Verteilte Systeme

Aufgabe 4

Beispielcode: Corba-Server

- Name Binding (Server):

```
ServerImpl server = newServerImpl();
orb.connect(server);  //POA Aktivierung
org.omg.CORBA.Object nameservice =
     orb.resolve_initial_references("NameService");
NamingContext namingcontext =
     NamingContextHelper.narrow(nameservice);
NameComponent name = newNameComponent("Datum","");
NameComponent path[] = {name};
namingcontext.rebind(path,server);
```

Anforderung des initialen Namensraum

Name

Art

Helper: vom Schnitt-stellencompiler erzeugt

rebind: stellt das Serverobjekt der Middleware vor.

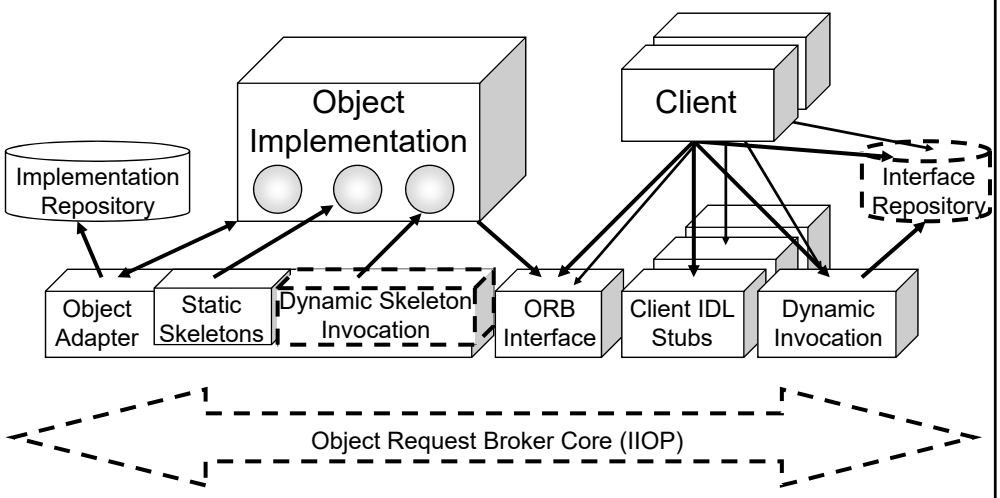
narrow: findet zu Objektreferenz die Klasse

Beispielcode: CORBA-Client

- Name Resolution (Client):

```
Server server;
org.omg.CORBA.Objectnameservice =
  orb.resolve_initial_references("NameService");
NamingContext namingcontext =
  NamingContextHelper.narrow(nameservice);
NameComponent name = newNameComponent("Datum","");
NameComponent path[] = {name};
server =
  ServerHelper.narrow(namingcontext.resolve(path));
```


Struktur eines CORBA-2.*-ORB's I

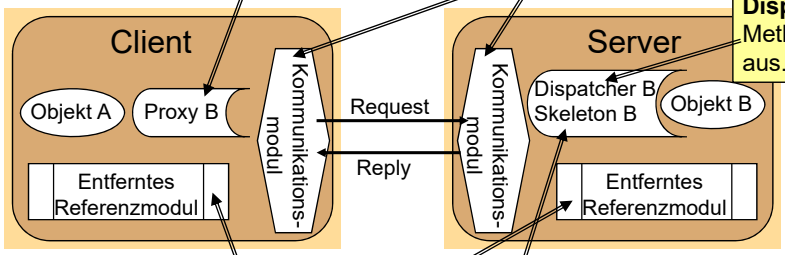


Rolle von Proxy und Skeleton

Proxy: macht RMI transparent für Client. Klasse implementiert entfernte Schnittstelle. Marshals Request und unmarshals Reply. Leitet Request weiter.

Ausführung des Request/Reply Protokolls

Dispatcher: wählt Methode im Skeleton aus.



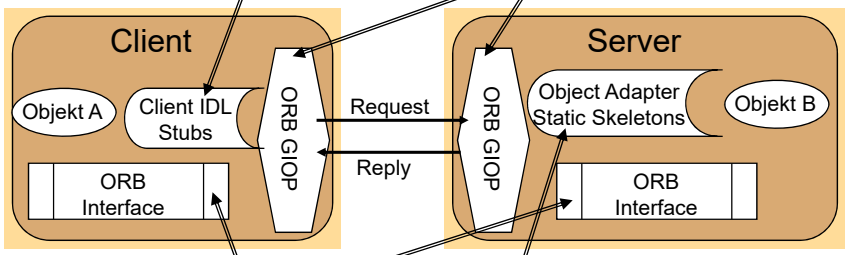
Übersetzung zwischen lokalen und entfernten Objektreferenzen

Skeleton: implementiert Methoden der entfernten Schnittstelle. Unmarshals Request und Marshals Reply. Ruft Methode in entferntem Objekt auf.

