

---

## Design von Schnittstellen mit CORBA-IDL

**Andres Koch**

El. Ing. HTL, M. Math

Object Engineering GmbH  
Birmensdorferstr. 32  
8142 Uitikon-Waldegg

E-Mail: [akoch@objeng.ch](mailto:akoch@objeng.ch)

## Einführung & Motivation

---



- “Programm to an interface, not an implementation” (Gamma et.al)
- Verteilte Applikationen benötigen Schnittstellen
- Oft wird über die CORBA-Technologie (u.a) mehr diskutiert, als über das Design von Schnittstellen
- Schnittstellen von CORBA COS (Common Object Services) sind komplex
- Das Design von Schnittstellen von Anwendungsapplikationen haben oft Mängel, welche sich erst mit der Zeit zeigen.
- CORBA-IDL anzuwenden und daraus Client und Serverseite zu implementieren ist “einfaches Handwerk”.
- Changemanagement gehört zur grossen Herausforderung für verteilte Systeme.
- Das Erfolgsrezept bei Schnittstellen liegt in der “Einfachheit“

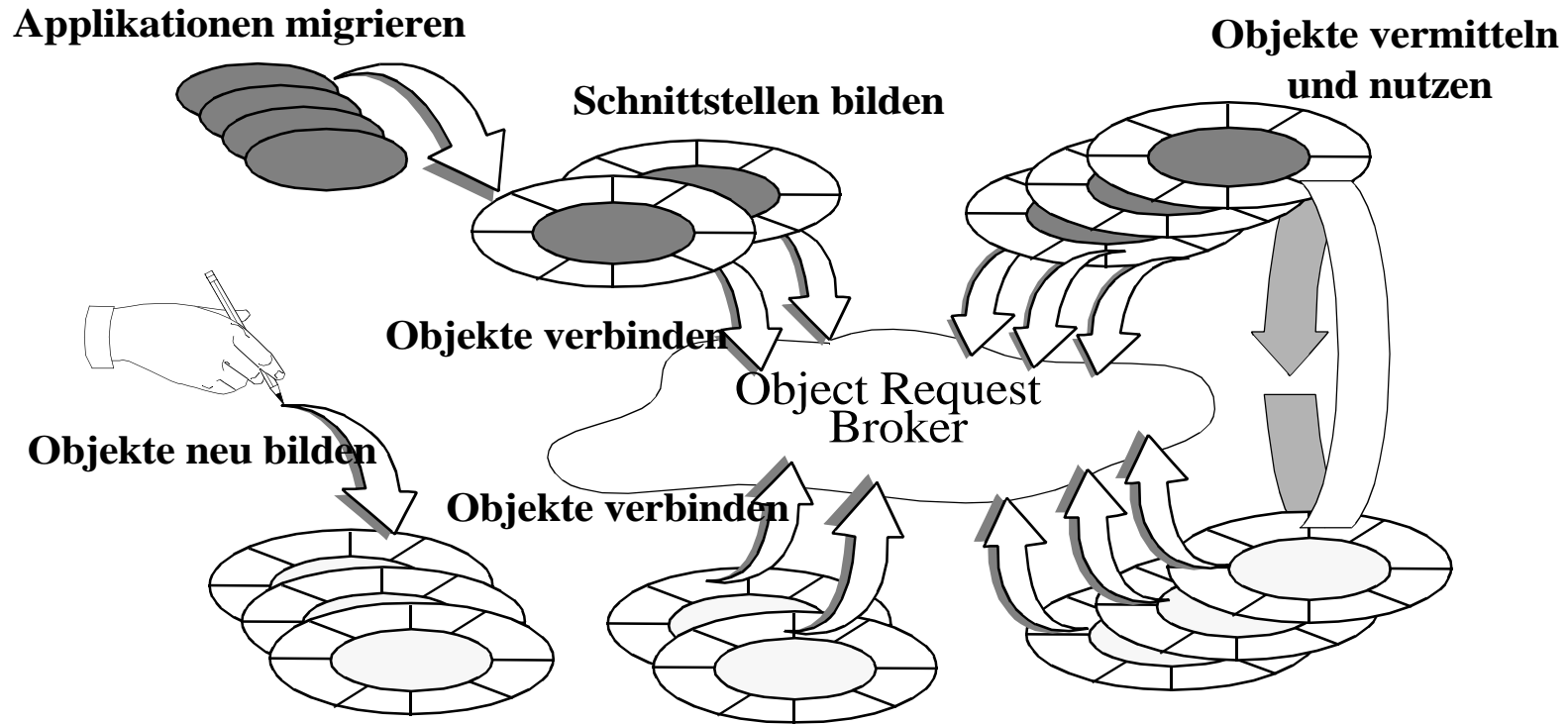
## Worauf man achten muss

---



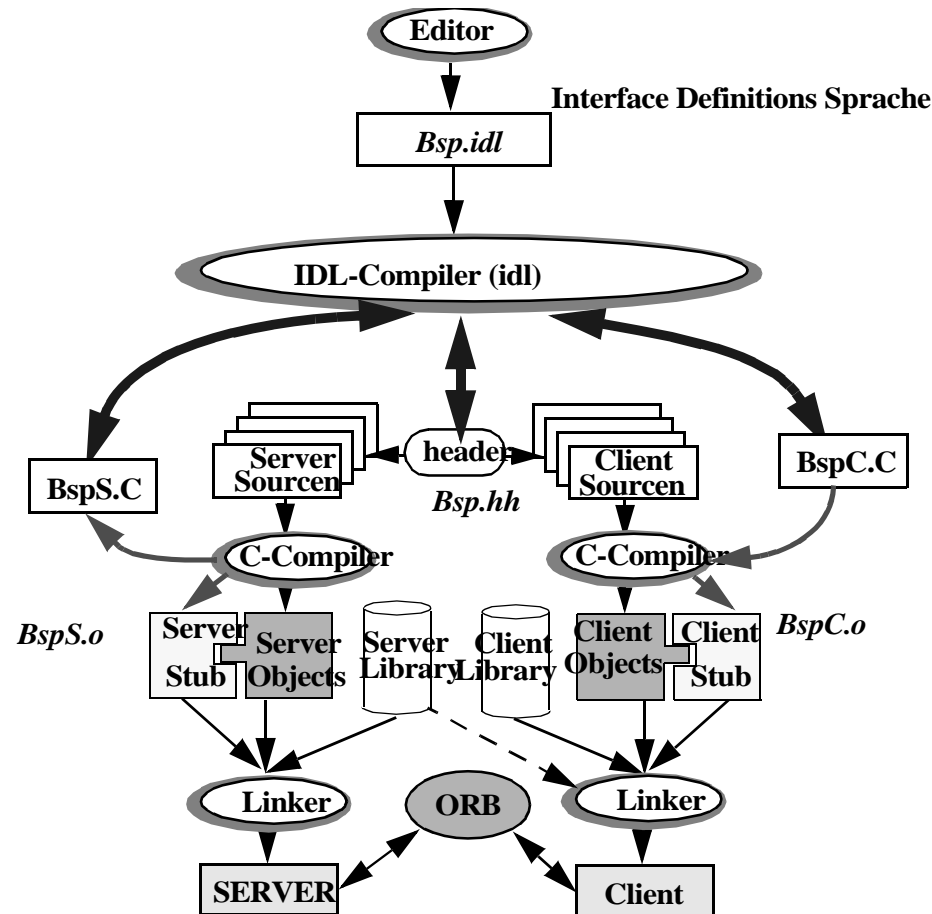
- IDL-Schnittstellen tendieren in der Regel zu komplex zu werden.
- Der Grad an Flexibilität ist umgekehrt proportional zu Performance
- Modernste Applikationen werden bereits nach kurzer Zeit zu “Legacy Systemen”
- Die Architektur bestimmt die relevanten Schnittstellen
- Wrapping von Altsystem-Applikationen ist eine wertvolle Methode, die zugehörige Schnittstelle ist eine Herausforderung für den Designer.
- Jede Schnittstellen-Änderung wirkt sich praktisch immer auf Client und Service aus.
- CORBA-IDL ist eine umfangreiche Sprache, sollte aber mit Bedacht angewendet werden (“Weniger ist oft mehr”).
- Schnittstellen welche die Implementation zeigen, werden zum Pferdefuss beim weiteren Ausbau und verletzen das Kapselungsprinzip.

# Typisches Wrapping zur Integration





# Von IDL zum Programm



```
float          // Floating Point
long           // integer long (32 Bit)
long long      // integer long (64 Bit)
short          // integer short (16 Bit)
unsigned long  // cardinal long (32 Bit)
unsigned short // cardinal short (16 Bit)
char           // 8-Bit Zeichentyp
boolean        // TRUE FALSE logischer Wert
octet          // 8-Bit ohne Konversion
any            // Irgend ein IDL-Typ
```

