

B 7. Übung: CORBA-Programmierung

B.1 Überblick über die 7. Übung

- Lösung der 5. Aufgabe (RMI)
- Einführung in CORBA
- Verwendung von CORBA-Objekten
- Interface Definition Language (IDL)

Übungen zu

Objektorientierte Konzepte in Verteilten Systemen und Betriebssystemen

Winter 2002/2003

Teil 2: CORBA

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

A Organisation

A Organisation

- 2./3. Dezember 2002
 - ◆ CORBA Einführung, IDL, *Bespr. Aufgabe 5*
- 8./9. Dezember 2002
 - ◆ Mapping von IDL nach Java, Java-Client
- 16./17. Dezember 2002
 - ◆ serverseitiges Mapping, POA, Namensdienst, (*Bespr. Aufgabe 7 ??*)
- 23. 12. 2002 - 6. 1. 2003 Weihnachtsferien!!
- 7. Januar 2003
 - ◆ Übung entfällt
- 13./14. Januar 2003
 - ◆ CORBA Dienste, Java Security API.(*Bespr. Aufgabe 8 ??*)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B.2 Lösung der 5. Aufgabe

B.2 Lösung der 5. Aufgabe

- Teilaufgabe (a)
 - ◆ ShapeContainer
 - ◆ RMIServer
 - ◆ WhiteBoard anpassen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 ShapeContainer

```
package whiteboard;

import whiteboard.shapes.Shape;

import java.io.*;
import java.rmi.*;
import java.util.Vector;

public interface ShapeContainer extends Remote {

    public void addShape(Shape shape) throws RemoteException;

    public void updateShape(Shape shape) throws RemoteException;

    public Vector getShapes() throws RemoteException;

    public Shape getShape(int x, int y) throws RemoteException;
    /* ... */
    /* Teilaufgabe (b) */
    public void registerCallback(RedrawListener listener)
        throws RemoteException;
} // ShapeContainer
```

2 RMIServer

```
package whiteboard.server;

import whiteboard.ShapeContainer;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class RMIServer {
    public static void main(String argv[]) {

        if (argv.length<1) { /* Fehlerbehandlung */ }

        ShapeContainer container = new ShapeContainerImpl();
        Remote rcontainer=null;
        try {
            rcontainer = UnicastRemoteObject.exportObject(container);
        } catch (Exception ex) { /* Fehlerbehandlung */ }

        try {
            Naming.rebind(argv[0], rcontainer);
        } catch (Exception ex) { /* Fehlerbehandlung */ }
    }
}
```

3 WhiteBoard

```
package whiteboard.client;

/*...*/
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject; /* (b) */

public class WhiteBoard extends JFrame implements RedrawListener {

    /*...*/

    public WhiteBoard (String regName) {

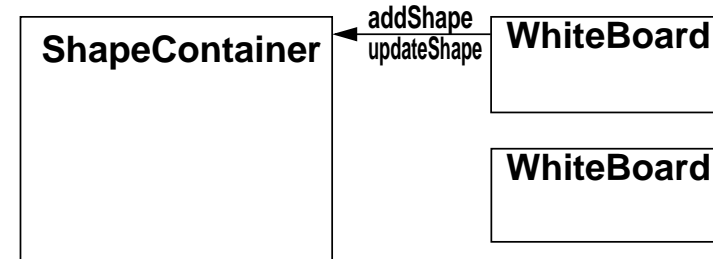
        super("WhiteBoard");

        try {
            shapeContainer = (ShapeContainer)Naming.lookup(regName);
        } catch (Exception ex) { /* Fehlerbehandlung */ }

        /* ... */
    }
}
```

