Client/Server-Programmierung

WS 2019/2020

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 17. Januar 2020

Client/Server-Programmierung

WS 2019/2020

3 CORBA

Inhalt

- CORBA-Architektur
- CORBA-Dienste
- Beispielprogramm
- CORBA im Detail
  - IDL und IDL-Java-*Mapping*
  - Namensdienst
  - POA
  - GIOP, IIOP und IOR
  - *Implementation-* und *Interface-Repository*

Literatur

- CORBA-Spezifikationen der OMG
  - http://www.omg.org/technology/documents/corba_spec_catalog.htm
- Farley / Crawford / Flanagan, Kap. 14
- Orfali / Harkey, Kap. 1, 4, 7-9, 17-22
- Hofmann / Jobst / Schabenberger, Kap. 2.2, 5, 6
- Michi Henning, Steve Vinoski: Advanced CORBA Programming with C++. Addison-Wesley, 1999.

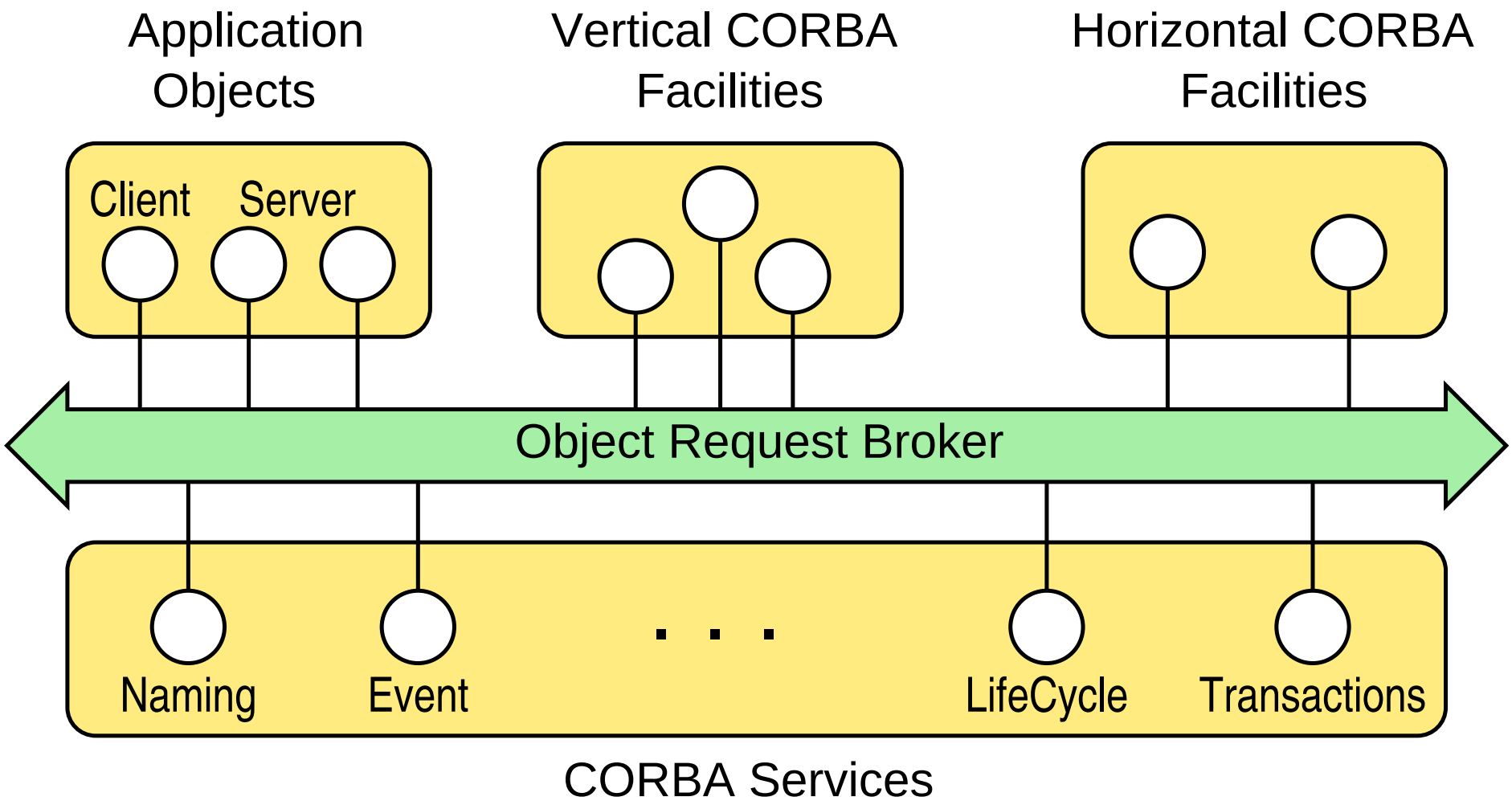
3.1 Einführung

- CORBA: *Common Object Request Broker Architecture*
- Ziel: Entwicklung und Integration verteilter objektorientierter Anwendungen in heterogenen Umgebungen
  - CORBA ist plattform- und sprachunabhängig
- Informeller Standard, definiert durch die OMG (*Object Management Group*)
  - 1989 gegründet, Ziel: Förderung objektorientierter Techniken
  - heute über 800 Mitglieder (größtes IT-Industriekonsortium)
- CORBA ist nur eine Spezifikation
  - verschiedenste Implementierungen, z.B. Orbix, ORBACUS, Java IDL (Teil des JDK), JacORB, ORBit (GNOME!) ...

3.2.1 *Object Management Architecture* (OMA)

- Definiert Objekt- und Referenzmodell
- Objektmodell
  - unterstützt Kapselung, (Mehrfach-)Vererbung, Polymorphie
  - Objekte bieten Dienste mit definierter Schnittstelle an
  - Client nutzt Dienste (lokal oder entfernt) über Schnittstelle, ist vollständig von Server-Implementierung unabhängig
  - Objektimplementierung mit beliebiger Programmiersprache
- Referenzmodell
  - Interaktion zw. Objekten und dazu notwendige Komponenten
  - Herzstück: *Object Request Broker* (ORB)
    - „Software-Bus“ für Kommunikation zw. Client und Server

Das OMA Referenzmodell



Bestandteile des OMA Referenzmodells

- *Object Request Broker* (ORB)
  - stellt Dienstanfragen an verteilte Objekte zu
  - realisiert Ortstransparenz für die Client-Objekte
- CORBA *Services* (*Object Services*)
  - domänenunabhängige (horizontale) Schnittstellen zu wichtigen Basisdiensten, z.B. Namensdienst
  - betriebssystem-ähnliche Funktion
- *Horizontal CORBA Facilities* (*Common Facilities*)
  - Schnittstellen zu anwendungsorientierten, domänenübergreifenden Diensten
  - z.B. Drucken, verteilte Dokumente, ...

Bestandteile des OMA Referenzmodells ...

- *Vertical CORBA Facilities* (*Domain Interfaces*)
  - Schnittstellen zu anwendungsorientierten Diensten für bestimmte Anwendungsdomänen
    - z.B. Finanzwesen, Medizin, Telekommunikation, ...
- *Application Objects*
  - anwendungsspezifische Schnittstellen
  - im Ggs. zu *Services* und *Facilities* nicht von der OMG standardisiert
- OMG spezifiziert nur die Schnittstellen, nicht die Implementierungen

3.2.2 *Common Object Request Broker Architecture* (CORBA)

- Zentrale Idee: transparente Kommunikation zwischen Client und Server-Objekten über ORB
  - ORB bietet Client eine lokale (Proxy-)Schnittstelle
- Vorteile beim Einsatz eines ORB:
  - Zugriffs- und Ortstransparenz
  - Transparenz der Implementierungssprache
  - Transparenz der Objektaktivierung
    - ORB übernimmt ggf. Aktivierung des Objekts
  - Transparenz der Kommunikationstechnik

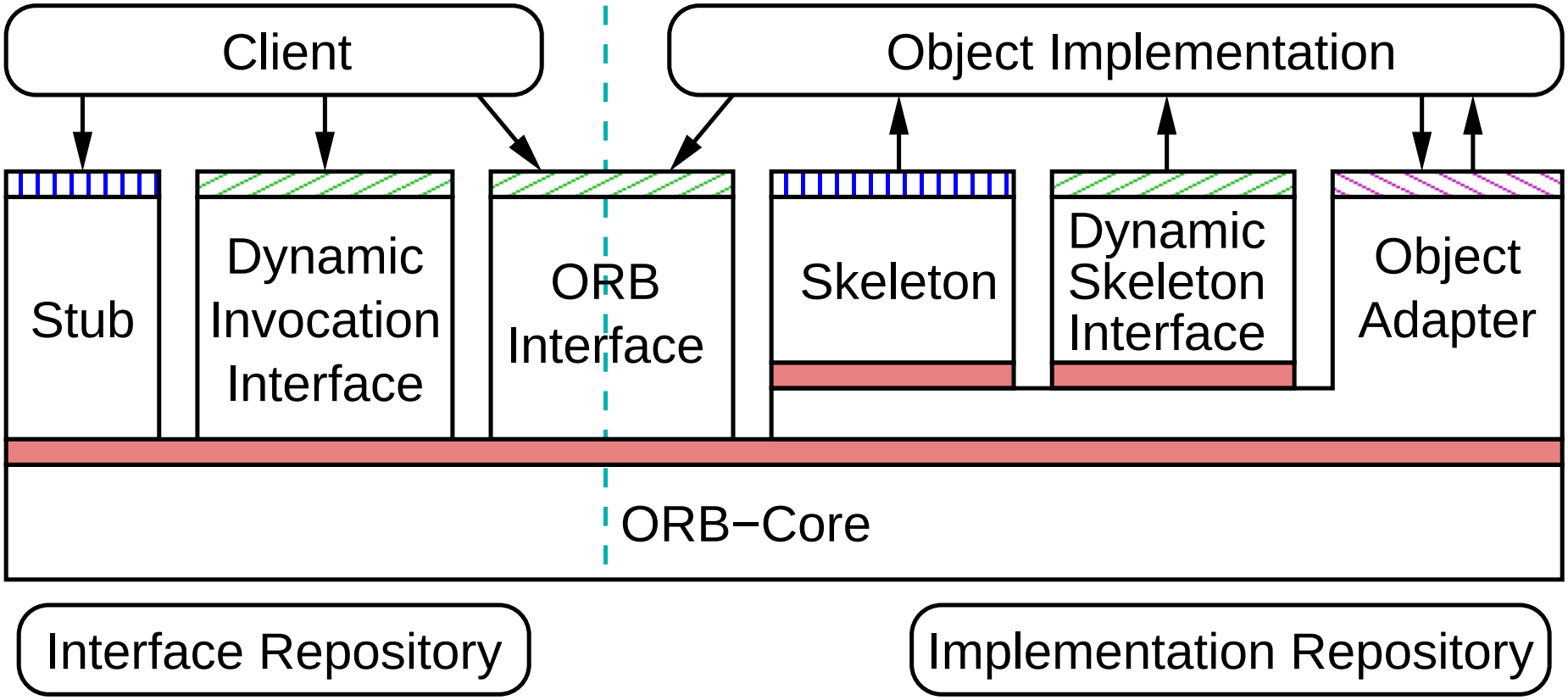
Objektreferenzen

- Zugriff auf Objekte erfolgt über Objektreferenzen
- Objektreferenzen
  - identifizieren genau ein Objekt
    - aber: verschiedene Referenzen für ein Objekt möglich
  - können `null` sein, d.h. auf kein Objekt zeigen
  - können „hängen“, wenn Objekt nicht mehr existiert
  - können persistent sein
  - sind typsicher
  - unterstützen spätes Binden (Polymorphismus)
  - sind interoperabel, d.h. Aufbau ist standardisiert
  - sind opak, d.h. Client darf Inhalt nicht betrachten

