Einführung
in
CORBA

Die Common Object Request Broker Architecture

der Object Management Group

Gerald Brose

Überblick

- Die Object Management Group
  - Wer, wie und warum?
- Die Object Management Architecture
  - Die CORBAfacilities
  - Die CORBAservices
  - Der Object Request Broker
    1 Objektmodell
    2 Architektur
    3 Objektaufruf
    4 Objektadapter

– Interface Definition Language

1 IDL Typen

2 Attribute und Operationen

3 Sichtbarkeit (Scoping)

4 Language Mapping

– Interoperabilität

1 ORB Domains, Bridging

2 IOR

3 GIOP

Die OMG (Object Management Group)

- gegründet 1989 von 8 Firmen (3Com, Canon, HP, Sun, Unisys, Philips, American Airlines, Data General) als unabhängige *non–profit* Organisation

- heute das weltgrößte Software–Konsortium mit über 600 Mitgliedern

  – aus Industrie: IBM, NCR, Novell, NeXT, Oracle, Siemens Nixdorf, Silicon Graphics, TI, Time Warner, Digital Equipment, HP, AT&T, Microsoft, Motorola, Xerox, ...

  – Universitäten: HU Berlin, TU Berlin, TU Dresden, Uni Hamburg, TU Ilmenau, Uni Karlsruhe, Uni München, Uni Stuttgart, Trinity College Dublin, MIT, ...

⇒ `http://www.omg.org`

OMG Standards

Die OMG verabschiedet Spezifikationen als Standards.

- Task Forces (Arbeitsgruppen) erarbeiten *Requests for Proposal* (RFPs)
- OMG–Mitglieder reichen Vorschläge ein
- Auswertung der eingereichten Vorschläge, evtl. Überarbeitung durch einreichende Partei
- Wahlverfahren:
  Task Force $\Rightarrow$ Technical Committee $\Rightarrow$ Board of Directors.
- Verabschiedung eines Vorschlags als OMG–Standard

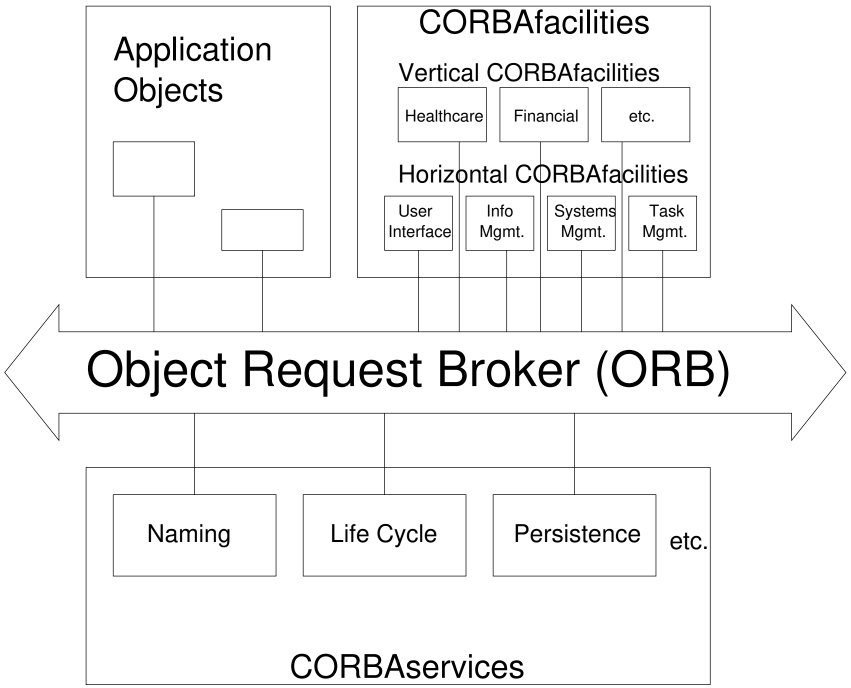
Wessen Vorschlag als Standard verabschiedet wird, der ist verpflichtet, innerhalb eines Jahres eine Implementierung auf den Markt zu bringen!

Ziele

> “OMG was formed to create a component–based software marketplace by hastening the introduction of standardized software. The organization’s charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Conformance to these specifications will make it possible to develop a heterogeneous computing environment across all major hardware platforms and operating systems.“ (http://www.omg.org/bacgrnd.htm)

$\Rightarrow$ “Integration“

Object Management Architecture



CORBAfacilities

(auch Common Facilities)

Horizontal: nützlich für fast alle Anwendungen

- User Interface (Präsentation von Compound Documents)
- Information Management (Austausch)
- Systems Management (Management Tools)
- Task Management (Workflow, Agents, Automation)

Vertikal: Einteilung nach Marktsegmenten, werden von der OMG in Zusammenarbeit mit Special Interest Groups für das jeweilige Arbeitsfeld erarbeitet.