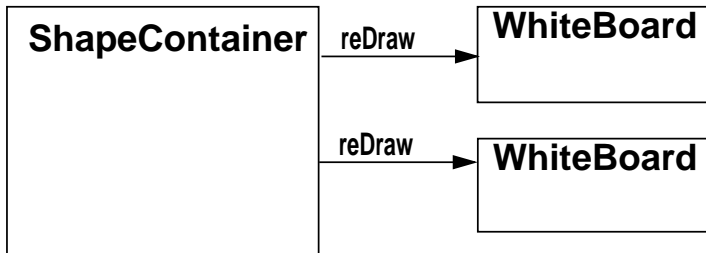
B.2 Lösung der 5. Aufgabe (2)

- Teilaufgabe (b)
 - ◆ RedrawListener



B.2 Lösung der 5. Aufgabe (3)

- Teilaufgabe (b)
- ◆ RedrawListener



4 RedrawListener

```

package whiteboard;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RedrawListener extends Remote {
    public void reDraw() throws RemoteException;
}
  
```

```

public class WhiteBoard extends JFrame implements RedrawListener {

    public WhiteBoard (String regName) {
        /* ... */
        try {
            RedrawListener l =
                (RedrawListener)UnicastRemoteObject.exportObject(this);
            shapeContainer.registerCallback(l);
        } catch (Exception ex) { /* Fehlerbehandlung */ }

        public void reDraw() {
            RepaintManager.currentManager(drawing).markCompletelyDirty(drawing);
        }
    }
}
  
```

B.3 Einführung in CORBA

- Inhalt: Programmierung von CORBA-Anwendungen
- Implementierungssprache: Java wird verwendet (aber: CORBA ist Programmiersprachenunabhängig!)
- Allgemeine Konzepte von CORBA: siehe Vorlesung
- Vor allem: Betrachtung von praktische Problemen
- Als Ergänzung: Behandlung von speziellen CORBA-Features

1 CORBA

- Common Object Request Broker Architecture and Specification (CORBA) – Zentrale Spezifikation ("The Core Spec")
- Weitere separate Spezifikationen, die auf CORBA aufbauen
 - ◆ CORBAServices
 - ◆ (CORBAfacilities)
 - ◆ Domain Interfaces
 - ◆ CORBA Component Model
 - ◆ (Unified Modelling Language & Meta Object Facility)
- Viele "Task Forces" und "Special Interest Groups" innerhalb der OMG
 - ◆ Ergänzung von neuer Konzepte und Erweiterungen
 - ◆ Revision von existierenden Standards
- Alle OMG-Spezifikationen werden ständig weiterentwickelt!

2 CORBA-Versionen

- CORBA 1.x (October 1991)
 - ◆ CORBA-Objektmodell und Architektur
 - ◆ Schnittstellenbeschreibungssprache (Interface Definition Language, IDL)
 - ◆ Sprachabbildungen für C, C++ und Smalltalk
- CORBA 2.0 (Juli 1996)
 - ◆ Interoperabilität durch IOP als Protokoll, das alle ORB-Implementierungen unterstützen müssen
- CORBA 2.1 (August 1997)
 - ◆ IDL-Erweiterungen
 - ◆ Neue Sprachabbildungen (Cobol, Ada)
- CORBA 2.2 (Februar 1998)
 - ◆ *Portable Object Adaptor* (POA) ersetzt *Basic Object Adaptor* (BOA)
 - ◆ Neue Sprachabbildung (Java)

2 CORBA Versionen (2)

- CORBA 2.3/2.3.1 (Juni/Oktober 1999)
 - ◆ Überarbeitete Sprachabbildungen zur Anpassung an den POA
 - ◆ Valuetypes, object-by-value-Parameter
 - ◆ Separate Dokumente für die Sprachabbildungen
- CORBA 2.4/2.4.1 (Oktober/November 2000)
 - ◆ CORBA Messaging, Minimum CORBA, Real-time CORBA
- CORBA 2.5 (Oktober 2001)
- CORBA 2.6 (Dezember 2001)
 - ◆ Common Security
- CORBA 3 – der nächste offizielle Release
 - ◆ "Huge hype"
 - ◆ CORBA-Komponentenmodell