OMG Interface Definition Language (OMG IDL)

- OMG IDL erlaubt formale Beschreibung der Objekt-Schnitt-stellen
  - unabhängig von Implementierung der Objekte (z.B. Programmiersprache)
  - Objekte können z.B. durch C++- oder Java-Objekte, aber auch durch eigene Programme oder OO-Datenbanken implementiert werden
- OMG definiert, wie IDL in verschiedene Sprachen abgebildet wird (*Language Mapping*)
  - derzeit für C, C++, Java, Smalltalk, Ada, Lisp, Phyton, Cobol, PL/1, Ruby

Modell des *Object Request Brokers*



Client/Server-Programmierung

WS 2019/2020

25.10.2019

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 17. Januar 2020

Komponenten des ORB

- *ORB Core*
  - stellt Basisfunktionalität zur Verfügung
    - Objekt-Repräsentation, Kommunikationsmechanismen
  - ist i.d.R. verteilt implementiert
- *Stub* und *Skeleton*
  - für entfernten Methodenaufruf
  - vom IDL-Compiler aus Schnittstellendefinition erzeugt
- *Dynamic Invocation Interface* und *Dynamic Skeleton Interface*
  - erlauben dynamische Methodenaufrufe/-implementierungen
  - d.h. Schnittstelle muß zur Übersetzungszeit des Clients bzw. Servers noch nicht festgelegt sein

Komponenten des ORB ...

- *Object Adapter*
  - stellt Objektimplementierung Dienste des ORB zur Verfügung
  - Funktionen u.a.:
    - Methodenaufrufe (über Skeletons)
    - Abbildung von Referenzen auf Impementierungen
    - Registrierung von Implementierungen
    - Generierung und Interpretation von Objektreferenzen
    - Objektaktivierung und -deaktivierung
  - unterschiedliche Objektadapter möglich
  - ab CORBA 2.2: *Portable Object Adapter* (POA) als Standard-Schnittstelle zum Objektadapter

Komponenten des ORB ...

- *ORB Interface*
  - Schnittstelle für Dienste, die der ORB dem Client und allen Objektimplementierungen zur Verfügung stellt
- *Interface Repository*
  - Dienst, der zur Laufzeit persistente Information zu den registrierten Objektschnittstellen zur Verfügung stellt
  - Nutzung ggf. durch ORB und / oder Clients
  - allgemein: jegliche Information zu Objekt-Schnittstellen
- *Implementation Repository*
  - enthält Information, die es dem ORB erlaubt, Objekte zu lokalisieren und zu aktivieren
  - allgemein: jegliche Information zu Objekt-Implementierungen

3.2.3 CORBA-Dienste

- Von der OMG spezifiziert, erweitern ORB-Funktionalität
  - realisiert durch CORBA-Objekte mit IDL-Schnittstelle
  - Anbieter von ORB und Diensten können unterschiedlich sein
- *Collection Service*
  - *Container*-Objekte, z.B. *Map*, *Set*, *Queue*
- *Concurrency Control Service*
  - realisiert Sperren für wechselseitigen Ausschluß
- *Event Service*
  - verteilt Ereignisse an interessierte Objekte
- *Externalization Service*
  - (De-)Serialisierung von Objekten

3.2.3 CORBA-Dienste ...

- *Licensing Service*
  - erfaßt Nutzung von Objekten für Abrechnung
- *Life Cycle Service*
  - Erzeugen, Löschen, Kopieren und Verschieben von Objekten
- *Naming Service*
  - Zuordnung von Namen zu Objektreferenzen
- *Notification Service*
  - *Event Service*-Erweiterung: beliebige Daten als Ereignisse
- *Persistent Object Service*
  - persistentes Speichern von Objekten in Datenbanken
- *Property Service*
  - Verwaltet Name/Wert-Paare

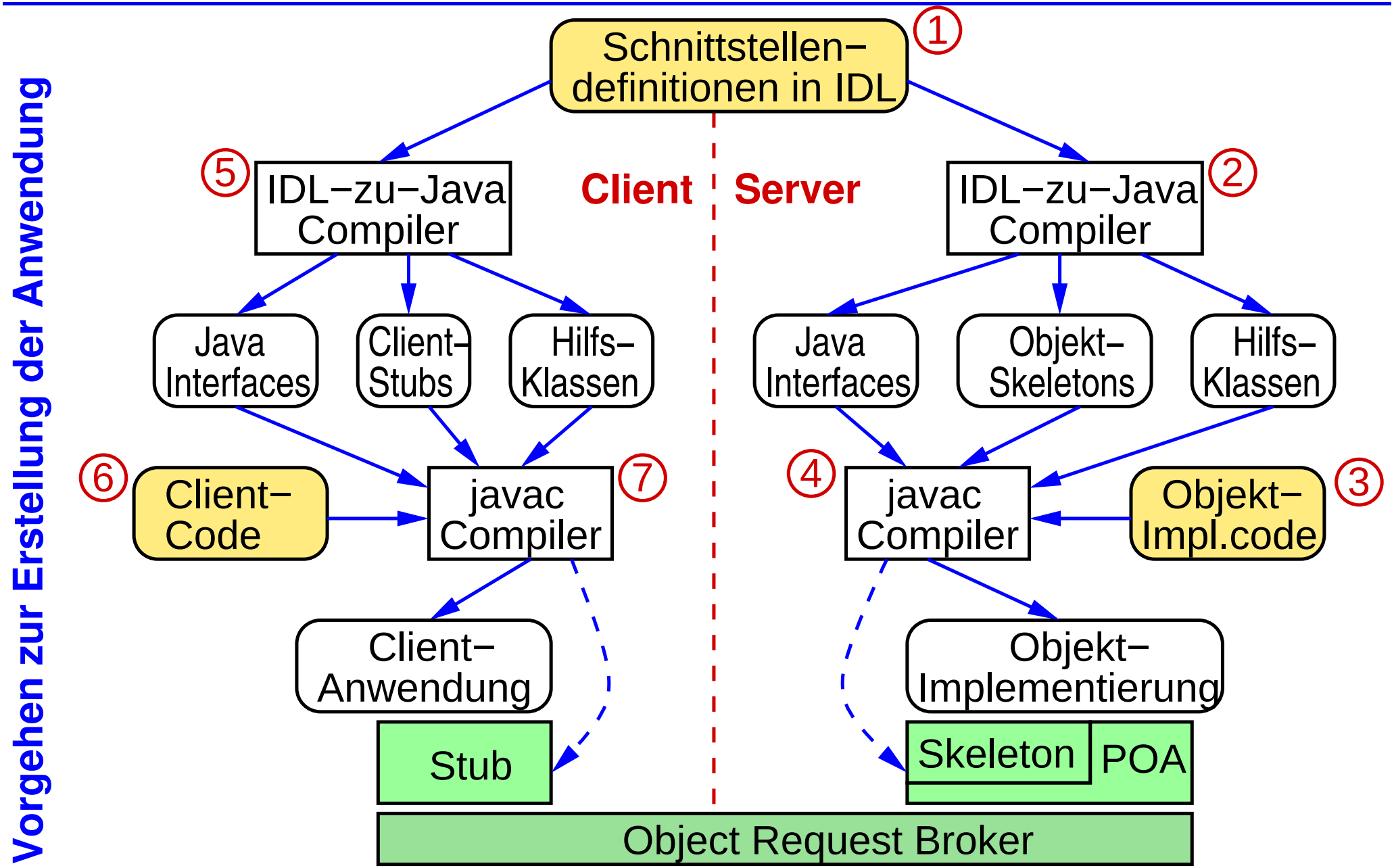
3.2.3 CORBA-Dienste ...

- *Query Service*
  - Anfrageoperationen an verteilte Objekte (SQL-Obermenge)
- *Relationship Service*
  - Erzeugung / Traversierung von Assoziationen zw. Objekten
- *Security Service*
  - Authentifizierung, Zugriffskontroll-Listen, Rechteweitergabe
- *Time Service*
  - Zeit-Synchronisation
- *Trading Service*
  - erlaubt CORBA-Objekte anhand ihrer Fähigkeiten zu finden
- *Object Transaction Service*
  - flache und verschachtelte verteilte Transaktionen

Vorbemerkungen

- In der Vorlesung und Übung:
  Verwendung zweier CORBA-Implementierungen
  - Java IDL
    - Seit JDK 1.2 fester Bestandteil der Java Entwicklungsumgebung
    - wenig Dienste (nur Namensdienst)
  - JacORB 2.3.1
    - frei verfügbarer ORB, in Java implementiert
    - mehr Dienste (u.a. *Naming*, *Event*, *Notification*, *Transaction*, *Trading*), sowie *Interface-* und *Implementation Repository*
- Wegen POA: Anwendungs-Quellcode ist für alle Implementierungen gleich

3.3 *Hello World* mit CORBA ...