- Gesundheitsfürsorge
- Finanzdienstleistungen

CORBAservices

(auch Common Object Services, COSS)

Bisher spezifiziert:

- Lyfecycle
- Relationship
- Naming
- Persistence
- Externalization
- Event
- Object Query

- Concurrency
- Transaction
- Licensing
- Object Properties

Noch in Arbeit:

- Trader
- Security
- Secure Time
- Object Collection
- Object Startup

CORBA - Die Infrastruktur

Release 1.1, (1991), aktuell ist Release 2.0, (1994)

CORBA

- ist “Middleware“
- ist der Kern des OMG–Referenzmodells (“Objekt–Bus“)
- erlaubt Client–Server Beziehungen zwischen Objekten (ROI, remote object invocation)

CORBA–Implementierungen:

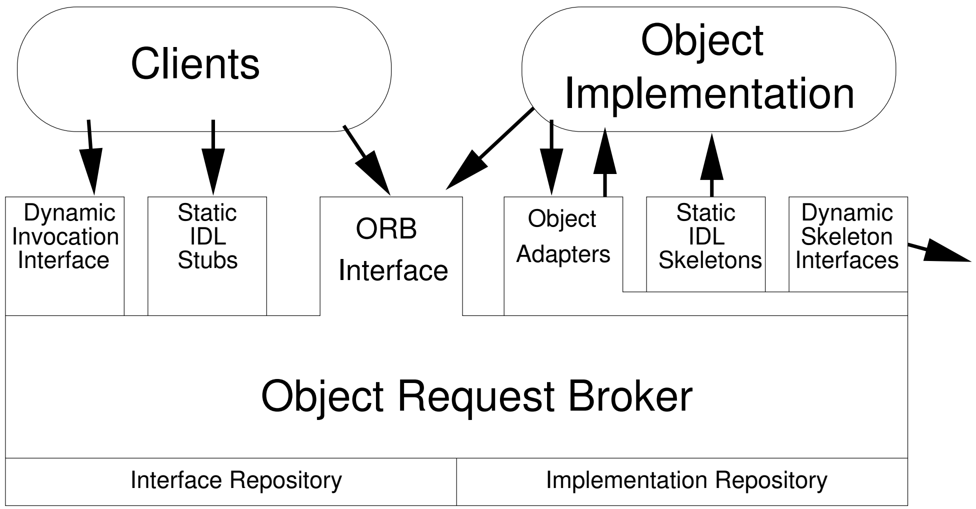
IONA Orbix, Visigenic VisiBroker (früher ORBeline), Sun NEO, IBM SOM, Digital ObjectBroker, HP ORB Plus ...

CORBA - Das Objektmodell

CORBA Objekte:

- sind gekapselt
- sind identifizierbar (Identität)
- bieten *Dienste* an, die als Operationen aufgerufen werden können
- Objektwerte sind opake Referenzen (keine Adressen!) $\Rightarrow$ Transparenz
- Objekttypen sind Schnittstellen, keine Klassen (Operationen und Attribute werden in IDL spezifiziert)
- Vererbung: overriding, Mehrfachvererbung
- Exceptions

