

B 7. Übung

B.1 Überblick

- Automatische Erzeugung von Code in RPC-Systemen
 - ◆ Allgemeiner Überblick
 - ◆ Übungsaufgabe zur automatische Codegeneration
- IDLflex als generisches Tool zur Codeerzeugung
 - ◆ Grundlagen: XML, CORBA IDL
 - ◆ Aufbau, interne Repräsentation von Schnittstellen
 - ◆ XML-basierte Beschreibung der Codegenerierung
 - ◆ Beispiele
- Aufgabe 5
 - ◆ Erweitertes Marshalling
 - ◆ Automatische Generierung von Stub und Skeleton

B.2 Codeerzeugung in RPC-Systemen

- Grundprinzip des Fernaufrufs
 - ◆ Umsetzung von Prozeduraufruf in Nachrichtenaustausch
 - ◆ Dabei Abstraktion der Örtlichkeit von Auftraggeber und -nehmer
 - ◆ Erfordert Ver-/Entpacken von Parametern/Rückgabewerten in Nachrichten
- Prozedurstümpfe
 - ◆ Client-Stub und Server-Stub (bzw. -Skeleton) kapseln obige Aufgaben
 - ◆ Bisherige Übungsaufgaben (Aufgabe 4):
Manuelle Implementierung der Stubs
 - ◆ Ziel in RPC-Systemen: Stümpfe möglichst automatisch erzeugen!

B.2 Codeerzeugung in RPC-Systemen

- Beispiel zu Aufgabe 4: Client-Stub:

```
int16_t MultiplyServerStub::multiply(int16_t val1,
                                   int16_t val2)
{
    ...
    Request req(...)
    req.write((int8_t) 0); // Funktions-ID
    req.write(val1);
    req.write(val2);
    comm->send(addr, req.get_buffer());

    Address source;
    int16_t result;
    comm->receive(&source, &buffer);
    Response res(&buffer);
    res.read(result);
    return result;
}
```

B.2 Codeerzeugung in RPC-Systemen

- Beispiel zu Aufgabe 4: Server-Skeleton:

```
Response MultiplyServerSkeleton::process(Request &request)
{
    int8_t mid;
    if (!request.read(mid)) { /* Fehlerbehandlung? */ }
    switch (mid) {
        case 0 : {
            int16_t v1_16, v2_16, r_16;
            request.read(v1_16);
            request.read(v2_16);
            r_16 = obj->multiply(v1_16, v2_16);
            Response response(...);
            response.write(r_16);
            return response;
        }
        ...
    }
}
```

B.2 Codeerzeugung in RPC-Systemen

- Automatisierung der Codeerzeugung: Stub- und Skeleton in Abhängigkeit von Funktionssignaturen (Parameter und Rückgabewerte)
 - ◆ Problem der lokalen Gültigkeit von Parametern
 - Speicheradressen (Zeiger) i.d.R. nur lokal gültig
 - Ebenso andere Ressourcen wie z.B. Datei-Handle, Dateinamen, etc.
 - ◆ Auslegungsproblem der Parameter
 - Übertragung von Referenzparameter als Kopie der referenzierten Daten
 - Eingabe- oder Ausgabeparameter?
(call by value, by result, by value/result)
 - Interpretation von Zeiger: Was sind die referenzierten Daten
char *: byte / byte-Array fester Länge / \x00-terminierter String
void *: ????
 - Übertragung als entfernte Referenzen: "Call back"-Mechanismus

rs - Übung

Übungen zu "Verteilte Systeme"

R 5

B.2 Codeerzeugung in RPC-Systemen

- Verschiedene Lösungen in realen Systemen
 - ◆ SUN RPC
 - Explizite Schnittstellen-Beschreibungssprache
 - Eingabeparameter "by value", Rückgabewert "by value"
 - ◆ Java RMI
 - Auslegung von Parameter direkt in der Programmiersprache beschreibbar
 - Eingabeparameter "by value", falls Basisklasse "Serializable";
Eingabeparameter "by (object) referenze", falls Basiskl. "RemoteObject"
 - ◆ CORBA
 - Explizite Schnittstellen-Beschreibungssprache; Explizite Spezifikation von "by value" (in), "by result" (out) oder "by value/result" (in/out)
 - Parameter vom Typ "interface" als entfernte Referenzen
 - ◆ Microsoft .NET
 - Schnittstelle in Bytecode beschrieben; Auslegung der Parameter durch Basisklassen (MarshalByRefObject / MarshalByValueObject)

vs - Übung

Übungen zu "Verteilte Systeme"
© Universität Erlangen-Nürnberg • Informatik 4, 2004