IDL-Beschreibung der Schnittstelle

```
module HelloWorld
{
  interface Hello
  {
    string sayHello(in string name);
  };
};
```

Paketname

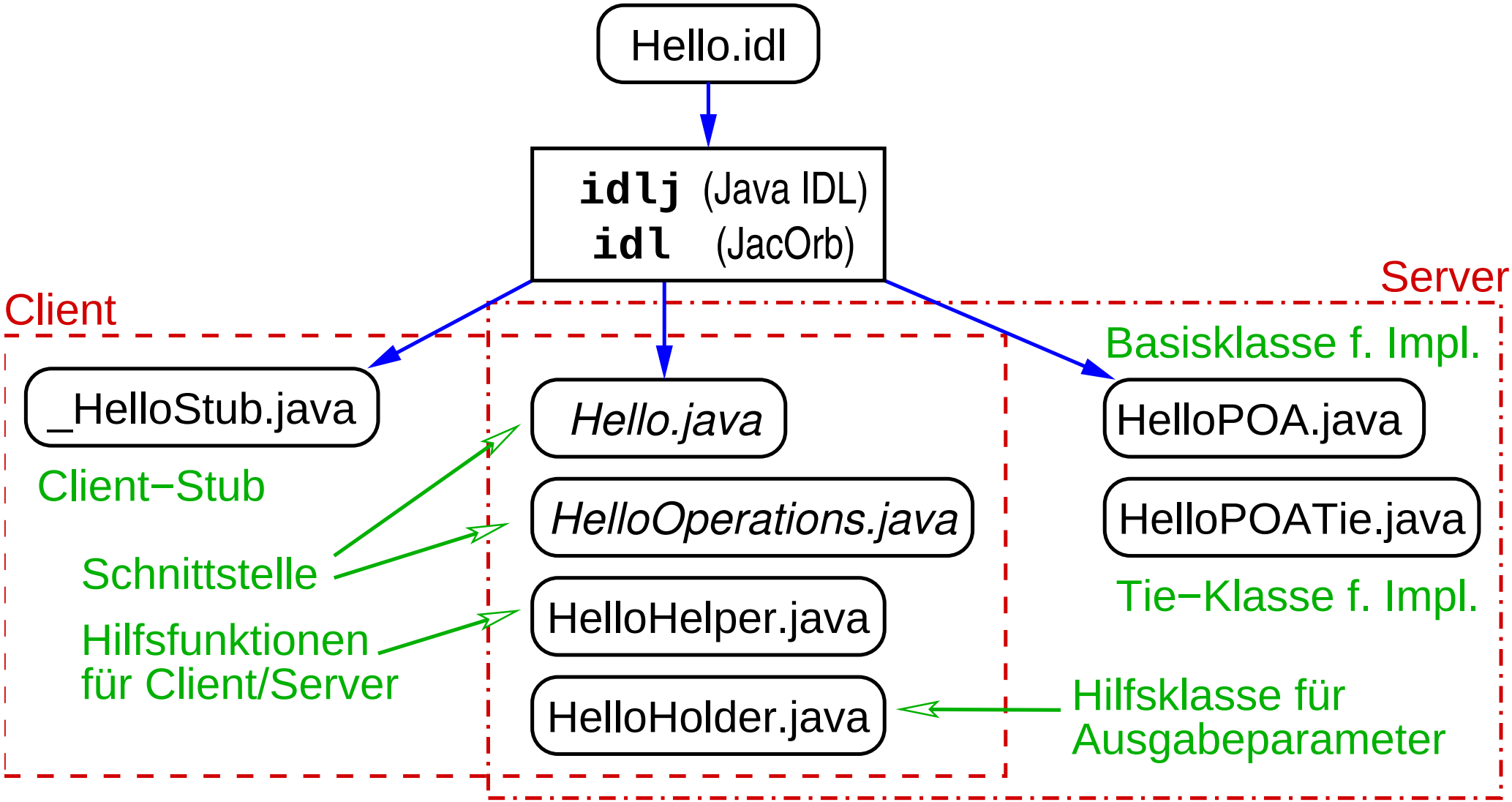
Schnittstelle

Methode

Eingabeparameter

- CORBA definiert eigene, C++/Java-ähnliche Sprache
- IDL-Compiler erzeugt Schnittstellen für jeweilige Implementierungssprache
  - kann für Client und Server verschieden sein

Vom IDL-Compiler generierte Dateien



Aufruf des IDL-Compilers

- Java IDL:
  - `idlj -fall` – erzeugt alle Dateien
  - `idlj -fclient` – erzeugt Dateien für Client
  - `idlj -fserver` – erzeugt Dateien für Server
    - ohne Helper-Klassen!
  - `idlj -fallTie` – erzeugt alle Dateien incl. Tie-Klassen
- JacORB:
  - `idl` – erzeugt alle Dateien incl. Tie-Klassen
  - `idl -noskel` – erzeugt Dateien nur für Client
  - `idl -nostub` – erzeugt Dateien nur für Server
- Weiteres Argument: Name der IDL-Datei

Vom IDL-Compiler erzeugte Java-Schnittstelle

- HelloWorld/Hello.java:

```
package HelloWorld;
public interface Hello extends HelloOperations,
  org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity
{
}
```

- HelloWorld/HelloOperations.java:

```
package HelloWorld;
public interface HelloOperations
{
  String sayHello (String name);
}
```

Objektimplementierung

```
// vom IDL—Compiler erzeuges Paket mit Hilfsklassen
import HelloWorld.*;

public class HelloImpl extends HelloPOA
{
    public String sayHello(String name)
    {
        return "The world says HELLO to " + name;
    }
}
```

Server-Programm

```
// vom IDL—Compiler erzeuges Paket mit Hilfsklassen
import HelloWorld.*;

// CORBA Namensdienst
import org.omg.CosNaming.*;

// Für Exceptions, die Namensdienst werfen kann
import org.omg.CosNaming.NamingContextPackage.*;

// Alle CORBA—Server benötigen diese Klassen
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
```

3.3 *Hello World* mit CORBA ...

```
public class HelloServer {
  public static void main(String args[]) {
    try {
      // Erzeuge und Initialisiere den ORB
      ORB orb = ORB.init(args, null);

      // Erzeuge das Servant−Objekt
      HelloImpl helloRef = new HelloImpl();

      // Aktivierung des POA
      POA rootpoa = POAHelper.narrow(
          orb.resolve_initial_references("RootPOA"));
      rootpoa.the_POAManager().activate();
```

```
// Registriere Servant und erzeuge (CORBA) Objektreferenz
org.omg.CORBA.Object ref =
    rootpoa.servant_to_reference(helloRef);
// Typ−Konvertierung
Hello href = HelloHelper.narrow(ref);

// Registriere Objektreferenz beim Namensdienst
org.omg.CORBA.Object objRef =
    orb.resolve_initial_references("NameService");
NamingContextExt ncRef =
    NamingContextExtHelper.narrow(objRef);
NameComponent path[] = ncRef.to_name("HelloWorld");
ncRef.rebind(path, href);

System.out.println("HelloServer is running...");
```

```
        // Starte ORB
        orb.run();

    } catch(Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
    }
  }
}
```

Client-Programm

```
import HelloWorld.*;              // Client Stubs
import org.omg.CosNaming.*;       // Namensdienst
import org.omg.CORBA.*;           // CORBA Klassen

public class HelloClient {
  public static void main(String args[]) {
    try {
      // Erzeuge und Initialisiere den ORB
      ORB orb = ORB.init(args, null);

      // Aufsuchen des Objekts beim Namensdienst
      org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
```

```
        NamingContextExt ncRef =
            NamingContextExtHelper.narrow(objRef);
        NameComponent path[] = ncRef.to_name("HelloWorld");
        Hello helloRef =
            HelloHelper.narrow(ncRef.resolve(path));

        // Aufruf der Methode des CORBA-Objekts
        System.out.println(helloRef.sayHello("Peter"));

    } catch(Exception e) {
        System.out.println("ERROR : " + e);
        e.printStackTrace(System.out);
    }
  }
}
```

Anmerkungen zum Code

- `resolve_initial_references()` dient der initialen Auflösung von Namen, insbesondere:
  - `RootPOA`: der Wurzel-POA des Servers (☞ **3.4.6**)
  - `NameService`: der Namensdienst

  Ergebnis ist `org.omg.CORBA.Object`
  - CORBA Objektreferenz
- `narrow()` dient zum Umwandeln einer Referenz vom Typ `org.omg.CORBA.Object` in einen konkreten Typ (*downcast*)
- `to_name()` wandelt String in strukturierten Namen für Namensdienst um (s. später)

Starten der Anwendung (Java IDL)

- Starten des ORB-Daemons (Namensdienst)
  - `orbd -ORBInitialPort 12345 [ -port` <port>]
  - startet Daemon auf Port 12345 (`-port` legt Port für Objektaktivierung fest)
- Starten des Servers
  - `java HelloServer -ORBInitialPort 12345 [ -ORBInitialHost` <addr>]
  - Angabe von Host und Port des ORB-Daemons
- Starten des Clients
  - `java HelloClient -ORBInitialPort 12345 [ -ORBInitialHost` <addr>]

Zur Nutzung von JacORB im Labor H-A 4111

- Umgebungsvariablen setzen (ggf. in `$HOME/.profile`):
  - `export JACORB_HOME=/opt/dist/JacORB`
  - `export PATH=$JACORB_HOME/bin:$PATH`
- Konfigurationsdatei einrichten:
  - `cp $JACORB_HOME/etc/jacorb_properties.template $HOME/orb.properties`
  - in `$HOME/orb.properties` alle `cspXXX` durch eigenes Login ersetzen
    - JacORB verwendet bei uns eine Datei unter `$HOME` zum Auflösen der initialen Referenzen
      - setzt Netzwerk-Dateisystem (NFS) voraus
    - auch möglich: Nutzung eines WWW-Servers

Starten der Anwendung (JacORB)

- Starten des Namensdienstes
  - `ns`
  - startet auf beliebigem freiem Port
  - schreibt eigene Objektreferenz in Datei `$HOME/.jaco_ns`
- Starten des Servers
  - `jaco HelloServer`
  - `jaco` ist ein Hilfsskript von JacORB
    - startet JVM mit nötigen *Properties* und *Classpath*
- Starten des Clients
  - `jaco HelloClient`

3.4.1 OMG IDL und Java-*Mapping*

- Struktur einer IDL-Datei:

```
module Identifikator {
    Typ-Deklarationen;
    Konstanten-Deklarationen;
    Exception-Deklarationen;
    interface Identifikator [ : Vererbung ] {
        Typ-Deklarationen;
        Konstanten-Deklarationen;
        Exception-Deklarationen;
        Attribut-Deklarationen;
        Methoden-Deklarationen;
    };
    ...
};
```