Rolle von Proxy und Skeleton

Erzeugt vom IDL-Sprachcompiler

Basis: RPC over TCP/IP



U.a. Zugang zur IOR

Erzeugt vom IDL-Sprachcompiler

Beispiel: CORBA-IOR

```
module ggt{
   interface unit{ … }; };
```

Inhalt der Schnittstellendefinition

Abgelegte Referenz im Namensdienst

```
IOR:010000001100000049444c3a6767742f756e69743a312e3000000000020000000000000003c000000010100001900000006c616232322
e6370742e6861772d68616d627572672e64650000b70e140000002f3131393031 2f313231333237363837322f5f3001000000240000000 1
0000000100000001000000140000000100000001000100000000000901010000000000
```

Inhalt der IOR

```
Repo Id:  IDL:ggt/unit:1.0
IIOP Profile
   Version:  1.0
   Address:  lab22.cpt.haw-hamburg.de:3767
      Key:  2f 31 31 39 30 31 2f 31 32 31 33 32 37 36 38 37 /11901/121327687
         32 2f 5f 30                                  2/_0
Multiple Components:
Codesets Tag
  Native char CS: ISO 8859-1:1987; Latin Alphabet No. 1
  Native wchar CS: ISO/IEC 10646-1:1993; UTF-16, UCS Transformation Format 16-bit form
```


Namen und Binden

- Zuordnung Name ---> Adresse
- Bindezeitpunkt:
  - beim Übersetzen (**statisches Binden**)
    (Call-by-Value)
    z.B. bei Programmiersprachen
  - beim Starten (**"halb"-dynamisches Binden**)
    z.B. moderne Binder (SunOS), nach dem Start in der Regel nicht änderbar
  - beim Zugriff (**dynamisches Binden**)
    (Call-by-Refernce/Call-by-Name)
    in verteilten Systemen angebracht:
    - Neue Dienste
    - Verlagerung existierender Dienste

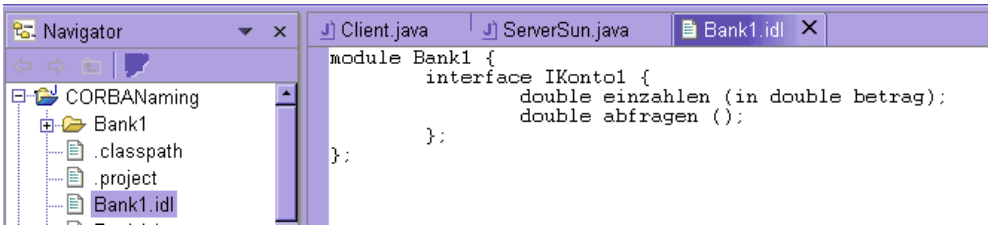
Schritt 1: Bank1.idl

Definiert den Namenskontext

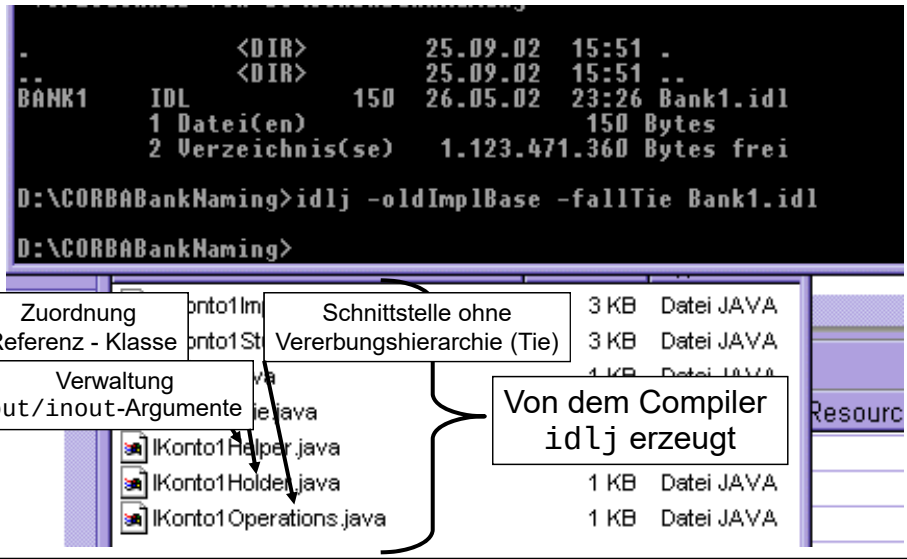
Definiert die CORBA-Klasse IKonto1

Definiert die Methoden `einzahlen` und `abfragen`

```
module Bank1 {
   interface IKonto1 {
      double einzahlen (in double betrag);
      double abfragen ();
   };
};
```