IDLflex.fm 2004-06-08 09:57

B.6

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

B.2 Codeerzeugung in RPC-Systemen

- Übungsaufgabe 5
 - ◆ Erstellung eines Code-Generators für das eigene RPC-System
 - ◆ Grundsätzlich: Komplexe Aufgabe
 - Definition einer Schnittstellen-Beschreibungssprache
 - Implementierung eines Parsers für diese Sprache
 - Erzeugung von Code aus dem Parser-Baum
 - ◆ Zur Vereinfachung: Tool "IDLflex" wird bereitgestellt und kann verwendet werden
 - Parser für CORBA IDL erzeugt objektorientierte Repräsentation
 - Zu erzeugender Code kann in einer simplen XML-basierten Sprache spezifiziert werden

rs - Übung

Übungen zu "Verteilte Systeme"

R 7

B.3 Einführung in XML (zum Nachlesen)

1 Überblick

- Aufbau eines XML-Dokuments
 - ◆ Prolog, DTD, Daten
- XML-Daten: Elemente, Attribute, Inhalte
- DTD: Document Type Definition

vs - Übung

Übungen zu "Verteilte Systeme"
© Universität Erlangen-Nürnberg • Informatik 4, 2004

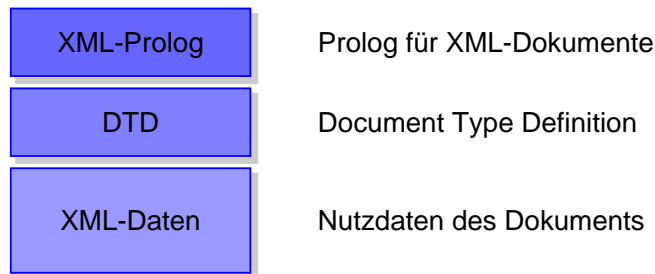
IDLflex.fm 2004-06-08 09:57

B.8

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

2 Aufbau eines XML-Dokumentes

- Jedes XML-Dokument ist aus drei Teilen aufgebaut:



- Achtung:
 - ◆ XML ist „case-sensitive“

3 Der XML-Prolog

- Festlegung, dass es sich um ein XML-Dokument handelt:

```
<?xml version="1.0" ... ?>
```

- Festlegung des verwendeten Zeichensatzes, z.B.:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

- ◆ Beispiele für Zeichensätze:

ISO-NORM	Zeichensatz
UTF-8, UTF-16	Internationale Zeichensätze
ISO-8859-1	Westeuropa (Latin-1)
ISO-8859-2	Osteuropa (Latin-2)
ISO-8859-3	Südeuropa (Latin-3)

4 XML-Daten (1): Elemente

- Haben eindeutigen Elementnamen;
Notation durch *Start-Tag* und *Ende-Tag* beschrieben

```
<MeinName>  
</MeinName>
```

- Können (optional) sowohl Attribute als auch Inhalte haben
- Optimale Kurznotation, falls Element keinen Inhalt hat

```
<MeinName />
```

4 XML-Daten (2): Attribute

- Bestehen Attributname und Wert
- Werden innerhalb des Start-Tags angegeben

```
<MeinName ID="4711" LastUpdate="2004-01-30">  
</MeinName>
```

4 XML-Daten (3): Inhalte

- Werden zwischen Start-Tag und Ende-Tag angegeben
- Können aus Text, weiteren XML-Elementen oder speziellen Elementtypen bestehen

```
<MeinName ID="4711" LastUpdate="2004-01-30">
  <Vorname>Xaver</Vorname>
  <Nachname>Niederhösl</Nachname>
  <Titel>Prof. Dr.</Titel>
</MeinName>
```

5 Die Document Type Definition (DTD)

- Legt die Struktur des XML-Dokumentes fest, d.h.
 - ◆ Die erlaubten Elemente
 - ◆ Die erlaubten Attribute eines Elements, incl. Default-Werte
 - ◆ Die erlaubte Schachtelung der Elemente
- DTD kann innerhalb des Dokuments spezifiziert werden:


```
<!DOCTYPE IDLflex [ .... ]>
```