3 Informationen zu CORBA

- Wer wirklich alle Details zu CORBA wissen will, muss letztendlich die Spezifikationen lesen!
- Spezifikationen sind öffentlich verfügbar
 - ◆ OMG Webseite: <http://www.omg.org/>
 - ◆ Webseiten der OOVS-Übung
- Viele Bücher von sehr unterschiedlicher Qualität
- Vorsicht vor CORBA-Produktdokumentation
 - ◆ Oft werden proprietäre Erweiterungen beschrieben

4 CORBA-Produkte vs. CORBA-Standard

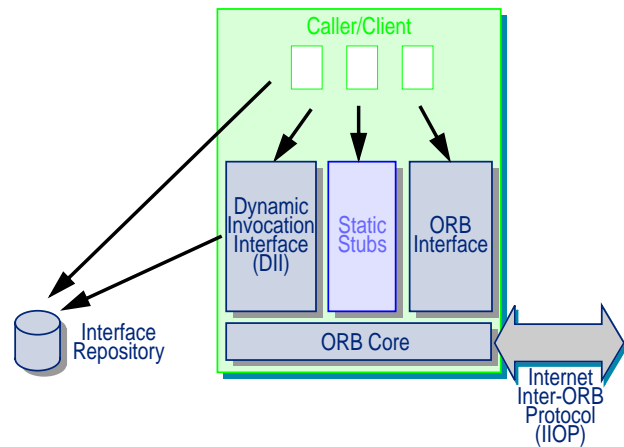
- Keine etabliertes "CORBA"-Zertifikat
 - ◆ Jeder kann behaupten, zu CORBA Version x.y kompatibel zu sein
 - ◆ Open Group beginnt mit Produkt-Zertifizierung (seit CORBA 2.1)
- CORBA-Produkte führ(t)en proprietäre Erweiterungen ein
 - ◆ Gut, solange man sich nicht darauf verlässt
 - ◆ In der CORBA-Frühzeit ging es oft nicht anders, z.B. BOA
 - ◆ Heute kann man (fast) alles auf einem standardisierten Weg erreichen
- Einige Features in Produkten entsprechen nicht 100%ig den Spezifikationen
 - ◆ Z.B. Sprachabbildungen
 - ◆ Spezifikationen ändern sich, Produkte erst etwas später

B.4 Verwendung von CORBA-Objekten

1 CORBA-Objekte aus Sicht des Klienten

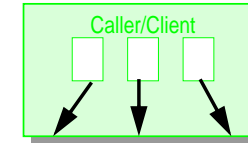
- CORBA-Objekte haben (genau) eine Schnittstelle (Interface)
 - ◆ Beschreibung der Schnittstelle in der CORBA Interface Definition Language
 - ◆ IDL-Schnittstellen sind ein "Vertrag" zwischen dem CORBA-Objekt und dessen Aufrufern
- Aufrufer eines CORBA-Objekts haben nur eine (opake) Objektreferenz
 - ◆ Objekt-Methoden werden über die Referenz aufgerufen
 - ◆ Das Objekt kann lokal oder entfernt sein
 - ◆ Objektschnittstelle kann abgefragt werden (vgl. Java Reflection)
 - ◆ Aufrufe können zur Laufzeit erzeugt werden (Dynamic Invocation)
- Object Request Broker (ORB) übermittelt Aufrufe und Antworten
 - ◆ Nur der ORB kann Objektreferenzen interpretieren
- Aufrufer/Klient ist nur eine *Rolle* bei einem Aufruf (z.B. Callbacks)