`module` und `interface`

- `module` definiert einen neuen Namensraum
  - ähnlich wie C++ *Namespaces* bzw. Java *Packages*
  - verschachtelte Namensräume möglich
- `interface` definiert neue Schnittstelle
  - ähnlich wie Java-Schnittstellen
  - Schnittstelle kann Methoden und Attribute enthalten
    - Attribute werden über automatisch generierte Deklarationen von *Get*- und *Set*-Methoden realisiert
  - Vererbung ist möglich, auch Mehrfachvererbung
    - vererbt werden nur Schnittstellen, keine Implementierungen

Methoden-Deklarationen

- Syntax:
  - [`oneway`] *Typ* *Identifikator* (*Parameterliste*)
    [`raises` (*Exceptions*)] [*Kontext*]
  - *Parameterliste*: Liste von Parameterdeklarationen:
    - { `in` | `out` | `inout` } *Typ* *Identifikator*
    - Klassifikation nach Ein- und Ausgabeparametern
  - *Kontext*: Liste von Kontextvariablen
    - ähnlich UNIX Umgebungsvariablen, werden an Server übergeben
  - `oneway`: asynchroner Methodenaufruf
    - ohne Ergebnis / Ausgabeparameter
- Überladen von Methoden ist nicht erlaubt

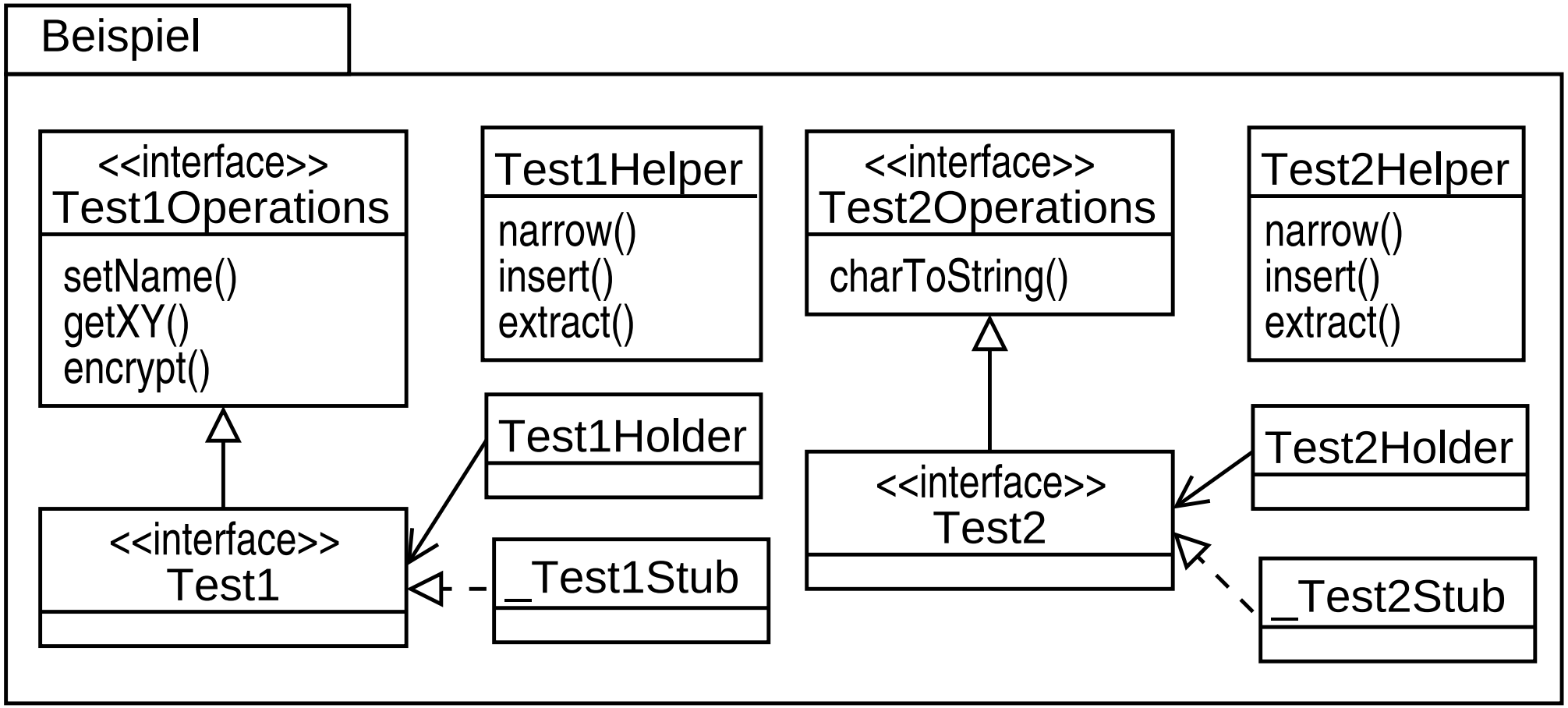
Wichtige Basis-Datentypen und ihre Java-Entsprechungen

IDL Datentyp	Beschreibung	Java Datentyp
[unsigned] short	Ganzzahl, 16 Bit	short
[unsigned] long	Ganzzahl, 32 Bit	int
[unsigned] long long	Ganzzahl, 64 Bit	long
float	Gleitkomma, 32 Bit	float
double	Gleitkomma, 64 Bit	double
char	Zeichen, 8 Bit	char
string	Zeichenkette	String
boolean (TRUE, FALSE)	Boole'scher Wert	boolean (true, false)
octet	Byte	byte
any	beliebiger Typ	org.omg.CORBA.Any

Beispiel

```
module Beispiel {
  interface Test1 {
    void setName(in string name);
    double getXY(out double y);
    long encrypt(in string key, inout string msg);
  };
  interface Test2 {
    string charToString(in char c1, in char c2);
  };
};
```

Vom IDL-Compiler erzeugte Klassen und Interfaces



Erzeugte Java-Schnittstellen

- Beispiel/Test1Operations.java:

```
package Beispiel;
public interface Test1Operations {
  void setName (String name);
  double getXY (org.omg.CORBA.DoubleHolder y);
  int encrypt (String key,
               org.omg.CORBA.StringHolder msg);
}
```

- Beispiel/Test2Operations.java:

```
package Beispiel;
public interface Test2Operations {
  String charToString (char c1, char c2);
}
```

Holder-Klassen

- Java unterstützt keine Ausgabeparameter
  - daher: Übergabe eines Objekts (per Referenz!), das den Parameter enthält (*Holder*-Klasse)
- Z.B. Code für `DoubleHolder`:

```
public final class DoubleHolder ... {
    public double value;
    public DoubleHolder() {
    }
    public DoubleHolder(double initialValue)
        value = initialValue;
    }
    ...
}
```

Helper-Klassen

- Hilfsmethoden für CORBA-Objekte
- Ausschließlich `static`-Methoden:
  - `narrow()`: Typumwandlung (*Down cast*), wandelt generische CORBA Objektreferenz in konkrete Java Objektreferenz um
  - `insert()`: packt Objekt in Datentyp `org.omg.CORBA.Any`
  - `extract()`: extrahiert Objekt aus `org.omg.CORBA.Any`
  - `id()`: Schnittstellen-ID für *Interface-Repository*
  - `read()`: Lesen eines Objekts aus einem Eingabestrom
  - `write()`: Schreiben eines Objekts in einen Ausgabestrom

Abgeleitete Datentypen

- OMG IDL erlaubt Definition abgeleiteter Datentypen
  - Interfaces (Objekte)
  - Sequenzen und Arrays
  - Strukturen (*Struct*)
  - Aufzählungen (*Enum*)
  - Vereinigungen (*Union*)
- Daneben: Typdefinitionen (*Typedefs*)
- Syntax / Semantik stark an C/C++ angelehnt

Typdefinitionen (*Typedefs*)

- IDL erlaubt es, neue Namen für existierende Typen zu definieren
  - auf allen Ebenen (global, Modul, Schnittstelle)
- Beispiele:

```
typedef short Temperatur; // neuer Typname: Temperatur
typedef Test2 MyTest;
```

- CORBA legt nicht fest, ob neuer Typname nur Alias für existierenden Typ ist, oder ob neuer Typ erzeugt wird

Sequenzen

- Vektoren (eindimensionale Felder) variabler Länge
  - beliebige IDL-Elementtypen
  - Länge kann begrenzt oder unbegrenzt sein
- Beispiel: IDL

```
typedef sequence<Dog> MySeq;
typedef sequence<Dog, 60> MyBoundedSeq;
void seqtest(in MySeq par1, in MyBoundedSeq par2,
             out MySeq par3);
```

- Umsetzung in Java:

```
void seqtest(Beispiel.Dog[] par1, Beispiel.Dog[] par2,
             Beispiel.Test2Package.MySeqHolder par3);
```

Arrays

- Ein- oder mehrdimensionale Felder fester Länge
- Beispiel: IDL

```
typedef Dog MyArray[60];
typedef Dog My2DArray[60][10];
void arraytest(in MyArray par1, in My2DArray par2,
               out MyArray par3);
```

- Umsetzung in Java:

```
void arraytest(Beispiel.Dog[] par1,
               Beispiel.Dog[][] par2,
               Beispiel.Test2Package.MyArrayHolder par3);
```

Strukturen (*Structs*)

- Zusammenfassung von mehrerer Variablen
  - entspricht Klasse mit *public*-Attributen ohne Methoden
- Beispiel: IDL

```
struct MyStruct {
  short age;
  string name;
};
```

- Umsetzung in Java:

```
public final class MyStruct {
  public short age = (short)0;
  public String name = null;
  public MyStruct() {}
  public MyStruct(short _age,
                  String _name) {
    age = _age; name = _name;
  }
}
```

Client/Server-Programmierung

WS 2019/2020

28.10.2019

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 17. Januar 2020

Aufzählungen (*Enums*)

- Neuer Typ mit explizit aufgezählten Werten
- Beispiel: IDL

```
enum Color { red, green, blue };
```

- Umsetzung in Java:
  - Klasse `Color` mit
    - `static final`-Attributen für die Werte
      - als `int`-Wert (z.B. `_red`) und als `Color`-Objekt (z.B. `red`)
    - Methode `value()`, liefert `int`-Wert zurück
    - `static`-Methode `from_int()` als *Factory*
  - Verwendung z.B.:

```
Color mycol = Color.red;
```

Vereinigungen (*Unions*)

- Beschreibt Daten, die verschiedene Typen haben können
  - Typ wird durch Diskriminator zur Laufzeit festgelegt
- Beispiel: IDL

```
union Animal switch (short) {
   case 1:  Dog dog;
   case 2:  Cat cat;
   default: Mouse mouse;
};
```