CORBA - Die Architektur

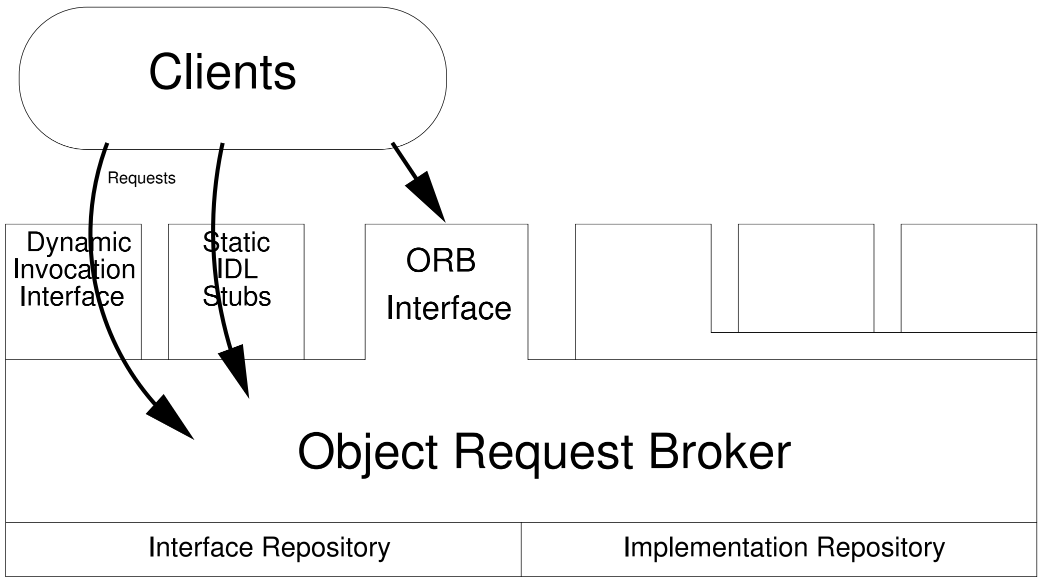


- Client Stubs, Server Skeletons und Object Adapter

- Dynamic Invocation Interface,
  Dynamic Skeleton Interface

- ORB Interface, ORB–Kern und Repositories

Objektaufruf aus Client–Sicht

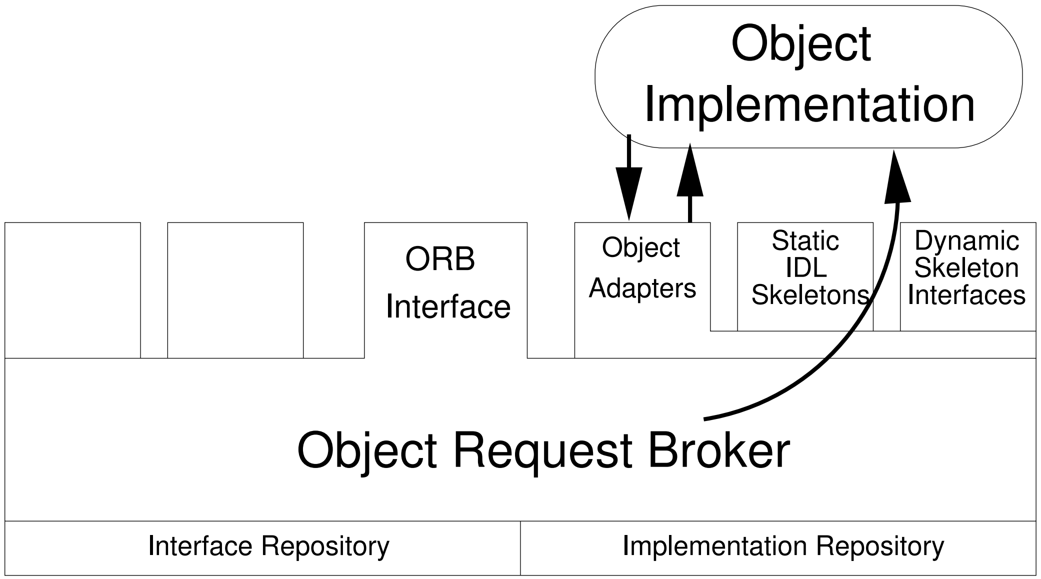


Objektaufruf durch einen Client mit einer Objektreferenz

- über statisch generierte Stubs (aus IDL Schnittstelle)
  $\Rightarrow$ Objekttyp statisch bekannt

- mittels des *Dynamic Invocation Interface* (DII)
  $\Rightarrow$ Objekttyp dynamisch ermitteln (via Trading Service)

Objektaufruf aus Server–Sicht



Aufrufweiterleitung an die Objetimplementierung durch den ORB:

- via statisch generierte Skeletons (aus IDL Schnittstelle)
- über das *Dynamic Skeleton Interface* (DSI)