2 Architektur des Aufrufer/Klienten



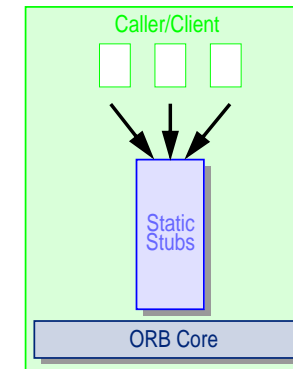
3 Der Klient

- Ruft Operationen an CORBA-Objekten auf
- Muss selbst kein CORBA-Objekt sein



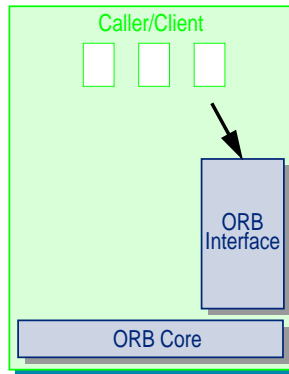
4 Statische Stubs

- Können automatisch aus der IDL-Schnittstelle erzeugt werden.
- Marshalling der Aufrufparameter
- Demarshalling der Rückgabewerte/Exceptions des Aufrufes



5 ORB-Schnittstelle

- Export von ersten Objektreferenzen (ORB, POA, Services, ...)
- Verarbeitung von Objektreferenzen (Umwandlung in Strings und umgekehrt)

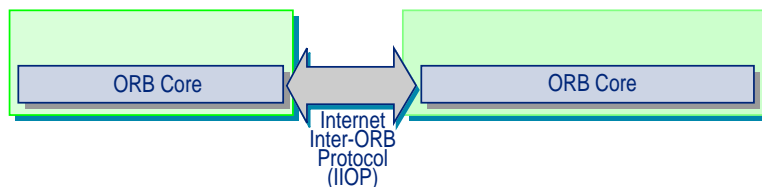


6 Der Kern des ORB (ORB Core)

- Übertragung von Aufrufen mit Hilfe von Informationen in den Objektreferenzen

7 General Inter-ORB Protocol (GIOP)

- Standard-Transportprotokoll zwischen ORBs
- Grundlage für die Interoperabilität
- GIOP über TCP-Verbindungen: Internet Inter-ORB Protocol (IIOP)
- Jeder CORBA 2.x ORB muss IIOP implementieren



8 Zusammenfassung Aufrufer/Klient

- Müssen selbst keine CORBA-Objekte sein
- Können Operationen an CORBA-Objekten aufrufen
- Opake Objektreferenzen
- ORB überträgt die Aufruf-Daten

B.5 Interface Definition Language (IDL)

- Grundlegendes
- Bezeichner (Identifiers)
- Primitive Datentypen
- Zusammengesetzte Datentypen
- Schnittstellen von CORBA-Objekten
- Valuetypes
- Designfragen

1 Grundlegendes

- IDL dient der Beschreibung von Datentypen und Schnittstellen
- Unabhängig von der/den Implementierungs-Programmiersprache(n)
- Syntax ist stark an C++ angelehnt
 - ◆ Beschreibung von Daten und Schnittstellen (Typen, Attribute, Methoden, ...)
 - ◆ Keine steuernden Anweisungen (if, while, for, ...)
- Präprozessor wie in C++
 - ◆ `#include` um andere IDL-Dateien einzubinden
 - ◆ `#define` für Makros
- Kommentare wie in C++ und Java:

```
// Das ist ein einzeliger Kommentar
/*
 * Das ist ein mehrzeiliger
 * Kommentar
 */
```

2 Bezeichner (Identifiers)

- Bestimmte reservierte Wörter
 - ◆ `module`, `interface`, `struct`, `void`, `long`, ...
- Alle anderen Kombinationen von kleinen und grossen Buchstaben, Zahlen und Unterstrichen sind erlaubt
 - ◆ Erstes Zeichen muss Buchstabe sein!
- "_" als Escape-Zeichen für reservierte Wörter
 - ◆ z.B. `"_module"`, um einen Bezeichner `"module"` zu erzeugen
 - ◆ vereinfacht Erweiterung der Menge der reservierten Wörter

2 Bezeichner (Identifiers)

- Sobald ein Bezeichner benutzt ist, sind alle anderen Varianten mit anderen Gross-/Kleinschreibung verboten!