- Umsetzung in Java: Klasse `Animal` mit
  - Methode `discriminator()` für Diskriminator
  - *Get-* und *Set*-Methoden für die Felder des *Union*
    - prüfen bzw. setzen auch Diskriminator-Wert

Attribute

- In einem Interface können auch Attribute definiert werden:
  - `attribute string name;`
    `readonly attribute long age;`
  - deklariert werden damit letztendlich nur Zugriffsmethoden zum Lesen und ggf. auch zum Schreiben
- Erzeugte Java-Schnittstelle im Beispiel:

```
String name ();
void name (String newName);
int age ();
```

Konstanten

- Konstanten können auf globaler, auf Modul- oder auf Interface-Ebene definiert werden
  - unterschiedlicher Gültigkeitsbereich
- Beispiel zur Syntax:

```
interface Test2 {
   const short maxItems = 40;
   const string myName = "Roland";
   ...
```

- Abbildung nach Java:

```
public interface Test2 extends Test2Operations ... {
   public static final short maxItems = (short)(40);
   public static final String myName = "Roland";
}
```

Exceptions

- IDL erlaubt Definition eigener Exceptions
  - daneben: jede Methode kann (implizit) eine Reihe von System-Exceptions werfen, z.B. *Marshaling*-Fehler
- Beispiel:

```
exception SyntaxError {
  unsigned short position;
};
void parse(in string command) raises (SyntaxError);
```

- Umsetzung in Java:

```
void parse (String command)
     throws Beispiel.Test2Package.SyntaxError;
```

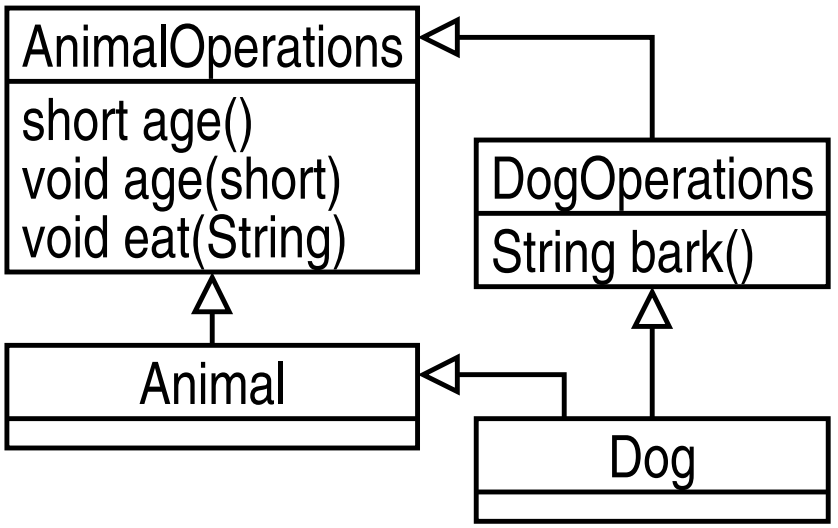
  - plus Exception-Klasse `SyntaxError`, analog zu *Struct*

Vererbung

- Methoden und Attribute von Schnittstellen können vererbt werden
  - auch Mehrfachvererbung möglich
  - kein Überladen / Überschreiben von Methoden erlaubt
- Beispiel:

```
interface Animal {
    attribute short age;
    void eat(in string what);
};
interface Dog : Animal {
    string bark();
};
```

- Umsetzung in Java:



Vererbung ...

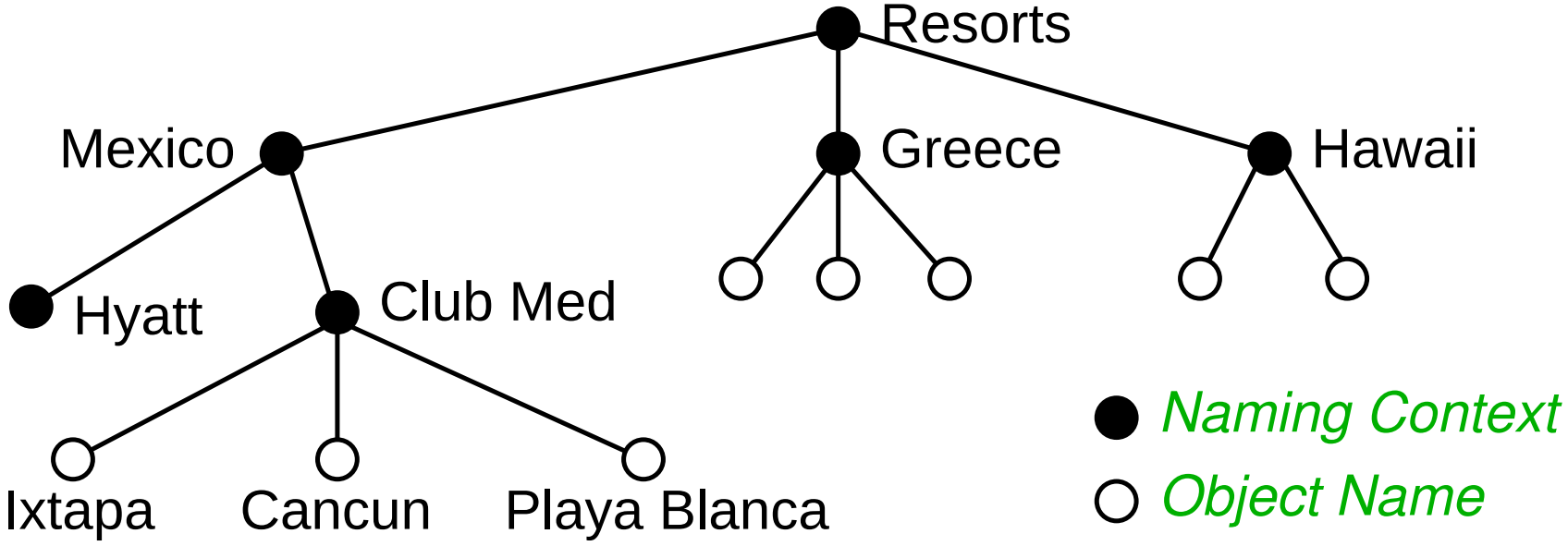
- Methoden der Basisklasse `org.omg.CORBA.Object` (Java *Bindings*):
  - `_is_a()`: implementiert Objekt gegebenes Interface?
  - `_non_existent()`: zugehöriges Server-Objekt zerstört?
  - `_is_equivalent()`: verweisen zwei Referenzen auf dasselbe Objekt? (soweit ORB das einfach bestimmen kann)
  - `_hash()`: Hashwert der Objektreferenz
  - `_duplicate()` / `_release()`: Kopie bzw. Freigabe einer Objektreferenz (i.a. keine Garbage Collection im ORB)
  - `_get_interface_def()`: liefert Schnittstellenbeschreibung aus *Interface Repository*
  - `_create_request()`: erzeugt *Request* für DII

Parameterübergabe-Semantik

- Alle Parametertypen außer Interfaces: *call-by-value*
  - Methoden dürfen `in`-Parameter jedoch nicht verändern
    - Verhalten ist sonst undefiniert
  - IDL unterstützt spezielle Typdeklarationen (`valuetype`), um Objekte zu definieren, die per Wert übergeben werden
    - analog zu `interface`, aber mit Angabe echter Attribute (*State Variables*)
- Interfaces: *call-by-reference*
  - übergeben wird die Objektreferenz (kein `narrow()` nötig)
- IDL unterstützt auch abstrakte Interfaces
  - können von `valuetype` und `interface` geerbt werden
  - damit: Übergabesemantik erst zur Laufzeit festgelegt

3.4.2 *Name Service*

- Der CORBA Namensdienst definiert einen hierarchischen Namensraum



- Aufbau eines Namens:

Resorts	Mexico	Club Med	Cancun
Context Name	*Context Name*	*Context Name*	*Simple Name*

3.4.2 *Name Service ...*

- Darstellung von Namen (in IDL):

```
struct NameComponent {
   string id;     // Eigentlicher Name
   string kind;   // Typ (analog zu Dateiendungen)
};
typedef sequence<NameComponent> Name;
```

- Wichtige Methoden des Namensdienstes (`NamingContext`)
  - `bind()` / `rebind()` : Binden von Objektnamen
  - `bind_context()` / `rebind_context()` : Binden eines *Naming Context* unter einem (Kontext-)Namen
  - `new_context()` : Erzeugen eines *Naming Context*
  - `unbind()` : Namen entfernen
  - `list()` : Ausgabe aller Namen
  - `resolve()` : Namen in Objekt auflösen

3.4.2 *Name Service ...*

- Erweiterte Schnittstelle `NamingContextExt`
  - von `NamingContext` abgeleitet
  - erlaubt u.a. Verwendung von Pfadnamen in Stringform
    - z.B.: **`a/.c /d.e`**

      `id`
      `kind`
      `NameComponent`
  - wichtige Methoden:
    - `to_name()` : wandelt Namen in Stringform in eine Sequenz von `NameComponents` um
    - `resolve_str()` : entspricht `resolve(to_name("name"))`
- (Java-)Referenz auf Wurzel-*Naming Context*:

```
NamingContextExt nc = NamingContextExtHelper.narrow(
        orb.resolve_initial_references("NameService"))
```

Beispiel

- Registrierung eines Objekts in einem neuen Kontext:

```
NamingContextExt nc = NamingContextExtHelper.narrow(
        orb.resolve_initial_references("NameService");

// Erzeugen und Binden des Naming Contexts
nc.rebind_context(nc.to_name("Hello"),
                  nc.new_context());

// Binden der Objektreferenz an hierarchischen Namen
nc.rebind(nc.to_name("Hello/HelloWorld"), objRef);
```

- Auflösen des Namens (ohne `NamingContextExt`):

```
NameComponent path[] = { new NameComponent("Hello",""),
                         new NameComponent("HelloWorld", "") };
Hello obj = HelloHelper.narrow(nc.resolve(path));
```

3.4.3 *Factories*

- Eine *Factory* kann auch eine Referenz auf ein neues CORBA-Objekt zurückliefern, das durch einen eigenen Servant implementiert wird
- Die Factory-Methode muß dazu:
  - einen neuen Servant erzeugen
  - diesen mit `servant_to_reference()` beim POA registrieren
  - die CORBA-Referenz mit `narrow()` in eine Java-Referenz umwandeln und als Ergebnis zurückliefern
- Eine Referenz auf den POA kann man u.a. über die von der Klasse `org.omg.PortableServer.Servant` geerbten Methoden `_default_POA()` oder `_poa()` erhalten
- Beispiel: siehe WWW