 - ◆ Muss vor den Daten der XML-Datei stehen
- Verweis auf externe DTD-Datei:


```
<!DOCTYPE IDLflex SYSTEM "Mapping.dtd"> oder
<!DOCTYPE IDLflex SYSTEM "http://www.myweb.de/mydtd.dtd" >
```

 - ◆ Datei wird vom spezifizierten Ort geladen
 - ◆ Essentiell bei Mehrfachverwendung (Konsistenz!)
- Mischung beider Varianten möglich
 - ◆ interne Variante überschreibt externe

5 Spezifikation einer DTD

- Bestandteile einer DTD:
 - ◆ Kommentartexte
 - ◆ Definition von Entities
 - ◆ Definition von Elementen
 - ◆ Definition der Attributlisten von Elementen
- Kommentare in XML-Dateien:

```
<!-- Hier kann beliebiger Kommentartext stehen -->
```

5 Definition von Entities

- Allgemeine Entities dienen der Festlegung von Abkürzungen
- Bei Mehrfachdefinition derselben Entity ist die erste Definition bindend
- Beispiel:


```
<!ENTITY MFG "Mit freundlichen Grüßen" >
```

 - ◆ Definiert eine Abkürzung für die angegebene Zeichenkette
 - ◆ Kann im Anschluss in den XML-Daten verwendet werden:


```
<p>&MFG;</p>
```
 - ◆ Referenzierung jeweils mittels: `&Entity-Name;`
 - ◆ Vordefinierte, allgemeine Entities: `<`; `>`; `&`; ...

5 Definition von Elementen

- Festlegung, welche Elemente bekannt sind, z.B.:

```
<!ELEMENT Component ...>
```

- Elementname:

- ◆ Eindeutiger Bezeichner für ein Element
- ◆ Muss mit einem Buchstaben beginnen
- ◆ Sonstige erlaubte Zeichen: Ziffern, „.“, „-“, „_“, „:“
- ◆ Verwendung des Präfix "xml" vermeiden!

- Anschließende Verwendung des Elements:

```
<Component> ... </Component>
```

- Achtung:

- ◆ Elemente müssen immer abgeschlossen werden!

5 Definition von Elementen

- Elemente können leer sein

- Spezifikation eines leeren Elements:

```
<!ELEMENT img EMPTY >
```

- ◆ Dienen einfachen Auszeichnungen, die keine Elemente oder Text enthalten
- ◆ Können selber aber Attribute enthalten
- ◆ Verwendung z.B. beim Image-Element in HTML:

```
</img>
```

- ◆ Leere Elemente können direkt abgeschlossen werden:

```

```

5 Definition von Elementen

- Elemente können Text klammern

- Spezifikation, dass ein Element beliebigen Inhalt enthält:

```
<!ELEMENT Whatever (#PCDATA) >
```

- ◆ Inhalt wird nicht weiter betrachtet
- ◆ Keine Festlegung der Struktur

- Verwendung:

```
<Whatever>20.10.2000</Whatever>
```

```
<Whatever></Whatever>
```

```
<Whatever />
```

```
<Whatever>
  <Name>Hans Meier</Name>
</Whatever>
```

5 Definition von Elementen

- Elemente können andere Elemente klammern

- Einzelnes Subelemente:

```
<!ELEMENT MusicArchive (CD) >
```

- ◆ Nur genau ein derartiges Element darf enthalten sein

- Liste von Subelementen:

```
<!ELEMENT Name (Vorname, Nachname) >
```

- ◆ **Nur die** angegebenen Elemente **dürfen** enthalten sein
- ◆ **Alle** angegebenen Elemente **müssen** enthalten sein
- ◆ Die **Reihenfolge** ist durch die Element-Definition **festgelegt**

- Optionale Liste von Subelementen:

```
<!ELEMENT Mitarbeiter (Angestellter | Arbeiter) >
```

- ◆ **Nur eines** der angegebenen Elemente **darf** enthalten sein

5 Definition von Elementen

■ Spezifikation von Multiplizitäten:

- ◆ ? optional
- ◆ + mindestens einmal
- ◆ * beliebig oft (auch 0 mal)

■ Beispiele komplexer Spezifikationen:

- ◆ Name mit optionalem Titel, mindestens einem Vornamen und genau einem Nachnamen

```
<!ELEMENT Name (Titel?, Vorname+, Nachname) >
```

- ◆ Mitarbeiterliste bestehend aus Arbeitern und Angestellten

```
<!ELEMENT MitarbeiterListe (Arbeiter | Angestellter)* >
```

- ◆ Mitarbeiterliste bestehend aus Arbeitern oder Angestellten

```
<!ELEMENT MitarbeiterListe (Arbeiter* | Angestellter* ) >
```

5 Attributlisten von Elementen

■ Jedes Element kann eine Menge von Attributen besitzen, z.B.:

```

```

- ◆ Attribute dienen der Parametrierung von Elementen
- ◆ Attribute sind selber keine Elemente!

■ Attribute bestehen aus:

- ◆ Einem im Element eindeutigen Namen
- ◆ Einem in Anführungszeichen eingeschlossenen Wert

■ Definition von Attributlisten in der DTD:

```
<!ATTLIST img src CDATA #REQUIRED
border (0|1) "1"
... >
```

- ◆ Attribute bestehend aus:
 - Name des Attributs, Wertebereich des Attributs, Wert-Beschreibung

5 Definition von Attributlisten in der DTD

■ Typ des Attributes:

- ◆ Legt fest, welche Werte ein Attribut annehmen darf
- ◆ Drei Klassen werden unterschieden:
 - Beliebige Zeichenketten (CDATA)
 - Aufzählungen
 - Spezielle Typen (ID, IDREF, IDREFS, ENTITY, ...)

■ Möglichkeiten für die Werte-Beschreibung:

- ◆ Standardwert festlegen: "1"
- ◆ Festlegung, dass das Attribut immer angegeben werden muss: #REQUIRED
- ◆ Festlegung, dass kein Standardwert existiert: #IMPLIED
- ◆ Fixierung eines Wertes den ein Attribut annehmen darf: #FIXED "yes"

5 Definition von Attributlisten in der DTD

■ Beispiele:

- ◆ Definition eines Attributes mit beliebigem Inhalt:

```
<!ATTLIST img src CDATA #REQUIRED />
```

- ◆ Definition eines Attributes mit einer Aufzählung:

```
<!ATTLIST img border (0 | 1) "1"/>
```

- ◆ Definition von Attributen mit speziellen Typen:

```
<!ATTLIST Mitarbeiter PersonalNumber ID #REQUIRED
Vorgesetzter IDREF #IMPLIED />
```

■ Verwendung:

```

<img border="0" ... />
<Mitarbeiter PersonalNumber="007">... </Mitarbeiter>
<Mitarbeiter PersonalNumber="0815" Vorgesetzter="007"> ...
```

5 Definition von Attributlisten

- Wann verwendet man Attribute?
 - ◆ Wenn es sich um kurze, einfache Inhalte handelt
 - ◆ Wenn man den Inhalt auf einige, festgelegte Möglichkeiten beschränken will
 - ◆ Wenn der Inhalt nur das Element parametrisiert
 - ◆ Wenn der Inhalt eher interner, technischer Natur ist (z.B. die ID)
- Wann verwendet man Elemente?
 - ◆ Wenn unterschiedliche Inhalte unter derselben Bezeichnung hinterlegt werden sollen
 - ◆ Wenn ein Element Substrukturen besitzen soll (Container-Prinzip)
 - ◆ Wenn der Inhalt typischerweise über mehrere Zeilen geht

6 XML Schema

- XML-basierte Alternative zur DTD
 - ◆ Offizieller W3C-Standard seit Mai 2001
- Vorteile
 - ◆ Schema-Definition erfolgt in normaler XML-Notation, dadurch einheitlicher und einfacher zu parsen
 - ◆ Größere semantische Ausdrucksfähigkeiten, Datentypen lassen sich besser beschreiben
- Kurzes Beispiel

```
?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"...>
  <xs:element name="MeinName">
    <xs:complexType><xs:sequence>
      <xs:element name="Vorname" type="xs:string"/>
      <xs:element name="Nachname" type="xs:string"/>
      <xs:element name="Titel" type="xs:string"/>
    </xs:sequence></xs:complexType>
  </xs:element>
</xs:schema>
```