- Beispiel:

```
module Beispiell { ... };
module BEISPIEL1 { ... }; // illegal in IDL
```

- Sinn:

- ◆ Erlaubte Abbildung von IDL zu Sprachen, die nicht "case-sensitive" sind
- ◆ Erhalte Schreibweise von Bezeichner für "case-sensitive" Sprachen

3 Module

- Namensraum (scope) für IDL-Deklarationen

- Syntax:

```
module Name {
    Deklarationen
};
```

- Zugriff auf andere Namensräume über den "::"-Operator

- Beispiel:

```
module Beispiell {
    typedef long IDNumber;
};
module Beispiel2 {
    typedef Beispiell::IDNumber MyID; // typedef long MyID;
};
```

4 Datentyp-Deklarationen

- Alias für einen existierenden Datentyp

- Syntax:

```
typedef existing_type alias;
```

- Beispiel:

```
typedef long IDNumber;
```

5 Primitive Datentypen

- Integer-Zahlen

◆ short	-2^{15} to $2^{15}-1$
◆ unsigned short	0 to $2^{16}-1$
◆ long	-2^{31} to $2^{31}-1$
◆ unsigned long	0 to $2^{32}-1$
◆ long long	-2^{63} to $2^{63}-1$
◆ unsigned long long	0 to $2^{64}-1$

- Fließkommazahlen (IEEE-Standard für binäre Fließkommazahlen-Arithmetik, ANSI/IEEE Std 754-1985)

◆ float	einfache Genauigkeit
◆ double	doppelte Genauigkeit
◆ long double	erweiterte Genauigkeit (mindestens 15 Bit Exponent und 64 Bit Basis)

5 Primitive Datentypen (2)

- Zeichen

◆ char	ISO 8859-1 (Latin1) Zeichen
◆ wchar	multi-byte character (z.B. Unicode)
◆ Länge hängt von Implementierung und Programmiersprache ab	

- **boolean**

◆ Nur die Werte TRUE und FALSE	
--------------------------------	--

- **octet**

◆ Länge 8 bit	
◆ Keine Konvertierung bei der Übertragung	

- **any**

◆ Kapselung für beliebigen CORBA-Datentyp	
---	--

- **void**

6 Strukturen

- Gruppierung von mehreren Typen in einer Struktur

- Syntax:

```
struct Name {
    Deklaration von Struktur-Elementen
};
```

- Beispiel:

```
struct AmountType {
    float value;
    char currency;
};
```

- Verwendung:

```
AmountType amount;
```


6 Geschachtelte Strukturen

- Strukturen können innerhalb anderer Strukturen definiert werden
- Beispiel:

```
struct AmountType {
    struct ValueType {
        long integerPart;
        short fractionPart;
    } amount;
    char currency;
};
```

- Strukturen erzeugen einen eigenen Namensraum (scope)!
- Kompletter Name von obigem Typ:

```
AmountType::ValueType
```

7 Aufzählungen

- Aufzählungen mit festgelegter Menge an möglichen Werten
- Syntax:

```
enum name {
    value1, value2, ...
};
```

- Beispiel:

```
enum Color {
    GREEN, RED, BLUE
};
```

7 Aufzählungen

- Achtung: Aufzählungen erzeugen keinen eigenen Namensraum!
- Zugriff auf Werte von Aufzählungen:

```
GREEN
not Color::GREEN
```

- Nicht erlaubt:

```
interface A {
    enum E { E1, E2, E3 }; // line 1
    enum BadE { E3, E4, E5 }; // Fehler: E3 is bereits
                                // definiert
}
```

8 Unions (Verbünde)

- Verbünde von verschiedenen Datentypen, die über einen Diskriminator-Wert unterschieden werden

