception&) {
    DEBUGSTREAM(10, "Caught CORBA::Exception.");
}
catch(...) {
    DEBUGSTREAM(10, "Caught unknown exception.");
}
return 0;
}

// end TimeClient.cpp
```

E Helper Class Corbainit

```
/*
$Revision: 1.1 $ written by Lothar Richter $Date: 2002/08/07 09:55:01 $
as part of the DWARF project

Technische Univertt Mnchen
*/

#ifndef CORBAINIT
#define CORBAINIT

class CorbaInit
{
private:
    static CorbaInit* theSingleton;

    // instance variables
    CORBA::ORB_var theOrb;
    PortableServer::POA_var thePoa;
    DWARF::ServiceManager_var theServiceManager;
    // member functions
    CorbaInit(int argc, char *argv[]);

public:
    // access to the basic corba objects is mediatied by Tutlib-object methods
    // usefule functions to handle IOR's

    static CorbaInit* initializeOrb(int argc, char* argv[]);

    // service functions
    CORBA::ORB_var getOrb();
    PortableServer::POA_var getPoa();

    const char* storeIORinFile(const char* outputFile, CORBA::Object_ptr inputIOR);
};

#endif
```

E Helper Class Corbainit

```
CORBA::Object_ptr restoreIORfromFile(const char* infile);

DWARF::ServiceManager_ptr getServiceManager();

void registerService(const char* serviceID, DWARF::Service_ptr serviceIOR);
void run();
void destroy();
};

#endif

/*
$Revision: 1.1 $ written by Lothar Richter $Date: 2002/08/07 09:55:01 $
as part of the DWARF project

Technische Univeritt Mnchen
*/

#include <unistd.h>
#include <iostream>
#include <stdlib.h>
#include <string>
#include <fstream>
#include <debug.h>
#include <cstdlib>
#include <omniORB3/CORBA.h>
#include <cmdlineprops.h>
#include <DWARF/Service.h>
#include <cmdlineprops.h>
#include "corbainit.h"

// first initialization, no orb init yet
// if you do not initialize explicitly you will get linking problems

CorbaInit* CorbaInit::theSingleton = 0;

// public methods
```

```
// static method
CorbaInit* CorbaInit::initializeOrb(int argc, char *argv[])
{
    if (theSingleton == 0){
        theSingleton = new CorbaInit(argc, argv);
        return theSingleton;
    }else{
        return theSingleton;
    }
};

// member functions

// using the constructor to initialize necessary corba objects
CorbaInit::CorbaInit(int argc, char *argv[])
{
    // parsing the command-line into key/value pairs
    cmdLineProps.init(argc, argv);

    try {
        // initialize orb
        theOrb = CORBA::ORB_init(argc, argv);
        DEBUGSTREAM(10, "ORB_init successful done");
        // Get reference to root POA
        // necessary for server role
        CORBA::Object_var obj = theOrb->resolve_initial_references ("RootPOA");
        //PortableServer::POA_var
        thePoa = PortableServer::POA::_narrow(obj);
        DEBUGSTREAM(10, "RootPOA successful resolved and narrowed");
        // Activate POA manager
        PortableServer::POAManager_var mgr = thePoa->the_POAManager();
        mgr->activate();
        DEBUGSTREAM(10, "POA manager activated");
        // determine host on which to start the service manager
        const char* svcMgrHost = 0;
        if (cmdLineProps.find("serviceManager")){
            svcMgrHost = cmdLineProps.get("serviceManager");
        }else{svcMgrHost = "localhost";}
        DEBUGSTREAM(10, "command line resolution done");
        std::string svcMgrInitStr ("corbaloc::");
        svcMgrInitStr = svcMgrInitStr + svcMgrHost + ":39273/ServiceManager" ;
        DEBUGSTREAM(10, "ServiceManagerHostString created");
        // cout << svcMgrInitStr << endl;
        // theServiceManager = DWARF::ServiceManager::_narrow(theOrb->string_to_
        theServiceManager = DWARF::ServiceManager::_narrow(theOrb->string_to_object
//FIXME: check if nil
```

```
    if (CORBA::is_nil(theServiceManager)){ // check for valid IOR
        cerr << "no service manager reference" << endl;
        throw CORBA::OBJECT_NOT_EXIST();
    }

    DEBUGSTREAM(10, "service manager resolved and narrowed");
} catch(CORBA::Exception&) {
DEBUGSTREAM(10, "initialization of CorbaInit failed");
exit(25);
}
};

// delivers orb object
CORBA::ORB_var CorbaInit::getOrb(){
    return theOrb;
};

// delivers service manager object
DWARF::ServiceManager_ptr CorbaInit::getServiceManager(){
    return theServiceManager;
};