Der Object Adapter

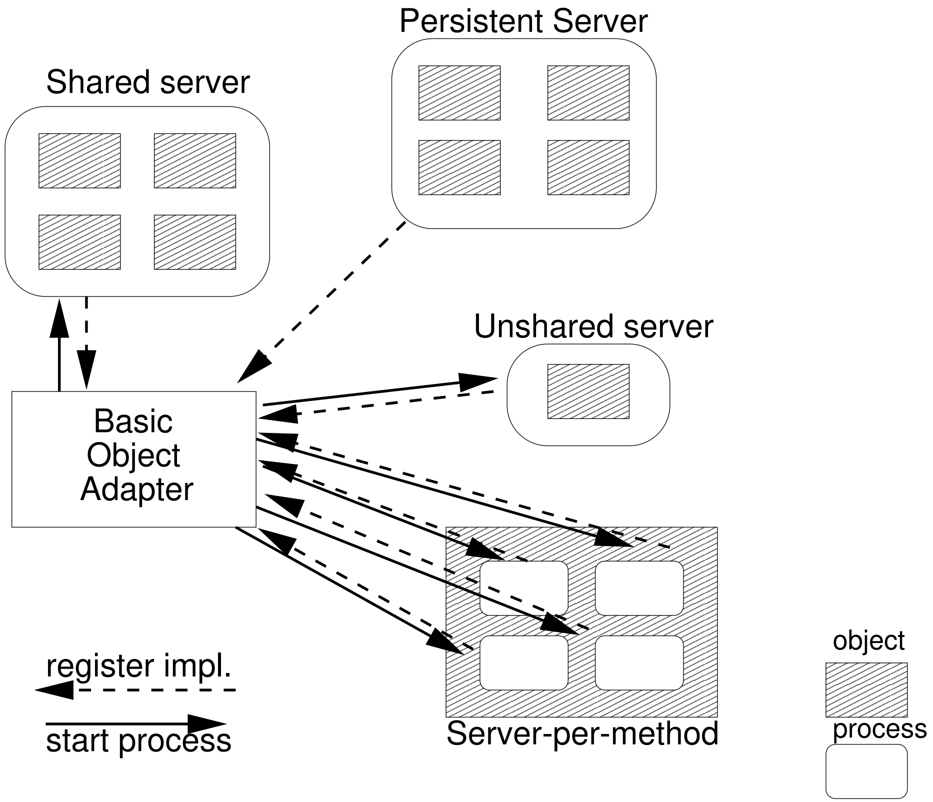
Client–Sicht: Objekte werden durch Objektreferenz identifiziert und sind immer bereit, Aufrufe entgegenzunehmen – das muß aber nicht immer tatsächlich so sein.

Wie werden Objekte gefunden und auf welche Art aktiviert? ($\neq$ erzeugt!)

- Für jedes Objekt ein Server–Prozeß?
- Für jeden Objekttyp ein Server–Prozeß?
- Für jeden Aufruf ein Server–Prozeß?
- Objekte auch manuell startbar? (persistent server)

$\Rightarrow$ Aufgabe des Objekt Adapters

Der Basic Object Adapter (BOA)



der einzig spezifizierte Object Adapter (im Moment)

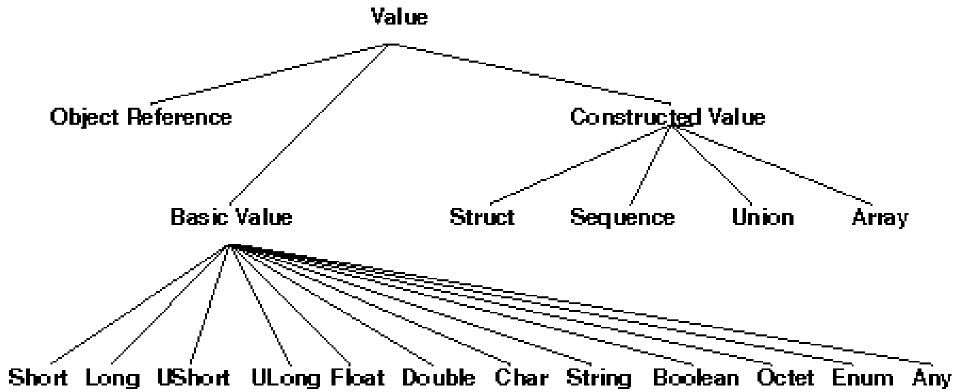
Interface Definition Language

- deklarative Sprache zur Spezifikation von Objektschnittstellen
  (Attribute und Operationen)
- “a subset of C++ with additional keywords“
- erlaubt C++ preprocessing
- strikte Typisierung
- programmiersprachenunabhängig ⇒ Für die Implementierungssprache muß eine IDL–Abbildung definiert sein (Language Mapping)

Beispiel:

```
#include "Patterns.idl"

module Cyberspace {

   typedef string Description;

   interface Inspectable {
      Description inspect();
   };

   typedef sequence <Inspectable> Inspectables;

   interface Entity: Inspectable,
                     Patterns::Observable {};

   interface Person: Entity,
                     Patterns::Observer {};

   interface Thing: Entity {};

}; // end module
```


IDL Typen



- nicht alle Werte sind Objektwerte!
  (Objekttypen versus Basis–Typen vs. Strukturierte Typen)
- `Any` für dynamische Typisierung
- Wertebereiche und Darstellung von Basistypen sind spezifiziert

Attribute in IDL

Beispiel:

```
interface Hallo {
   attribute unsigned long count;
   readonly attribute TypeCode element_type;
};
```

Syntax:

```
<attr_decl> ::= ["readonly"] "attribute"
   <param_type_spec> <simple_declarator>
   {"," <simple_declarator>}*
```

Die Deklaration eines Attributes ist logisch äquivalent mit der Vereinbarung zweier Zugriffsoperationen für den Attributwert: einer zum Setzen und einer zum Auslesen des Wertes.