- Syntax:

```
union Name switch( switch_type ) {
    case switch_constant: Deklaration
    ...
    default: Deklaration
};
```

- Mögliche Diskriminator-Typen:
Ganzzahlen, Zeichen, `boolean`, Enumerations
- Bezeichner in den Deklarationen müssen eindeutig sein
- Beispiel:

```
union Beispiel switch( long ) {
    case 1: long l;
    case 2: float f;
};
```

9 Arrays

- Ein- und Mehrdimensionale Arrays
 - ◆ Feste Grösse in jeder Dimension

- Syntax:

```
typedef element_type name[positive_constant][positive_constant]...;
```

- Beispiel:

```
typedef long Matrix[3][3];
```

- Achtung:
Array-Datentypen müssen mit `typedef` deklariert werden, bevor man sie verwenden kann!

10 Sequences

- Eindimensionales Array
 - ◆ Variable Grösse
 - ◆ Optional maximale Grösse ("bounded sequence")

- Syntax:

```
typedef sequence<element_type> name; // unbounded
typedef sequence<element_type, positive_constant> Name; // bounded
```

- Beispiel:

```
typedef sequence<long> Longs;
typedef sequence< sequence<char> > Strings;
```

- Achtung:
Auch Sequence-Datentypen müssen vor Verwendung mit `typedef` deklariert werden!

11 Zeichenketten

- Zeichenketten
 - ◆ Ähnlich zu `sequence<char>` und `sequence <wchar>`
 - ◆ Spezieller Datentyp aus Performance-Gründen
 - ◆ Zeichenketten müssen nicht mit `typedef` deklariert werden
 - ◆ Ebenfalls optional maximale Grösse festlegbar

- Syntax:

```
typedef string name; // unbounded
typedef string<positive_constant> name; // bounded
typedef wstring name; // unbounded
```

- Beispiel:

```
typedef string<80> Name;
```

12 Festkomma-Zahlen

- Ähnlich zu Integer-Zahlen
 - ◆ Bis zu 31 Stellen
 - ◆ Skalierungsfaktor für das Dezimalkomma

- Syntax:

```
typedef fixed<positive_constant, scaling_constant> name;
```

- Beispiel:

```
typedef fixed<10, 2> Amount;
```

- Achtung:
In vielen ORBs noch nicht implementiert!

13 Konstanten

- Symbolische Namen für spezielle Werte
- Syntax:


```
const type Name = Konstantenausdruck;
```
- Konstantenausdruck
 - ◆ Konstante Werte (Zahlen/Zeichen/Zeichenketten/Enums je nach *type*)
 - ◆ Arithmetische Operationen
 - ◆ Logische Operationen
- Beispiel:


```
const Color WARNING = RED;
```

14 Schnittstellen - Attribute

- Öffentliche Objektvariablen
 - ◆ Schreibzugriff kann verhindert werden (Nur-Lese-Attribute)
 - ◆ Keine Instanzvariablen
- Syntax:


```
attribute type name; // read & write
readonly attribute type name; // read-only
```
- Beispiel:


```
interface Account {
    readonly attribute float balance;
};
```

14 Schnittstellen (Interfaces)

- Sichtbare Schnittstelle von CORBA-Objekten
- Enthält:
 - ◆ Attribute
 - ◆ Operationen
 - ◆ Lokale Typen, Konstanten, Exceptions
- Syntax:


```
interface name {
    Deklaration von Attributen und Operationen (sowie Typen und Exceptions)
}
```
- Schnittstellen definieren ebenfalls einen eigenen Namensraum
- Die Namen von Attributen und Operationen müssen eindeutig sein!
 - ◆ Kein "Overloading!"