// delivers poa object
PortableServer::POA_var CorbaInit::getPoa(){
    return thePoa;
};

const char* CorbaInit::storeIORinFile(const char* outputFile, CORBA::Object_ptr
{
    // open output file
    std::ofstream iorFile(outputFile);
    if (!outputFile){
        throw "cannot open file \"" + string(outputFile) + "\" to write";
    }

    // stringify IOR

    const char* outString = theOrb->object_to_string(inputIOR);
    const char* c = outString;
    while (*c!=0){
        iorFile.put(*c);
        c++; // classic c-style
    }
}
```

```
    }

    return outputFile;
}; // closes file automatically

CORBA::Object_ptr CorbaInit::restoreIORfromFile(const char* infile)
{
    string tmpString;
    std::ifstream iorFile(infile);
    if (!iorFile){
        throw "cannot open file \"" + string(infile) + "\"" to read";
    }
    char c;
    while (iorFile.get(c)) {
        tmpString += c;
    }

    //tell orb to restore IOR back from paramater string
    CORBA::Object_ptr tmpVar = theOrb->string_to_object(tmpString.c_str());

    if (CORBA::is_nil(tmpVar)){ // check for valid IOR
        cerr << "Nil Sample reference" << endl;
        throw 0;
    }
    return tmpVar;
} // closes file automatically

void CorbaInit::registerService(const char* serviceID, DWARF::Service_ptr servi
{
    theServiceManager->registerService(serviceID, serviceIOR);
}
void CorbaInit::run(){theOrb->run();}
void CorbaInit::destroy(){theOrb->destroy();}
```

F Glossary

API. *see* APPLICATION PROGRAMMER'S INTERFACE

AR. *see* AUGMENTED REALITY

Application Programmer's Interface. "Set of fully specified operations provided by a subsystem" [1]

Augmented Reality. A technique that uses virtual objects to enhance the user's perception of the real world.

CORBA. Common Object Request Broker Architecture. CORBA is a specification for a system whose objects are distributed across different platforms. The implementation and location of each object are hidden from the client requesting the service.

Class Diagram. UML class diagrams depict associations, references and inheritance relationships between classes. They also represent the attributes and operations of individual classes.

GNU. GNU's Not Unix. An initiative to reimplement the most common UNIX tools on an open source basis.

GPL. GNU Public Licence. A software license developed by the GNU initiative that allows redistribution of software in source code but restricts the rights of the licensee to commercialize software derived from the original code.

IDL. *see* INTERFACE DEFINITION LANGUAGE

Interface Definition Language. A language that allows the programming language independent specification of software interfaces. It is used to describe the interfaces of CORBA objects.

Middleware. A piece of software that is used to combine various subsystems of a software system.

OMG The Object Management Group. This institution has specified the OMG architecture model with its components like CORBA.... <http://www.omg.org>

ORB. Object Request Broker. An ORB is at the heart of a CORBA system. Every process communicating via CORBA must have its own ORB running.

RAD. Requirements Analysis Document. A document describing the requirements of a software project and the way they were derived.

SDD. System Design Document. A document describing the general design of a software system. It serves as a basis for the implementation.

Sequence Diagram. “UML notation representing the behavior of the system as a series of interactions among a group of objects. Each object is depicted as a column in the diagram. Each interaction is depicted as an arrow between columns.” [1]

UML. *see* UNIFIED MODELING LANGUAGE

Unified Modeling Language. “A standard set of notations for representing models.” [1]. See [4] for details.

Use Case Diagram. UML notation to represent the functionality of a system.

VR. *see* VIRTUAL REALITY

VRML. Virtual Reality Markup Language. Allows the convenient description of virtual objects and scenes for AR and VR applications.

Virtual Reality. A computer based technology that allows its user to act in purely virtual environments.

WaveLAN. A midrange wireless communication standard used to replace common Ethernet connections.

XML. Extensible Markup Language. XML is a simple, standard way to delimit text data with so-called tags. It can be used to specify other languages, their alphabets and grammars.

Bibliography

- [1] B. BRÜGGE and A. H. DUTOIT, *Object-Oriented Software Engineering. Conquering Complex and Changing Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.
- [2] F. BUSCHMANN, R. MEUNIER, H. ROHNERT, P. SOMMERLAD, and M. STAL, *A System of Patterns*, John Wiley & Sons, Chichester, GB, 1996.
- [3] S. V. MICHI HENNING, *Advanced CORBA Programming with C++*, Addison-Wesley, Reading, MA, USA, 1999.
- [4] J. RUMBAUGH, I. JACOBSON, and G. BOOCH, *The Unified Modeling Language Reference Manual*, Addison Wesley, Reading, MA, 1999.
- [5] A. SAYEGH, *CORBA 2. aktualisierte und erweiterte Auflage*, O'Reilly, Köln, D, 1999.