`readonly` bedeutet, daß keine Operation zum Setzen des Werts definiert wird.

```
interface Hallo {
   unsigned long get_count();
   void set_count( in unsigned long );
   TypeCode get_element_type();
};
```


Operationen in IDL

Syntax:

```
<op_decl> ::= [ <op_attribute>] <op_type_spec>
              <identifier> <parameter_decls>
              [<raises_expr>] [<context_expr>]

<op_attribute> ::= "oneway"

z.B:          oneway void update();
```

Operationsaufruf–Semantik:

- “exactly once“, wenn keine Exception auftritt
- “at most once“, wenn eine Exception auftritt
  (“at most once“ bedeutet keine Garantie, daß der Aufruf stattfindet)
- “at most once“, wenn `oneway` angegeben wird
  (`void` Resultat, keine Ausgabe–Parameter, keine `raises`-Klausel)
  ⇒ in einigen Implementierungen werden `oneway`–Operationen asynchron aufgerufen.

Parameter:

```
<op_decl> ::= [ <op_attribute>] <op_type_spec>
              <identifier> <parameter_decls>
              [<raises_expr>] [<context_expr>]
...
<parameter_decls> ::= "("<param_dcl>
                      {"," <param_dcl>}* ")"
<param_dcl> ::= <param_attrib><param_type_spec>
                <simple_declarator>
<param_attrib> ::= "in" | "out" | "inout"
```

Beispiel:

```
    void dispose (in Object obj);
```

- Parameterdeklarationen haben immer einer Richtungsangabe
  (`in, out, inout`)
- Objekte werden immer als Referenzen übergeben (*by reference*), alles andere immer als Werte (*by value*) - auch `structs`.

Exceptions:

```
<op_decl> ::= [ <op_attribute>] <op_type_spec>
              <identifier> <parameter_decls>
              [<raises_expr>] [<context_expr>]
...
<raises_expr>  ::= "raises" "("<scoped_name>")"
                    {"," <scoped_name> }* ")"
```

Beispiel:

```
interface ORB {
   Object resolve_initial_references (
      in ObjectId identifier)
         raises (InvalidName);
}; // end interface
```

- wenn eine Exception auftritt, sind die Werte von `out` und `inout`–Parametern undefiniert
- Außer den angegebenen Exceptions können immer sog. “Standard Exceptions“ auftreten (z.B. NO_MEMORY). Diese dürfen nicht in einer `raises`–Klausel angegeben werden.

Sichtbarkeitsregeln:

- Alle in IDL deklarierten Namen sind zunächst nur in ihrem jeweiligen Sichtbarkeitsbereich sichtbar und müssen innerhalb dieses Bereichs eindeutig sein.
- In IDL werden Sichtbarkeitsbereiche durch Dateien, Module, Interfaces, Strukturen, Unions, Operationen und Exceptions definiert.
- Ein unqualifizierter Name wird zunächst nur in seinem lokalen Sichtbarkeitsbereich gesucht, dann in dem jeweils umschließenden.
- Namen werden mit dem Skopusoperator `::` zu qualifizierten Namen verbunden.
- Beginnt ein Name mit `::`, so beginnt die Namensauflösung im äußersten Sichtbarkeitsbereich (der Datei).

Beispiel:

```
typedef string t;
module idltest{
   module Example {
     typedef char t;
     interface One {
       typedef short t;
       attribute t a; // short
       attribute Example::t b; // char
       attribute ::t c; // string
     };
     module Another{
       typedef octet t;
       interface Two {
          typedef long t;
          attribute t a; // long
          attribute Example::Another::t b;
          attribute idltest::Example::One::t c;
       }; // end Two
     }; // end Another
   };// end Example
};// end idltest
```

Language Mappings

Abbildungen von einer in IDL spezifizierten Schnittstelle auf eine Implementierung in einer bestimmten Programmiersprache. (realisiert durch IDL–Compiler und ORB–Runtime)

Abbildungen notwendig für

- alle IDL Typen
- Semantik von Operationsaufruf und Parameterübergabe
- Attributwerte
- Exceptions
- Sichtbarkeitsregeln

CORBA standardisiert *Language Mappings* für C, C++ und Smalltalk Ada und Java sind in Arbeit, COBOL wird erwartet.