Schritt 2: IDL-Compiler



Zuordnung Referenz - Klasse

Schnittstelle ohne Vererbungshierarchie (Tie)

Verwaltung `out/inout`-Argumente

Von dem Compiler `idlj` erzeugt

IDL-Compiler

IDL-Datei	Zu generierender Java-Code

module math_ops {
  class Calculator {
  double add(double a, double b);
  string getStr(double a);
  };
};
	package math_ops;
	
	public abstract class _Calculator**ImplBase** ... {
	  public abstract double add(double a, double b);
	  public abstract String getStr(double a);
	  public static _Calculator**ImplBase narrowCast**(
	  Object rawObjectRef) { ... }
	  ...
	}
	
	<ggf. weitere hier benötigte Klassen/Interfaces>

accessor_one

```
public abstract class ClassOneImplBase {
   public abstract String methodOne(String param1, int param2)
                                    throws SomeException112;
   public static ClassOneImplBase narrowCast(Object rawObjectRef) {...}
}

public abstract class ClassTwoImplBase {
   public abstract int methodOne(double param1) throws SomeException110;
   public abstract double methodTwo() throws SomeException112;
   public static ClassTwoImplBase narrowCast(Object rawObjectRef) {...}
}

public class SomeException110 extends Exception {
   public SomeException110(String message) { super(message);}
}

public class SomeException112 extends Exception {
   public SomeException112(String message) { super(message);}
}
```

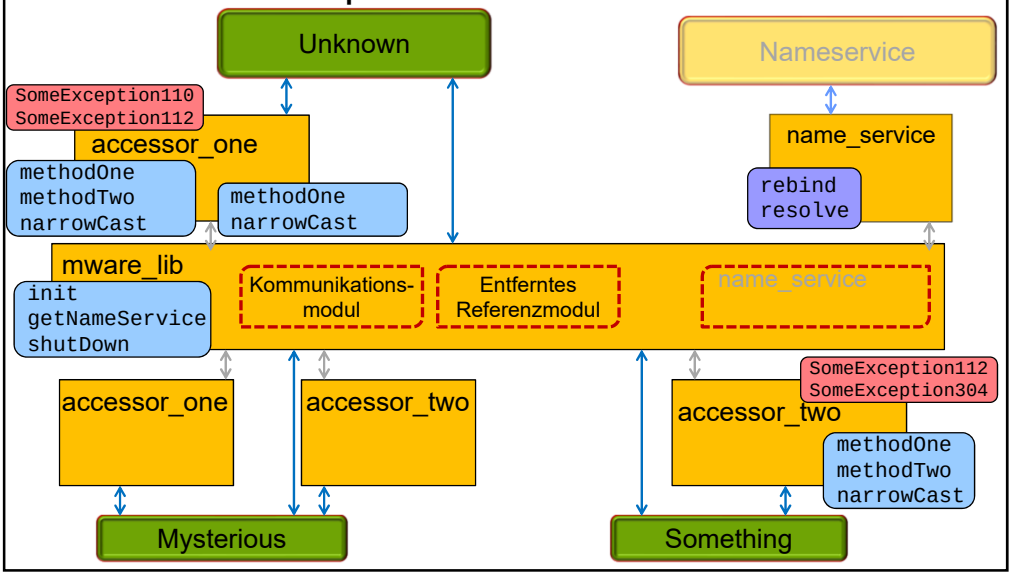
accessor_two

```
public abstract class ClassOneImplBase {
   public abstract double methodOne(String param1, double param2)
                                throws SomeException112;
   public abstract double methodTwo (String param1, double param2)
                                throws SomeException112, SomeException304;
   public static ClassOneImplBase narrowCast(Object rawObjectRef) {...}
}

public class SomeException112 extends Exception {
   public SomeException112(String message) { super(message);}
}

public class SomeException304 extends Exception {
   public SomeException304(String message) { super(message);}
}
```

