F 7. Übung

F.1 Überblick

- Fragen/Probleme zu Aufgabe 3
- Grundlagen
  - ◆ Einführung in XML
  - ◆ CORBA IDL
- IDLflex als generisches Tool zur Codegenerierung
  - ◆ Aufbau, interne Repräsentation von Schnittstellen
  - ◆ XML-basierte Beschreibung der Codegenerierung
  - ◆ Beispiele
- Aufgabe 4
  - ◆ Erweitertes Marshalling
  - ◆ Automatische Generierung von Stub und Skeleton

F.2 Aufgabe 3

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


F.2 Besprechung Aufgabe 3

- Gibt es Fragen/Probleme??

F.2 Aufgabe 3

- Beispiel 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;
            }
        ...
    }
}
```

F.3 Einführung in XML

1 Überblick

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

2 Aufbau eines XML-Dokumentes

- Jedes XML-Dokument ist aus drei Teilen aufgebaut:

XML-Prolog	Prolog für XML-Dokumente
DTD	Document Type Definition
XML-Daten	Nutzdaten des Dokuments

- 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 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

4 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: `< > & ...`

6 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!

6 Definition von Elementen

- Elemente können <u>leer</u> 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 src="picture.gif"></img>
```

  - Leere Elemente können direkt abgeschlossen werden:

```
<img src="picture.gif" />
```

6 Definition von Elementen

- Elemente können <u>Text</u> klammern
- Spezifikation, dass ein Element beliebigen Inhalt enthält:

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

  - Inhalt wird <u>nicht</u> weiter betrachtet
  - Keine Festlegung der Struktur
- Verwendung:

```
<WhatEver>20.10.2000</WhatEver>

<WhatEver></WhatEver>

<Whatever />

<WhatEver>
    <Name>Hans Meier</Name>
</WhatEver>
```


6 Definition von Elementen

- Elemente können <u>andere Elemente</u> 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

6 Definition von Elementen

- Spezifikation von Multiplizitäten:
  - `?` optional
  - `+` mindestens einmal
  - `*` belibig 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* ) >
```


7 Attributlisten von Elementen

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

```
<img src="picture.gif" />
```

  - 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 Atributs, Wert-Beschreibung

7 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"**

7 Definition von Attributlisten in der DTD

- Beispiele:
  - Definition eine 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 PersonalNummer ID #REQUIRED
                      Vorgesetzter IDREF #IMPLIED />
```

- Verwendung:

```
<img src="http://www.wherever.com/images/logo.gif" ... />

<img border="0" ... />

<Mitarbeiter   PersonalNummer="007">... </Mitarbeiter>
<Mitarbeiter   PersonalNummer="0815" Vorgesetzter="007"> ...
```


7 Definition von Attribut-Listen

- 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 parametriert
  - 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

8 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)

F.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
    - Kommantare mit // und /* .. */)

2 Bezeichner

- Alle sKombinationen 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 Beispiel1 { ... };
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 Beispiel1 {
    typedef long IDNumber;
};
module Beispiel2 {
    typedef Beispiel1::IDNumber MyID;     // typedef long MyID;
};
```


3 Primitive Datentypen

- Ganzzahlen
  - ◆ **{,unsigned} short** $-2^{15}...2^{15}-1$ / $0...2^{16}-1$
  - ◆ **{,unsigned} long** $-2^{31}...2^{31}-1$ / $0...2^{32}-1$
  - ◆ **{,unsigned} long long** $-2^{63}...2^{63}-1$ / $0...2^{64}-1$
- 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 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 );
};
```


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
  - Deklariere einen Bezeichner für einen Typ, aber nicht den Typ selbst
- Beispiel:

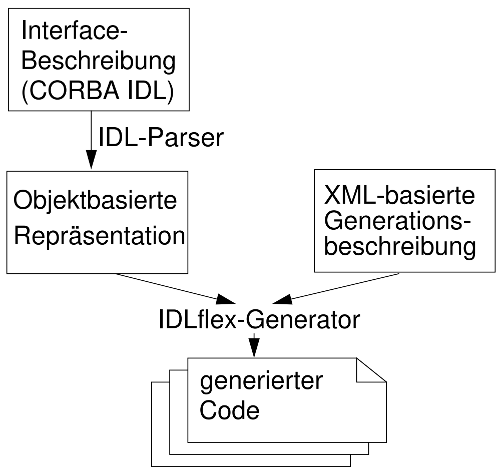
```
interface B;      // Forward declaration
interface A {
    B op_b();
};
interface B {
    A op_a();
};
```


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 "OOVS" im Wintersemester

F.5 IDLflex

1 Grundstruktur



2 IDL Objektrepräsentation

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

IDLObject
getName(String spec): String getAttribute(String spec): boolean getContent(String spec): IDLObject getContentList(String spec): IDLObject[] is_a(String type): boolean

2 IDL Objektrepräsentation

- Beispiel

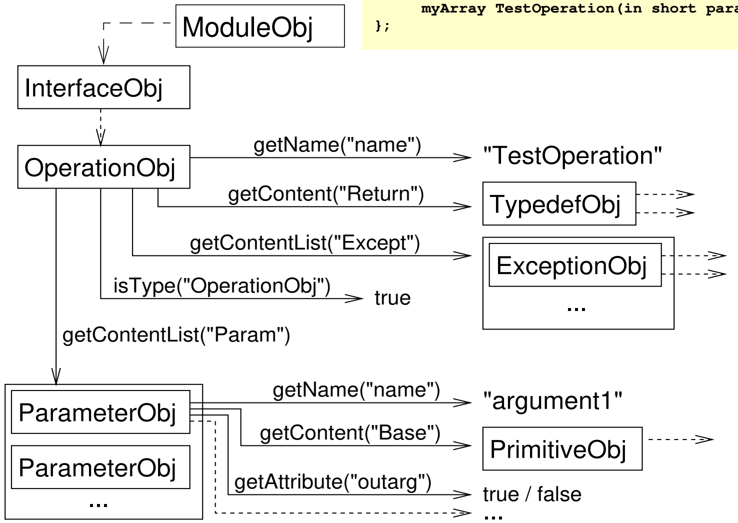
```
typedef short myArray[10];

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="FAXWriter">

    <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 <XMl-Mapping-Datei> <IDL-Datei>
  - Immer '-m ...' verwenden, sonst wird Standard-CORBA-Java-Mapping verwendet
- Auf 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>

5 Aufgabe 4

- Bisher in Aufgabe 3:
  - Marshalling für Primitive Datentypen (char, short, int, long, float, double)
  - Manuelle Implementierung von Stubs für Client und Server
- Neu in Aufgabe 4:
  - 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