Beispiel: IDL-Datei

```
module HelloWorld
{
   // Schnittstelle der Objekte, werden vom Client über Factory erzeugt
   interface Hello
   {
    string sayHello();
   };
   // Schnittstelle des Factory−Objekts
   interface HelloFactory
   {
     Hello createHello(in string name);
   };
};
```

Beispiel: Client (Ausschnitt)

```
// Resolve the object reference in naming
NameComponent path[] = ncRef.to_name("HelloWorld");
HelloFactory fact = HelloFactoryHelper.narrow(
    ncRef.resolve(path));

// Erzeuge Objekte über Factory
Hello helloRef = fact.createHello("Klaus");
Hello helloRef2 = fact.createHello("Uwe");

// Rufe Methode für jedes Objekt auf
System.out.println(helloRef.sayHello());
System.out.println(helloRef2.sayHello());
```

Beispiel: Factory-Implementierung im Server

```
class HelloFactoryServant extends HelloFactoryPOA {
  public HelloFactoryServant() {
  }
  public Hello createHello(String name) {
    try {
      HelloServant helloRef = new HelloServant(name);
      org.omg.CORBA.Object ref =
          _poa().servant_to_reference(helloRef);
      return HelloHelper.narrow(ref);
    } catch(...) { ... }
  }
}
```

3.4.4 Delegation und *Tie*-Klassen

- In der Regel: Objekt-Implementierung erweitert die vom IDL-Compiler erzeugte POA-Klasse, z.B.:

```
public class HelloImpl extends HelloPOA {
    public String sayHello(String Name) {
    ...
```

- Falls dies nicht möglich ist: IDL-Compiler kann *Tie*-Klasse erzeugen, die Aufrufe delegiert
  - Objekt-Implementierung muß nur noch die Schnittstelle implementieren, z.B.:

```
public class HelloImpl implements HelloOperations {
    public String sayHello(String Name) {
    ...
```

Aufbau der *Tie*-Klasse

```
public class HelloPOATie extends HelloPOA
{
  private HelloOperations _delegate;
  public HelloPOATie(HelloOperations delegate) {
    _delegate = delegate;
  }
  // _this() registriert den Servant auch beim RootPOA
  public HelloWorld.Hello _this() {
    return HelloWorld.HelloHelper.narrow(_this_object());
  }
  ...
  public java.lang.String sayHello(java.lang.String Name) {
    return _delegate.sayHello(Name);
  }
}
```

Server-Programm mit *Tie*-Klasse

```
ORB orb = ORB.init(args, null);

HelloImpl helloImpl = new HelloImpl();
HelloPOATie helloTie = new HelloPOATie(helloImpl);
Hello href = helloTie._this();

POA rootpoa = POAHelper.narrow(
    orb.resolve_initial_references("RootPOA"));
rootpoa.the_POAManager().activate();

NamingContextExt ncRef = NamingContextExtHelper.narrow(
    orb.resolve_initial_references("NameService"));
ncRef.rebind(ncRef.to_name("HelloWorld"), href);

orb.run();
```

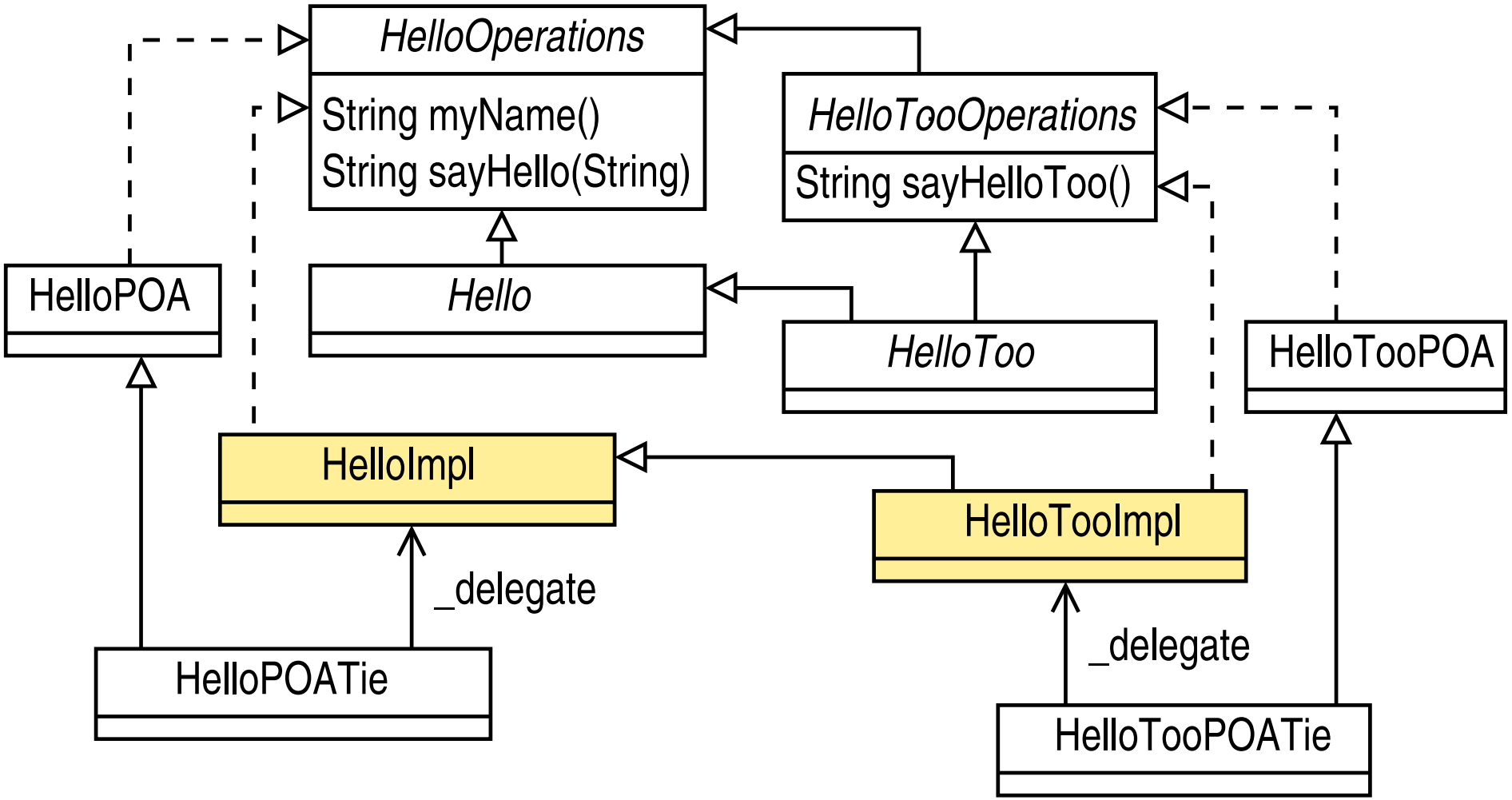
Beispiel: Vererbung der Implementierung

- IDL-Schnittstelle:

```
module HelloWorld
{
  interface Hello
  {
    string myName();
    string sayHello(in string Name);
  };
  interface HelloToo : Hello
  {
    string sayHelloToo();
  };
};
```

Beispiel: Vererbung der Implementierung ...

- Klassendiagramm der Implementierung:



3.4.5 Statische und dynamische Aufrufe

- Statische Aufrufe:
  - schnittstellenspezifische *Stubs* und *Skeletons* werden zur Übersetzungszeit aus IDL generiert
- *Dynamic Invocation Interface* (DII):
  - erlaubt dem Client Methoden aufzurufen, ohne daß Schnittstelle zur Übersetzungszeit bekannt sein muß
    - für generische Clients, z.B. Property-Editor, graphische Entwicklungsumgebung
  - Information zur Schnittstelle kann zur Laufzeit z.B. über *Interface Repository* gewonnen werden
  - Client muß *Request* explizit erzeugen und absenden
    - auch asynchrone Aufrufe möglich

3.4.5 Statische und dynamische Aufrufe ...

- *Dynamic Skeleton Interface* (DSI):
  - ermöglicht Objekt-Implementierungen, die zur Übersetzungszeit die Schnittstelle nicht kennen, z.B.:
    - Interpreter, Debugger, dynamisch getypte Sprachen wie Lisp
    - generische CORBA-Schnittstelle für Objekte einer objektorientierten Datenbank
    - generische Proxies (in Verbindung mit DII)
  - Objektadapter reicht alle Aufrufe an dieselbe Methode `invoke()` der Objektimplementierung weiter

Beispiel für dynamischen Methodenaufruf über DII

```
NVList argList = orb.create_list(1);   // Argumentliste aufbauen
Any arg = orb.create_any();
arg.insert_string("Roland");
NamedValue nv = argList.add_value("name", arg, ARG_IN.value);

Any res = orb.create_any();   // Ergebnisobjekt erzeugen
res.insert_string("");
NamedValue resv = orb.create_named_value("result", res,
                                         ARG_OUT.value);

// Request erzeugen: result = sayHello(name)
Request req = helloRef._create_request(null, "sayHello",
                                       argList, resv);

req.invoke();   // Request ausführen
                // Alternativ: req.send_deferred(); ...; req.get_response();

String s = res.extract_string();   // Ergebnis extrahieren
```

3.4.6 Portable Object Adapter (POA)

- Komponenten des POA-Modells:
  - *Servant*:
    - laufende Instanz einer Objektimplementierung
    - Lebenszeit von *Servant* und Objekt entkoppelt, Objekt ggf. nacheinander durch mehrere Servants realisiert
    - *Servant* kann mehrere Objekte implementieren
  - Objekt-ID:
    - im Bereich eines POA eindeutige ID
    - zur Zuordnung von Anfragen an Servants
  - Objektreferenz:
    - enthält eindeutige ID des POA und Objekt-ID
    - zusätzlich ggf. Ort des Servers / POAs, ...