Java Language Mapping

OMG RFP August 1996,
aktuell: diverse nicht–standardisierte Mappings (SunSoft, IONA, Visigenic, JacORB).

Abbildungen für IDL–Datentypen

1. Basistypen:

IDL:

```
boolean, char, octet,
long/unsigned long, short/unsigned short,
float, double,
string
```

Java:

```
boolean, char, byte,
int, short,
float, double,
java.lang.String
```

1a Konstanten:

IDL: `const long aLong = 3123444;`

Java hat keine Konstanten ⇒ Abbildung auf `static final` Variablen von `final` deklarierten Java–Klassen:

```
public final class aLong {
   public static final int value = 3123444;
}
```

1b Aufzählungen:

IDL: `enum colour { Red, Green, Blue };`

Java hat keine Aufzählungstypen ⇒ Abbildung auf `static final` Integer–Variablen von Java–Klassen:

```
public class colour {
   public static final int Red = 0;
   public static final int Green = 1;
   public static final int Blue = 2;
}
```


2. strukturierte Typen:

Arrays und Sequences:

IDL Arrays und Sequences werden auf Java–Arrays abgebildet.

Interfaces:

IDL Interfaces werden auf Java–Interfaces abgebildet, Attribute auf *Accessor–Methods*:

IDL:

```
interface grid2{
    attribute long q;
    readonly attribute string s;
};
```

Java:

```
public interface grid2{
    int get_q();
    void set_q(int a);
    java.lang.String get_s();
}
```

Structures und Unions:

IDL hat neben Objektinterfaces auch noch `struct` und `union`:

> "By definition, a `struct` is a class where members are by default public ... . A named union is defined as a `struct` where every member has the same address. If one knows that only one member of a structure will have a useful value at any one time, union can save space." (B. Stroustrup, *The C++ Programming language*, 2nd ed., 1991, p. 167.)

⇒ Abbildung auf Java–Klassen.

Structures:

IDL:

```
struct try_me {
   short a;
   long b;
   char c[7], d;
};
```

Java:

```
public final class try_me {
   public short a;
   public int b;
   public char c[], d;

   public try_me( short a, int b,
                  char c[], char d)
   {
      this.a = a;
      this.b = b;
      this.c = c;
      this.d = d;
   }
}
```

Unions:

IDL:

```
union unionTest switch (short) {
   case 1: long l;
   default: char c;
};
```

Wird als äquivalent zu
`typedef jacorb::Idl::Union unionTest`
angesehen, d.h. Variablen von diesem Typ werden in Java zu Instanzen der Klasse Union:

```
public class Union {
   public Object element = null;
   public Union(){}
   public Union( Object o ){
      element = o;
   }
}
```


Typedefs:

In IDL führt das Schlüsselwort `typedef` einen neuen Typnamen ein. Java hat kein `typedef`, d.h. der IDL–Compiler muß den neuen Typnamen jeweils durch seine Definition ersetzen.

IDL:

```
interface x {
   typedef char aType;
   typedef struct newStruct{ aType c;} newType;
   attribute newType n;
};
```

Java:

```
public final class newStruct {
        public char c;
        public newStruct( char c){
                this.c = c;
        }
        public newStruct(){}
}
public interface x {
        idltest.newStruct get_n();
        void set_n(idltest.newStruct a);
}
```

Operationen, Parameter, Exceptions

Operationsdeklarationen nur in Interfaces. Operationen sind (potentiell) Fernaufrufe.

Parametermechanismen:

Java unterscheidet nicht zwischen `in`, `out`, `inout`–Parametern. `in`–Parameter entsprechen der “normalen“ Paramterübergabe, für `out` und `inout`–Parameter müssen *Halterobjekte* verwendet werden.