7 Zusammenfassung

- XML-Dokument besteht aus Header, DTD, und dem eigentlichen Dokument
- Im Dokument gibt es
 - ◆ Elemente (immer abgeschlossen!)
 - ◆ Attribute (mögliche Namen in DTD definiert)
 - ◆ Inhalt (beliebiger Text oder weitere Elemente)

B.4 Kurzbeschreibung von CORBA IDL

1 Grundlegendes

- IDL dient der Beschreibung von Datentypen und Schnittstellen
- Unabhängig von einer bestimmten Programmiersprache
- Syntax stark angelehnt an C++
 - ◆ Beschreibung von Datentypen und Schnittstellen
 - ◆ Keine steuernden Anweisungen
 - ◆ Präprozessor wie in C++
 - #include
 - #define
 - Kommentare mit // und /* .. */)

2 Bezeichner

- Alle 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
- **Sobald ein Bezeichner benutzt ist, sind alle anderen Varianten mit anderen Gross-/Kleinschreibung verboten!**
 - ◆ Sinn: Erlaubte Abbildung von IDL zu Sprachen, die nicht "case-sensitive" sind; erhalte Schreibweise von Bezeichner für "case-sensitive" Sprachen

- Beispiel:

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

3 Namensräume in CORBA IDL

- 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;
};
```

3 Primitive Datentypen

- Ganzzahlen
 - ◆ {,unsigned} short $-2^{15} \dots 2^{15}-1 / 0 \dots 2^{16}-1$
 - ◆ {,unsigned} long $-2^{31} \dots 2^{31}-1 / 0 \dots 2^{32}-1$
 - ◆ {,unsigned} long long $-2^{63} \dots 2^{63}-1 / 0 \dots 2^{64}-1$
- Fließkommazahlen (ANSI/IEEE Std 754-1985)
 - ◆ float einfache Genauigkeit
 - ◆ double doppelte Genauigkeit
 - ◆ long double erweiterte Genauigkeit (mindestens 15 Bit Exponent und 64 Bit Basis)
- Zeichen
 - ◆ char ISO 8859-1 (Latin1) Zeichen
 - ◆ wchar multi-byte character (Unicode)
 - ◆ Lokale Repräsentation kann implementierungsabhängig sein

3 Primitive Datentypen

- boolean
 - ◆ Nur die Werte TRUE und FALSE
- octet
 - ◆ Länge 8 bit, Keine Konvertierung bei der Übertragung
- void

4 Datentyp-Deklarationen

- Alias für einen existierenden Datentyp
- Syntax:

```
typedef existing_type alias;
```

- Beispiel:

```
typedef long IDNumber;
```


5 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 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!

7 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!

8 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;
```

9 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 = 0x00FF00;
```

10 Schnittstellen (Interfaces)

- Sichtbare Schnittstelle von Objekten

- Kann enthalten:

- ◆ Operationen
- ◆ *Attribute*
- ◆ *Lokale Typen, Konstanten, Exceptions*

- Syntax:

```
interface name {
    Deklaration von Attributen und Operationen (sowie Typen und Exceptions)
};
```

- Schnittstellen definieren ebenfalls einen eigenen Namensraum

10 Schnittstellen – Operationen

- Methoden von CORBA-Objekten mit:

- ◆ Methoden-Name
- ◆ Rückgabe-Datentyp
- ◆ Aufruf-Parameter
- ◆ (*Exceptions*)

- Syntax:

```
return_type name( parameter_list ) raises( exception_list );
```

- Nur der Methodenname ist signifikant

- ◆ **Kein Overloading durch Parametertypen**

10 Schnittstellen – Parameterübertragung

- Für jeden Parameter muss die Übertragungsrichtung angegeben werden:

- ◆ *in* nur vom Auftraggeber zum Auftragnehmer
- ◆ *out* nur vom Auftragnehmer zum Auftraggeber
- ◆ *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 );
};
```

11 Vorwärtsdeklarationen

- Problem: Zirkuläre Abhängigkeiten in den Deklarationen
 - ◆ 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
 - ◆ Deklare einen Bezeichner für einen Typ, aber nicht den Typ selbst
- Beispiel:

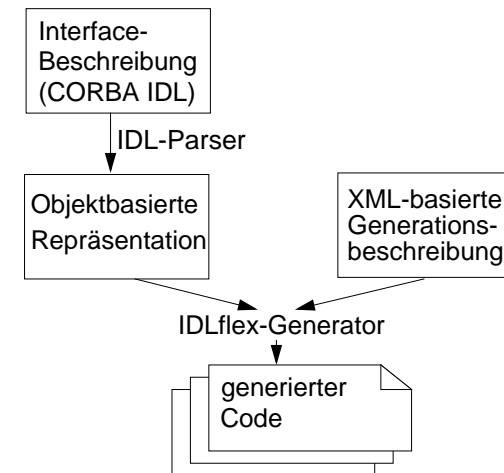
```
interface B;           // Forwärtsdeklaration
interface A {
    B get_b();
};
interface B {
    A get_a();
};
```

12 Zusammenfassung CORBA IDL

- Umfangreiche Beschreibungssprache für Datentypen und Schnittstellen
 - ◆ Genauere Spezifikation möglich als in C++
 - Arrays, Sequences, Strings; mit/ohne Längenbeschränkung
 - in/out/inout
- Für VS-Übung: Verwendung nur von einem Teil der Möglichkeiten
 - ◆ Im wesentlichen nur Interface-Deklarationen und ein Teil der Datentypen
- Vertiefte Behandlung von CORBA in der Vorlesung "Middleware" im Wintersemester

B.5 IDLflex

1 Grundstruktur



2 IDL Objektrepräsentation

- IDL-Datei wird intern als Objekt-Baum repräsentiert
- Basisklasse IDLObject

IDLObject
<pre> getName(String spec): String getAttribute(String spec): boolean getContent(String spec): IDLObject getContentList(String spec): IDLObject[] is_a(String type): boolean </pre>

2 IDL Objektrepräsentation

■ Beispiel

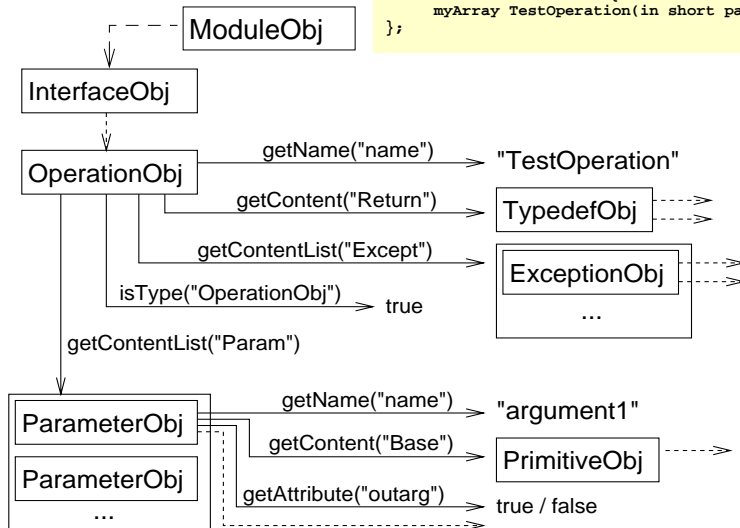
```
typedef short myArray[10];

interface TestInterface {
    myArray TestOperation(in short param1, ...);
    ...
};
```

2 IDL Objektrepräsentation

■ Beispiel

```
interface TestInterface {
    myArray TestOperation(in short param1, ...);
};
```



2 IDL Objektrepräsentation

■ Abgeleitete Klassen und deren Zusammenhang

- ◆ Bei alle Klassen liefert getName("name") den IDL-Namen
- ◆ Alle Klassen, die einen Typ definieren, sind von TypedefObj abgeleitet
StructObj, UnionObj, EnumObj, AliasObj
- ◆ IDL-Modul (module) => ModuleObj
getContentList("MEMBER"):
{ModuleObj, ConstantObj, TypedefObj, ExceptionObj, InterfaceObj}*
- ◆ IDL-Konstante (const) => ConstantObj
getContent("BASE"):
{PrimitiveObj, TypedefObj}

2 IDL Objektrepräsentation

◆ IDL-Schnittstellen (interface) => InterfaceObj

```
getContentList("MEMBER"):  
{OperationObj, AttributeObj, ConstantObj,  
ExceptionObj, TypedefObj}*
```

◆ IDL-Methodendeklarationen => OperationObj

```
getContent("RETURN"):  
{PrimitiveObj, TypedefObj, InterfaceObj}  
getContentList("PARAM"):  
{ParameterObj}*  
getContentList("EXCEPT"):  
{ExceptionObj}*
```