14 Schnittstellen – Operationen

- Methoden von CORBA-Objekten mit:
 - ◆ Methoden-Name
 - ◆ Rückgabe-Datentyp
 - ◆ Aufruf-Parameter
 - ◆ Exceptions
 - ◆ (Aufruf-Kontext)
- Syntax:


```
return_type name( parameter_list ) raises( exception_list );
```
- Nur der Methodenname ist signifikant
 - ◆ Kein Overloading durch Parametertypen
- Methodenaufruf mit "best-effort"-Semantik (keine Rückgabe-Werte und keine Exceptions erlaubt)


```
oneway void name( parameter_list );
```

14 Schnittstellen – Parameterübertragung

- Für jeden Parameter muss die Übertragungsrichtung angegeben werden:

- ◆ **in** nur vom Klienten zum Server
- ◆ **out** nur vom Server zum Klienten
- ◆ **inout** in beiden Richtungen

- Syntax:

```
( copy_direction1 type1 name1, copy_direction2 type2 name2, ... )
```

- Beispiel:

```
interface Account {
    void makeDeposit( in float sum );
    void makeWithdrawal( in float sum,
                        out float newBalance );
};
```

14 Schnittstellen – Vererbung (2)

- Weder "Overloading" noch "Overriding" ist erlaubt:

```
module Foo {
    interface A {
        void draw( in float num );
    };

    interface B {
        void print( in float num);
        void print( in string str); // Fehler: Overloading
    };

    interface C: A, B {
        void draw( in float num); // Fehler: Overriding
    };
};
```

14 Schnittstellen – Vererbung

- Ableitung von neuen Schnittstellen von existierenden
- Mehrfache Vererbung möglich

- Syntax:

```
interface name : inherited_interface1, inherited_interface2, ... {
    Declaration of additional attributes and operations
};
```

- Namen von geerbten Attributen und Operationen müssen eindeutig sein

- ◆ Ausnahme: Bezeichner, die auf verschiedenen Pfaden geerbt werden, aber von der selben Basisklasse stammen, sind erlaubt

14 Schnittstellen – Vererbung (3)

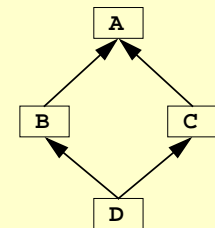
- Erlaubter Vererbungsgraph in CORBA:

```
module Foo {
    interface A {
        void draw( in float num );
    };

    interface B : A {
    };

    interface C : A {
    };

    interface D : B, C {
    };
};
```



15 Exceptions – User Exceptions

- Benutzer-Exceptions werden im Benutzercode auf Serverseite erzeugt und zum Klienten weitergereicht

- Syntax:

```
exception name {
    Declaration of data elements
};
```

- Exceptions sind eine spezielle Form von Strukturen

- ◆ Nur Datenelemente, keine Operationen

- ◆ **Keine Vererbung von Exceptions!**

- Beispiel:

```
interface Account {
    exception Overdraft { float howMuch; };
    void makeWithdrawal( in float sum )
        raises( Overdraft );
};
```

16 Vorwärtsdeklarationen

- Problem: Zirkuläre Abhängigkeiten in den Deklarationendeklarations
 - ◆ Schnittstelle **A** enthält Operation `op_b()`, die Objekt vom Typ **B** liefert
 - ◆ Schnittstelle **B** enthält Operation `op_a()`, die Objekt vom Typ **A** liefert

- Lösung: Vorwärtsdeklaration

- ◆ Deklariere einen Bezeichner für einen Typ, aber nicht den Typ selbst

- Beispiel:

```
interface B;           // Forward declaration
interface A {
    B op_b();
};
interface B {
    A op_a();
};
```

15 Exceptions – System Exceptions

- "System Exceptions" werden vom ORB bei internen Fehlern erzeugt

```
module CORBA {
    enum completion_status { COMPLETED_YES, COMPLETED_NO,
        COMPLETED_MAYBE };

    exception UNKNOWN {
        unsigned long    minor;
        completion_status completed;
    };
    exception BAD_PARAM {
        unsigned long    minor;
        completion_status completed;
    };
    exception NO_MEMORY {
        unsigned long    minor;
        completion_status completed;
    };
    exception COMM_FAILURE {
        unsigned long    minor;
        completion_status completed;
    };
};
```