Client/Server-Programmierung

WS 2019/2020

08.11.2019

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 17. Januar 2020

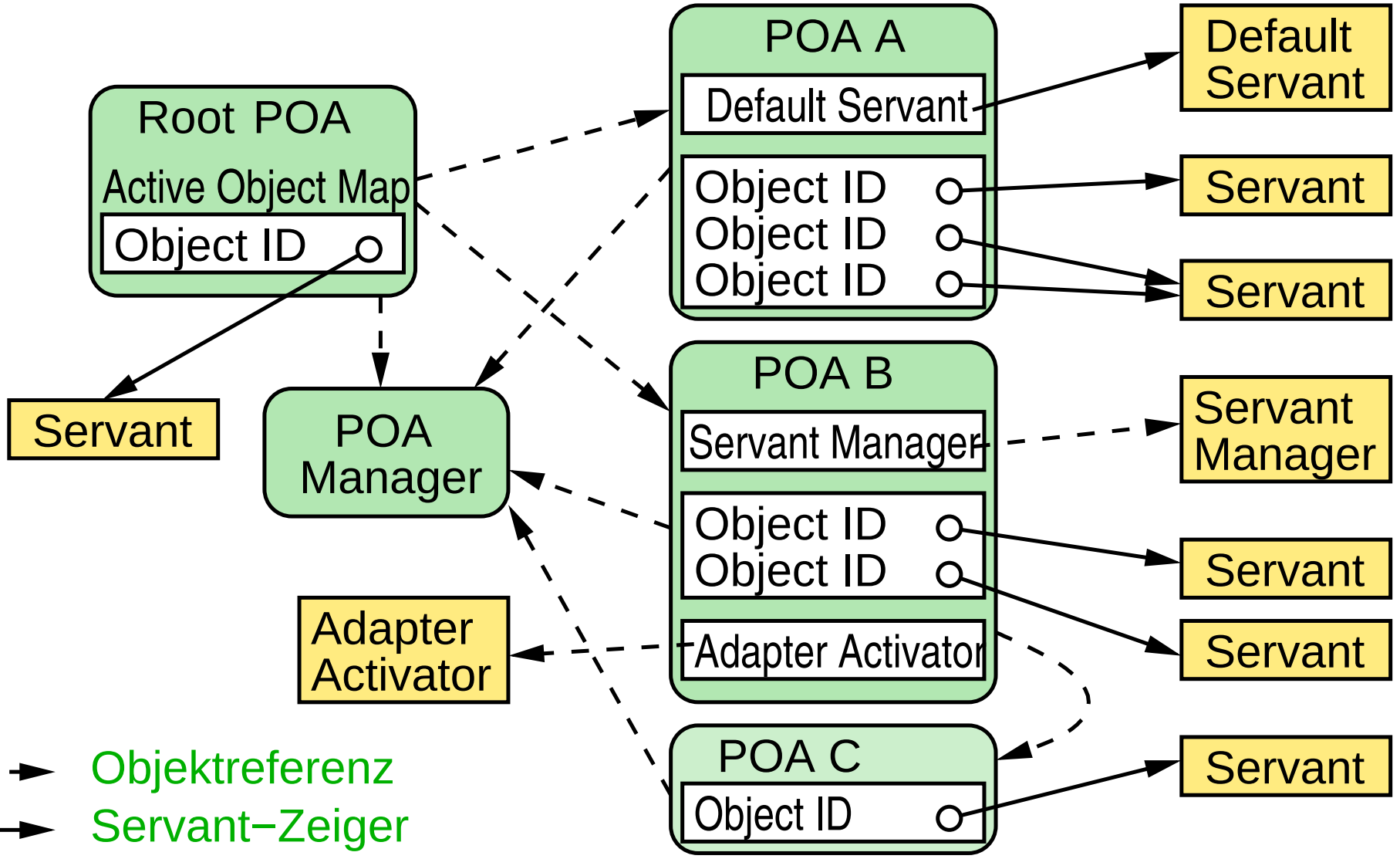
3.4.6 Portable Object Adapter (POA) ...

- Komponenten des POA-Modells: ...
  - *POA*:
    - Namensraum für Objekt-IDs
    - bildet Objekt-IDs auf *Servant*s ab
    - bestimmt über *POA Policies* das Verhalten der von ihm verwalteten *Servants*
  - *Active Object Map*:
    - Abbildung von Objekt-IDs auf *Servant*s
    - aktives Objekt: Objekt-ID steht in *Active Object Map*
    - aktiver *Servant*: ist in *Active Object Map* eingetragen
  - *Default Servant*:
    - bearbeitet Anfragen für inaktive Objekte
    - muß bei Bedarf von der Anwendung registriert werden

3.4.6 Portable Object Adapter (POA) ...

- Komponenten des POA-Modells: ...
  - *Servant Manager*:
    - kann bei Bedarf von der Anwendung registriert werden
    - übernimmt bei Anfragen an inaktive Objekte ggf.:
      - Aktivierung eines neuen *Servants*
      - Zuordnung der Objekt-ID zu vorhandenem *Servant*
  - *POA Manager*:
    - kapselt Zustand von POAs
    - kann POA aktivieren oder veranlassen, daß Anfragen (temporär) zwischengespeichert oder verworfen werden
  - *Adapter Activator*:
    - aktiviert (Kind-)POA, wenn Server Anfrage an nichtexistierenden POA erhält

POA-Architektur



POA *Policies*

- *Thread Policy*: Multithreading im Server?
  - `SINGLE_THREAD_MODEL`: POA ruft Methoden eines *Servants* sequentiell auf
  - `ORB_CRTL_MODEL`: implementierungsspezifisch, mehrere Threads erlaubt
- *Lifespan Policy*: Lebensdauer der Objekte
  - `TRANSIENT`: Objekte leben nur so lange wie POA
  - `PERSISTENT`: Objekte können länger leben wie POA
    - POA mit gegebenem Namen; bei Server-Neustart:
      - Server muß wieder ursprünglichen Port benutzen
      - Server muß POA gleichen Namens erzeugen
    - (nicht: Objektzustand wird persistent gespeichert)

POA *Policies* ...

- *Object ID Uniqueness Policy*
  - `UNIQUE_ID`: nur eine Objekt-ID pro *Servant*
  - `MULTIPLE_ID`: mehrere Objekt-IDs pro *Servant* erlaubt
- *ID Assignment Policy*: wer erzeugt Objekt-IDs?
  - `USER_ID`: Anwendung, `SYSTEM_ID`: POA
- *Request Processing Policy*
  - `USE_ACTIVE_OBJECT_MAP_ONLY`: Objekt-IDs werden nur über *Active Object Map* (AOM) auf *Servants* umgesetzt
  - `USE_DEFAULT_SERVANT`: Wenn Objekt-ID nicht in AOM ist: Anfrage an registrierten *Default Servant* leiten
  - `USE_SERVANT_MANAGER`: Wenn Objekt-ID nicht in AOM ist: über registrierten *Servant Manager* neuen Servant erzeugen

POA *Policies* ...

- *Servant Retention Policy*
  - [NON_]RETAIN: *Servants* werden [nicht] in AOM aufgenommen
- *Implicit Activation Policy*
  - IMPLICIT_ACTIVATION: implizite Aktivierung von Servants durch POA, z.B. bei
    - POA.servant_to_reference(), POA.servant_to_id()
    - ggf. Typkonvertierung von Servant nach Objektreferenz
  - NO_IMPLICIT_ACTIVATION: Servant muß explizit aktiviert werden
    - POA.activate_object(): POA generiert Objekt-ID
    - POA.activate_object_with_id(): Anwendung generiert Objekt-ID

Standard-*Policy* des Root-POA

- *Thread Policy*: `ORB_CRTL_MODEL`
- *Lifespan Policy*: `TRANSIENT`
- *Object ID Uniqueness Policy*: `UNIQUE_ID`
- *ID Assignment Policy*: `SYSTEM_ID`
- *Request Processing Policy*: `USE_ACTIVE_OBJECT_MAP_ONLY`
- *Servant Retention Policy*: `RETAIN`
- *Implicit Activation Policy*: `IMPLICIT_ACTIVATION`

- *Policy* kann nicht geändert werden, ggf. muß neuer POA erzeugt werden

Beispiel: Leichtgewichtige Objekte ...

- Objekte mit „kleinem“ Zustand
  - Attribute können in Objekt-ID eincodiert werden
- Alle Objekte werden durch einen einzigen (*Default-*)*Servant* realisiert
- Vorteil: Skalierbarkeit
  - geringer Ressourcenverbrauch beim Server
- Anmerkung: mit DSI kann ein einziger *Servant* sogar Objekte unterschiedlicher Klassen implementieren ...
- Vollständiger Code: siehe WWW

3.4.7 GIOP, IIOP und IOR

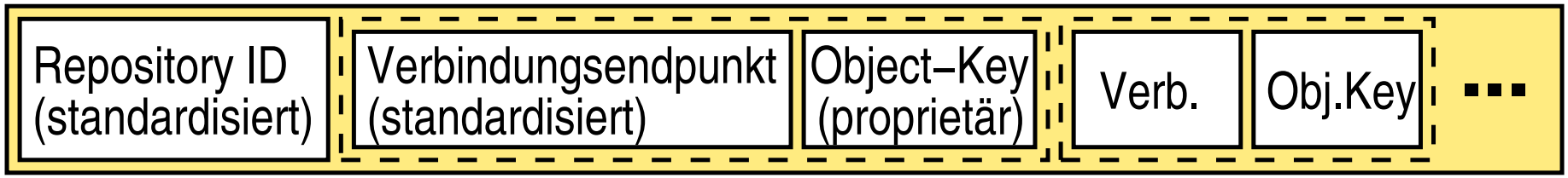
- Seit CORBA 2.0: Kommunikationsprotokoll zwischen Objekten bzw. ORBs ist standardisiert
  - GIOP: *General Inter-ORB Protocol*
    - Spezifikation, wie Protokolle auszusehen haben
  - IIOP: *Internet Inter-ORB Protocol*
    - konkretes Protokoll auf Basis von TCP/IP
- Damit: Interoperabilität zwischen verschiedenen ORB-Implementierungen
- IOR: *Interoperable Object Reference*
  - Aufbau von Objektreferenzen ist ebenfalls standardisiert

GIOP

- Spezifiziert unter anderem:
  - Annahmen über Transportschicht
    - verbindungsorientiert, duplex, zuverlässig, stromorientiert
  - binäres Übertragungsformat für IDL-Datentypen
    - CDR: *Common Data Representation*
    - unterstützt *Little-Endian* und *Big-Endian*
      - über *Tag*, Empfänger konvertiert bei Bedarf
    - keine Typ-*Tags*, d.h. Empfänger muß Datentyp kennen (Problem ggf. bei dynamischen Aufrufen)
  - acht Nachrichtenformate
    - *Request*, *Reply*, *CloseConnection*, ...