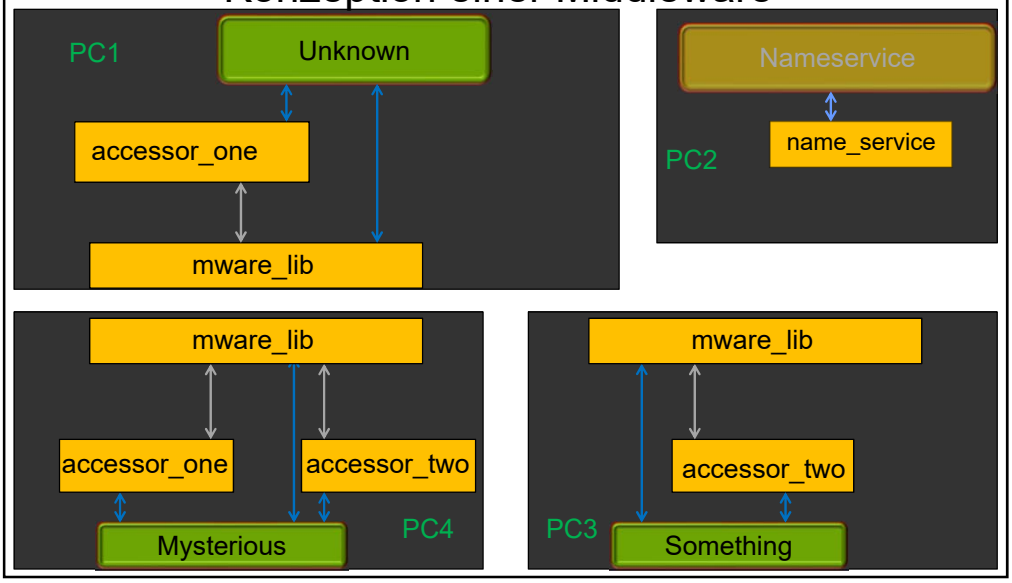

Sprachzuordnung

IDL-Typ	Entspricht in Java
`module` (keine Schachtelung, 1 Modul pro Datei)	`package`
`class` (nicht als Parameter oder Returnwert, keine Schachtelung)	`class`
`int`	`int`
`double`	`double`
`string`	`String`

Konzeption einer Middleware



Konzeption einer Middleware



Server-Anwendung

```
import math_ops.*
...

public class Calculator extends _CalculatorImplBase {
   public double add(double a, double b) { return a + b; }
   ...
}
...
   ObjectBroker objBroker = ObjectBroker.init(host, port, false);
   NameService nameSvc = objBroker.getNameService();
   nameSvc.rebind((Object) myObject, "zumsel"); …
...
   objBroker.shutDown();
...
```


Client-Anwendung

```
import math_ops.*;
...
ObjectBroker objBroker = ObjectBroker.init(host, port, false);
NameService nameSvc = objBroker.getNameService();
Object rawObjRef = nameSvc.resolve("zumsel");
       // generische Objektreferenz

_CalculatorImplBase remoteObj = _CalculatorImplBase.narrowCast(rawObjRef);
       // liefert klassenspezifisches Stellvertreterobjekt
...
try { // Entfernter Methodenaufruf
       double s = remoteObj.add(1, 567);
catch ( RuntimeException e ) { … }
...
objBroker.shutDown();
...
```

Hinweise und Tipps

- Überlegen Sie sich, welche Informationen für einen Aufruf ausgetauscht werden müssen und wie. Entwerfen Sie ein **geeignetes Request/ReplyProtokoll**.
- Sockets sollten nur in möglichst wenigen Klassen verwendet werden.
- Es kann vorkommen, dass zwei oder mehr Klienten ein und dieselbe Objektreferenz **zeitgleich nutzen** wollen. Die Erfüllung solcher Anforderungen soll innerhalb der Middleware nicht zu Deadlocks führen. (Die Behebung von Deadlocks in den Anwendungen ist hingegen nicht Aufgabe der Middleware.)
- An einigen Stellen ist ein Übergang zwischen der (statischen) Bibliothek *mware_lib* und den vom IDLCompiler generierten Code notwendig. Hier helfen die Vererbungsmechanismen weiter. Der Einsatz der JavaReflection ist nur mit entsprechender Erfahrung zu empfehlen.
- Den Namen einer Klasse können Sie in Java zur Laufzeit mit dem Stacktrace von Throwable ermitteln:
  `new Throwable().getStackTrace()[0].getClassName()`