IDL:

```
interface grid2{
    void write( in string s);
    long read( out string buf);
};
```


Java:

```
public interface grid2{
    void write(java.lang.String s);
    int read(jacorb.Orb.StringHolder buf);
}
```

Exceptions:

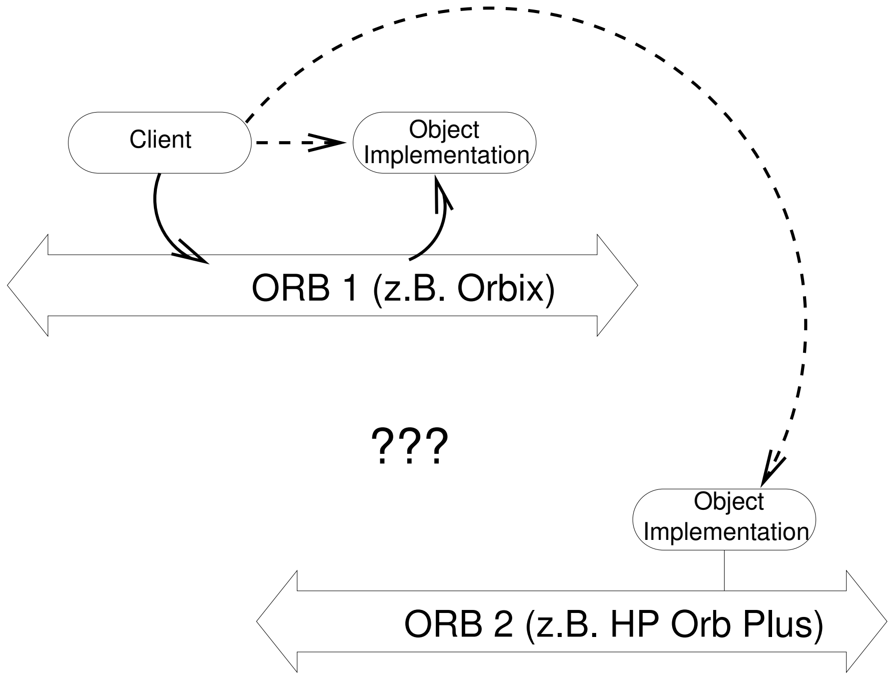
IDL Exceptions werden auf Java–Exceptions abgebildet.

Sichtbarkeitsregeln:

In Java werden Sichtbarkeitsbereiche nur durch Packages und Klassen/Interfaces definiert, außerdem lassen sich Typen nicht lokal zu Klassen/Interfaces deklarieren.

⇒ Hierarchien von Namensräumen werden flacher.

Interoperabilität



ORB–Domains

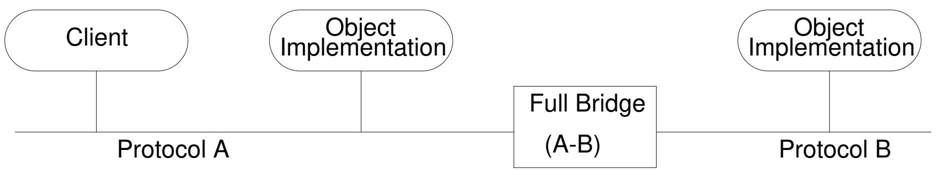
Warum überhaupt verschiedene ORB–Domains?

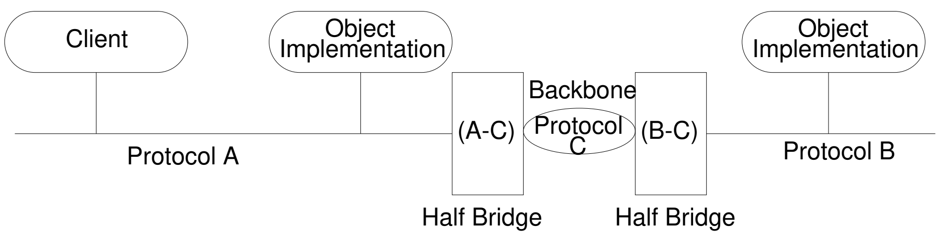
- administrative Grenzen (z.B. zwischen Abteilungen)
- technologische Grenzen (zwischen heterogener Hard– und Software)
- Sicherheitsgrenzen (z.B. zwischen “Top Secret“ und “Confidential“)

Bridging

das Überbrücken von Domain–Grenzen,
Abbildung von Konzepten einer Domain in Konzepte einer anderen Domain.

- immediated bridging (full bridges)
- mediated bridging (half bridges)





Inter–ORB Kommunikation in CORBA

CORBA spezifiziert:

- *Interoperable Object Reference* (IOR)
- *General Inter–ORB Protocol* (GIOP)
  - *Common Data Representation* (CDR)
  - Nachrichtenformate
- *Internet Inter–ORB Protocol* (IIOP)
  $\Rightarrow$ GIOP über TCP/IP

Optional können andere verbindungsorientierte Transportprotokolle definiert werden, um GIOP zu implementieren, z.B. IPX, OSI X.25.