17 Value types

- Semantische Verknüpfung von Strukturen und Schnittstellen
 - ◆ Unterstützt die Beschreibung eines komplexen Zustandes (z.B. beliebige Graphen, mit Rekursion und Zyklen)
 - ◆ Instanzen sind immer lokal im Kontext, in dem sie verwendet werden (weil sie immer kopiert werden, wenn sie als Parameter eines fernen Aufrufes übergeben werden)
 - ◆ Unterstützt sowohl öffentliche als auch private (für die Implementierung) Daten

- "Value types" unterstützen einfache **Vererbung** (von valuetype) und können eine Schnittstelle (interface) unterstützen

- Beispiel:

```
valuetype Person {
    public string name; // A public state
    private long id;   // A private state

    void print();      // An operation
};
```

18 Designfragen

- Problem: Große Datenobjekte ("High-volume data objects")
- Lösung 1: Schnittstelle mit Attributen oder Zugriffs-Operationen
 - + Saubere OO-Abstraktion
 - + Freie Verteilungsmöglichkeiten
 - Hohe Netzlast für Datenzugriff
 - Skalierungsprobleme in manchen ORBs
- Lösung 2: Struktur mit Daten, lokale Verpackung in Objekten
 - + Lokaler Datenzugriff
 - Verletzte OO-Abstraktionen
 - Mehrfache, nicht synchronisierte Kopien
- Lösung 3: Value type
 - + Lokaler Datenzugriff und OO-Abstraktion
 - Mehrfach, nicht synchronisierte Kopien

19 Beispiel

- IDL-Beschreibung einer Bibliotheksverwaltung
 - ◆ neuen Leseausweis ausstellen
 - ◆ Buch suchen
 - ◆ Buch ausleihen / zurückgeben

```
module library {
    struct Date {
        unsigned short day, month, year;
    };

    exception DeniedCard { string reason; };
    exception NoSuchCard {};
    exception BookBorrowed { Card borrower; };
    exception BookNotBorrowed {};

    //.....
}
```

19 Beispiel (2)

```
//....
// forward declaration
interface Card;

interface Book {
    readonly attribute string      authors;
    readonly attribute string      title;
    readonly attribute string      publisher;
    readonly attribute unsigned short year;
    readonly attribute string      registration;
    readonly attribute boolean     isBorrowed;

    Card borrowedBy() raises( BookNotBorrowed );

    // internal use only
    void setBorrowed( in Card c ) raises( BookBorrowed );
    void unsetBorrowed() raises( BookNotBorrowed );
};
//....
```

19 Beispiel (3)

```
//...

typedef sequence<Book> Books;

interface Card {
    readonly attribute Date    expires;
    readonly attribute string  owner;
    readonly attribute Books   borrowedBooks;
    readonly attribute string  number;

    void borrowBook( in Book b ) raises( BookBorrowed );
    void returnBook( in Book b ) raises( BookNotBorrowed );
};

interface Librarian {
    Card issueCard( in string owner ) raises( DeniedCard );
    Card findCardByOwner( in string owner ) raises( NoSuchCard );
    Books findBookByTitle( in string title );
};
};
```

20 IDL-Zusammenfassung

- Beschreibung von Datentypen und Schnittstellen von CORBA-Objekten
- C++-ähnliche Syntax
- Primitive Datentypen (**short**, **long**, **boolean**, **char**, ...)
- Zusammengesetzte Datentypen (**struct**, **union**, **enum**)
- Arrays
- "Template types" (**sequence**, **string**, **fixed**)
- Objekt-**interface** mit Attributen und Operationen
- Fehlersignalisierung mit Exceptions
- Object-by-value durch **valuetypes**