◆ Parameterdeklaration => ParameterObj

```
getAttribute("{in,out,inout}arg")  
getContent("BASE"):  
{PrimitiveObj, TypedefObj, InterfaceObj}
```

3 XML Mapping-Beschreibung

■ Struktur des XML-Dokuments zur Mapping-Beschreibung

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE IDLflex SYSTEM "Mapping.dtd">

<IDLflex ROOT="RootComponent" UTILITY="FAXUtility"
  WRITER="JavaFileWriter">

  <COMPONENT NAME="RootComponent">
    ...
  </COMPONENT>

  <COMPONENT NAME="Component2">
    ...
  </COMPONENT>
</IDLflex>
```

- IDL-Beschreibung wird ausgehend von einer Root-Komponente abgearbeitet. Es gibt stets eine implizite Referenz auf ein Element der Objektrepräsentation der IDL

3 XML Mapping-Beschreibung

■ Komponenten-Beschreibung (Beispiel RootComponent)

```
<COMPONENT NAME="RootComponent">
  <ITERATE NAME="MEMBER">
    <SWITCH>
      <CASE TYPE="ConstantObj">
        <CALL NAME="ConstantGenerator"/> </CASE>
      <CASE TYPE="ModuleObj">
        <CALL NAME="RootComponent"/> </CASE>
      <CASE TYPE="InterfaceObj">
        <CALL NAME="InterfaceGenerator"/></CASE>
      ...
    <DEFAULT>
      <ERROR>Illegal member in IDL module</ERROR>
    </DEFAULT>
    </SWITCH>
  </ITERATE>
</COMPONENT>
```

3 XML Mapping-Beschreibung

■ Komponenten-Beschreibung (2)

◆ Input (CORBA IDL)

```
interface test {
    short test(in short a, in short b);
    long test(in long a, in long b);
};
```

◆ Output (C++)

```
class test {
public:
    virtual int16_t test( const int16_t value0,
                        const int16_t value1 ) = 0;
    virtual int32_t test( const int32_t value0,
                        const int32_t value1 ) = 0;
};
```

3 XML Mapping-Beschreibung

■ Komponenten-Beschreibung (3)

```
<COMPONENT NAME="SimpleInterface">
  <FILE SPEC="header">
    class <GET T="IDL:name"/> {
    public:
      <ITERATE NAME="MEMBER">
        <IF TYPE="OperationObj">
          virtual <CALL OBJ="RETURN" NAME="TypeMapper"/>
            <GET T="IDL:name"/> (
              <ITERATE NAME="PARAM">
                <IF COND="!LOOP:First">, </IF>
                const <CALL OBJ="BASE" NAME="TypeMapper"/>
                  value<GET T="LOOP:Index"/>
              </ITERATE>
            ) = 0;
          </IF>
        </ITERATE>
      </COMPONENT>
```

4 Verwenden von IDLflex

- Verwenden (im CIP-Pool, am LS4) mit
/local/idlflex/bin/idlflex -m <XMI-Mapping-Datei> <IDL-Datei>
 - Immer '-m ...' verwenden, sonst wird Standard-CORBA-Java-Mapping verwendet
- Am eigenem PC verwenden: /local/idlflex/idlflex-dist.tgz kopieren, entpacken, bin/idlflex anpassen (Shellskript)
- Weitere Dokumentation findet sich in /local/idlflex/doc
- Beispiel der vorherigen Seite, erweitert um "Call-by-Value-Result", findet sich in /local/idlflex/xml/mapping/FAX/sample.xml
- Bei Problemen: Mail an <{reiser,felser}@cs.fau.de>

B.6 Aufgabe 5

- Bisher in Aufgabe 4:
 - ◆ Marshalling für primitive Datentypen (char, short, int, long, float, double)
 - ◆ Manuelle Implementierung von Stubs für Client und Server
- Neu in Aufgabe 5:
 - ◆ Erweiterung des Marshallings
 - Unterstützung von Arrays
 - "InOut"-Parameter: Call-by-value/result
 - "Out"-Parameter: Mehrere Parameter von Server zu Klienten übertragen
 - ◆ Automatische Generierung von Stub und Skeleton aus IDL-Beschreibung der Server-Schnittstelle
 - Generisches Tool *IDLflex* als Basis