Außerdem kann zusätzlich zu GIOP ein eigenes, umgebungsspezifisches IOP definiert werden. In CORBA 2.0 wird ein solches ESIOP via DCE RPC spezifiziert.

Interoperable Object Reference (IOR)

- Objektreferenzen sind *opake Handles*

- ihre Repräsentation implementierungsabhängig

⇒ außerhalb des eigenen ORBs nicht interpretierbar - also nicht interoperabel!

IOR zur ORB–unabhängigen Repräsentation von Objektreferenzen

IORs sind ein reines Bridging–Konstrukt:

- Verwendung nur ORB–intern

- für die Anwendung völlig transparent

- ORBs erzeugen domain–lokale Proxies

Eine IOR enthält eine *TypeId* sowie eine Folge von *Tagged-Profiles*, die protokollspezifische Daten für den Objektaufruf enthalten.

```
// IOP module, chap. 10 CORBA V2,7-95
module IOP {
   typedef unsigned long  ProfileId;
   const ProfileId TAG_INTERNET_IOP = 0;
   ...

   struct TaggedProfile {
      ProfileId  tag;
      sequence <octet> profile_data;
   };

   // an Interoperable Object Reference
   struct IOR {
      string type_id;
      sequence <TaggedProfile> profiles;
   };

   ...
};
```

General Inter–ORB Protocol (GIOP)

Annahmen über das Transportprotokoll ($\Rightarrow$ TCP):

- verbindungsorientiert
- zuverlässig
- Transportverbindungen können als *byte streams* angesehen werden
- "vernünftige" Benachrichtigung über Verbindungszusammenbruch
- Klienten initiieren Verbindungen
- Server warten auf Verbindungsanfragen (initiieren keine) und akzeptieren ggf. Verbindungsanfragen
  $\Rightarrow$ Verbindungen sind asymmetrisch

Common Data Representation (CDR):

- ist die Transfer Syntax, in der GIOP die IDL Datentypen als *Byte Stream* darstellt.
- erlaubt *big–endian* wie *little–endian byte ordering*
- spezifiziert die Übertragung von Werten aller Typen
- Beispiel Basistypen:
  - an Bytegrenzen ausgerichtet
  - Integer sind 16 oder 32 Bit,
  - Floats sind 32 Bit, Doubles 64 (nach IEEE standard)
  - etc.

Nachrichtenformate:

Alle GIOP–Nachrichten beginnen mit einem GIOP *Message Header*, gefolgt von einem eigenen Header für jeden Nachrichtentyp (bis auf *CloseConnection* und *MessageError*). Dann folgt der Nachrichteninhalt.

GIOP kennt sieben Nachrichtentypen: (Sender)

- Request (Client)
- Reply (Server)
- CancelRequest (Client)
- LocateRequest (Client)
- LocateReply (Server)
- CloseConnection (Server)
- MessageError (beide)


```
// GIOP module described in CORBA V2, 7-95
// chap 12
module GIOP {
   enum MsgType {
      Request, Reply, CancelRequest,
      LocateRequest, LocateReply,
      CloseConnection, MessageError
   };

   struct Version {
      char major; // 1
      char minor; // 0
   };

   struct MessageHeader {
      char magic [4];     // 'G''I''O''P'
      Version GIOP_version;
      boolean byte_order;   // 0: big-endian
      octet message_type;   // s.o. enum vals
      unsigned long message_size;
   };
...
};
```

Beispiel Request–Nachricht:

```
struct RequestHeader {
   IOP::ServiceContextList service_context;
   unsigned long request_id;  // Zuordnung
   boolean response_expected; // oneway: false
   sequence <octet> object_key; // object id
   string operation;          // e.g. "_get_x"
   CORBA::Principal requesting_principal;
};
```

Der Nachrichtenrumpf enthält alle `in` und `inout`–Parameter und ein optionales *Context*–Objekt.

```
double example(in short m, out string str,
inout Principal p);
```

wäre äquivalent zu der Struktur:

```
struct example_body {
   short m; // linker in-Parameter
   Principal p; // naechster inout-Paramter
};
```


IIOP

Transportprotokoll: TCP Abbildung aller GIOP–Nachrichten auf TCP–Verbindungen

```
// Complete IDL for IIOP module:
// CORBA V2, 7-95 p 12-31
module IIOP {
   struct Version {
      char major;
      char minor;
   };

   // s. "profile_data" in IOR tagged profile!
   struct ProfileBody {
      Version iiop_version; // 1.0
      string host;          // IP address
      unsigned short port;  // TCP port no.
      sequence <octet> object_key; // oid
    };
};
```