IIOP und IOR

- Implementierung des GIOP auf Basis von TCP/IP
- Hauptaufgabe: Festlegung des konkreten Aufbaus von IORs
- Genereller Aufbau einer IOR:



  - *Object-Key* enthält Objekt-ID und POA-Namen
  - mehrere Einträge für Verbindungsendpunkt / *Object-Key* möglich, damit z.B.
    - Objekt über verschiedene GIOP-Protokolle erreichbar
    - Objekt mehrfach vorhanden (Lastausgleich, Fehlertoleranz)

IIOP und IOR ...

- Verbindungsendpunkt bei IIOP: Host / IP-Adresse und Port
- Beispiel für Inhalt einer IOR

```
TypeId : IDL:omg.org/CosNaming/NamingContextExt:1.0
TAG_INTERNET_IOP Profiles:
  Profile Id:      0
  IIOP Version :   1.2
  Host:            141.99.179.102
  Port:            33523
  Object key (URL):     StandardNS/NameServer-POA/_root
  Object key (hex):     0x53 74 61 6E 64 61 72 64 4E 53
                        2F 4E 61 6D 65 53 65 72 76 65
                        72 2D 50 4F 41 2F 5F 72 6F 6F 74
```

- Formatiert mit dem `dior`-Tool des JacORB

Nutzung von IORs

- IORs in String-Form können beliebig zwischen Objekten ausgetauscht werden, z.B.
  - als Methoden-Parameter oder -Ergebnis
  - über Dateien, WWW, ...
- Damit: Objekte können auch ohne *Name Service* bekanntgegeben werden
- Relevante Methoden (in Klasse ORB):
  - `object_to_string()` : Erzeugt String-Form einer CORBA-Objektreferenz
  - `string_to_object()` : Wandelt String wieder in CORBA-Objektreferenz um

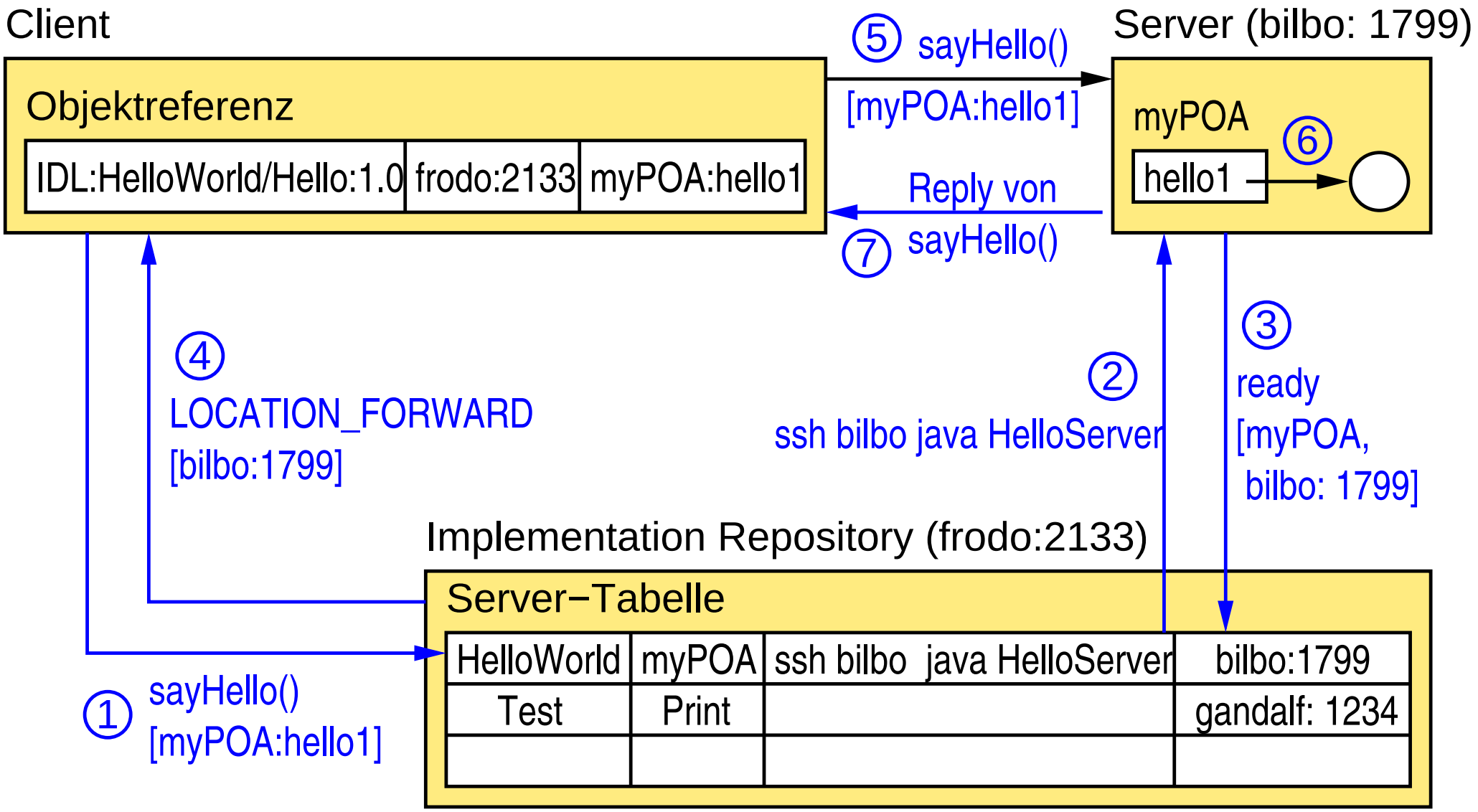
3.4.8 *Implementation Repository* (IMR)

- ORBs unterstützen typischerweise zwei Methoden, wie IORs an *Servants* gebunden werden:
  - direktes Binden: IOR enthält Host/Port des Servers, in dem *Servant* läuft
  - indirektes Binden: IOR enthält Host/Port eines externen Brokers, d.h. des IMR
    - IMR kennt Ort (Host/Port) des Servers
- IMR unterstützt u.a.:
  - automatischer Server-Startup
  - Migration von Servern bzw. Objekten
  - automatischer Lastausgleich

3.4.8 *Implementation Repository* (IMR) ...

- CORBA Spezifikation standardisiert nur Interaktion zwischen Clients und IMR
  - ausreichend um Interoperabilität zu sichern
- Realisierung und Funktionsumfang des IMR sowie Schnittstelle zu Servern ist ORB-spezifisch

Beispiel: Automatischer Server-Start



Beispiel: Automatischer Server-Start ...

1. Client ruft Methode über IOR auf, Anfrage wird an IMR geleitet
2. IMR sieht in Server-Tabelle nach, ob POA existiert.
   Falls nicht, wird Server (mit POA) gestartet
3. Server sendet Fertig-Meldung an IMR, mit Host/Port.
   IMR trägt Information in Server-Tabelle ein
4. IMR konstruiert neue IOR, die Host/Port des Servers enthält
   und sendet `LOCATION_FORWARD`-Nachricht an Client
5. Client sendet Anfrage zum zweiten Mal, diesmal an Server
6. Server findet POA über POA-Namen. POA findet passenden
   Servant über Objekt-ID und ruft Methode auf
7. Ergebnis wird an Client zurückgegeben

3.4.9 *Interface Repository* (IR)

- Online „Datenbank“ mit Schnittstellenbeschreibungen
- CORBA spezifiziert nur Zugriffsmethoden (lesen / schreiben)
  - Implementierung ist ORB-spezifisch
  - ebenso, wie IDL-Spezifikationen in das IR kommen
- IR nützlich u.a. für:
  - Typprüfung der Parameter durch den ORB (auch bei DII)
  - Verbindung mehrerer ORBs
  - Metadaten für Clients und Server (für DII, DSI)
    - z.B. Klassenbrowser, Anwendungsgeneratoren, ...
  - selbstbeschreibende Objekte (*Introspection*)

Wichtige Klassen und Methoden

- IR definiert Klassen (genauer: Schnittstellen) für alle IDL-Konstrukte, z.B.:
  - `ModuleDef`, `InterfaceDef`, `OperationDef`, `ParameterDef`, ...
- Objekte der Klassen enthalten einander, entsprechend der Verschachtelung in der IDL
  - Basisklassen `Container` und `Contained`
  - `InterfaceDef` enthält mehrere `OperationDefs`, ...
- Von `Contained` erbt jede Klasse die Methode `describe()`
  - liefert Beschreibung des IDL-Konstrukts als *Struct*
    - z.B. `struct OperationDescription` enthält Name, Repository-ID, Ergebnistyp, Parameterliste, Exceptions, ...

Startpunkte für die IR-Information

- Methode `_get_interface_def()` liefert `InterfaceDef` eines Objekts:

```
Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));
org.omg.CORBA.Object c = helloRef._get_interface_def();
org.omg.CORBA.InterfaceDef id =
    org.omg.CORBA.InterfaceDefHelper.narrow(c);
```

- Methode `lookup_id()` des IR liefert IR-Objekt zu einer Repository-ID
  - Repository-ID beschreibt IDL-Element eindeutig, z.B. im Hello-World-Beispiel:
    - `IDL:HelloWorld/Hello:1.0` für Interface `Hello`
    - `IDL:HelloWorld/Hello/sayHello:1.0` für Operation

3.4.10 Sicherheit

- CORBA-Spezifikation enthält *Security Attribute Service*
- Basis ist GIOP über (z.B.) SSL/TLS
  - sichert Vertraulichkeit/Integrität der Nachrichten
  - sichert Authentifizierung des Servers
- *Security Attribute Service* realisiert zusätzlich:
  - Authentifizierung des Clients
    - u.a. über Benutzername und Paßwort
  - Weitergabe von Sicherheitsattributen (z.B. Identität, Privilegien) des Clients an den Server
    - damit: Delegation von Rechten möglich, d.h. Server kann für Benutzer agieren

3.5 Zusammenfassung

- CORBA definiert ein verteiltes, sprach- und plattform-unabhängiges Objektmodell
  - standardisiertes Protokoll (GIOP, IIOP) stellt Interoperabilität sicher
- Zentral: Schnittstellenbeschreibung über OMG IDL
  - automatische Erzeugung von *Stubs* und *Skeletons*
- Keine 1-zu-1 Beziehung zwischen Objekten und Objektimplementierungen (*Servants*)
  - POA kann unterschiedliche *Policies* realisieren
- CORBA spezifiziert Schnittstellen zu etlichen Diensten
  - Namensdienst, Ereignisdienst, Lebenszyklus, ...