## CORBA IDL: Interface-Spezifikation

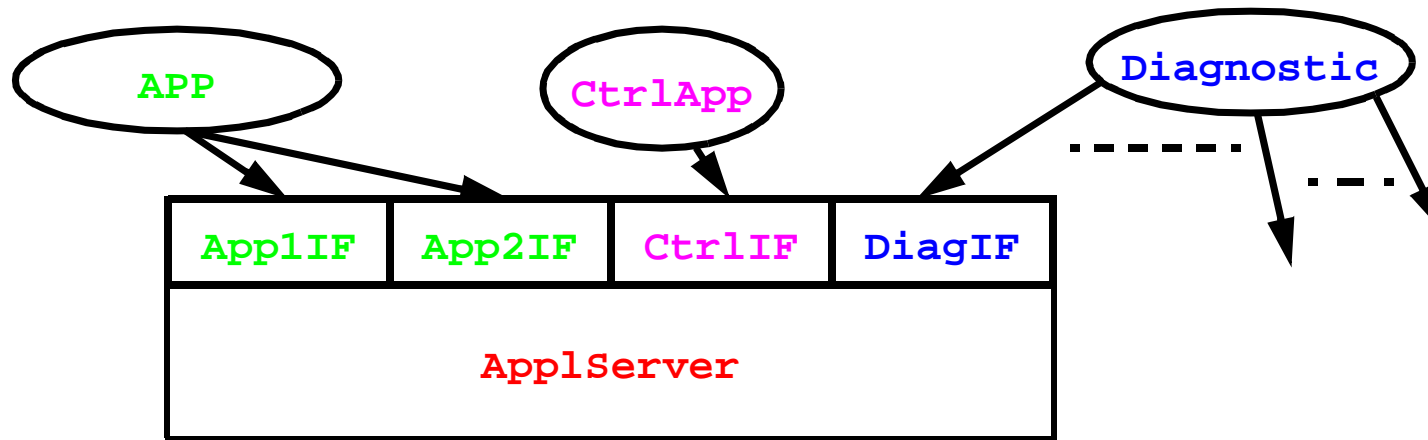


### Interface

```
interface Name
{
    // Schnittstellen Definition
};
```

### Interface mit Vererbung

```
interface Name_abgeleitet :
Name_VererbtesInterface ,
Name_VererbtesInterface
{
    // Schnittstellen Definition
};
```



```
interface AppServer :  
    App1IF , App2IF , DiagIF , CtrlIF  
{  
    ...  
}
```

```
module Modulename
{
    interface IfName1
    {..}
    interface IfName2
    {..}
    .....
    interface IfNamen
    {...}
}
```

- gruppiert diverse Spezifikationen als logische Einheit.
- sollte für jede logisch abgeschlossene Schnittstelle verwendet werden
- bildet einen eigenen Namensraum

## CORBA IDL: Attribute und Operationen



### Attribute Deklaration

```
interface IfName
{
    attribute Type memberName; // Read/Write
    readonly attribute string member2;
}
```

### Operations-Deklaration

```
interface IfName
{
    short initSystem (in float val1);
    void revert (out short rc, inout vec10 rec);
    oneway void start(in val1);
}
```

## Prinzipielles Vorgehen beim Entwurf

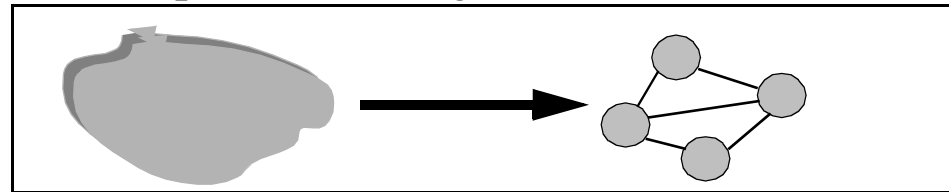
- Architektur definieren (vgl. Booch-Design)
- Aufteilung in sinnvolle Subsysteme und Module
- Definition der Schnittstellen der Module (OMG-IDL)
- Definition der Funktionalität der Server
- Definition der Client-Seite
- Definition der physikalischen Verteilung (Deployment)

## Realisation

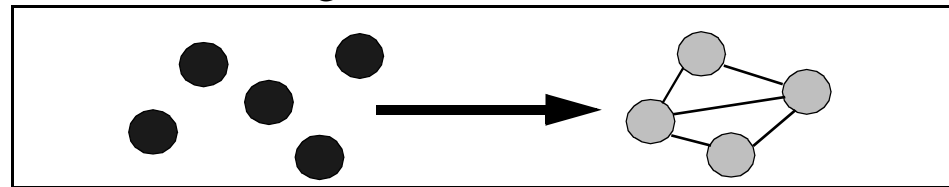
- Compilieren der Schnittstellen und verifizieren mit Stub-Prototypen.
- Implementieren der Schnittstellen-Teile
- Implementation der Server
- Integration der Server (damit für Client-Prog. verfügbar)
- Implementation der Clients

## Modularisierung: Meyer's Modularisierungsschritte (gemäss Bertrand Meyer):

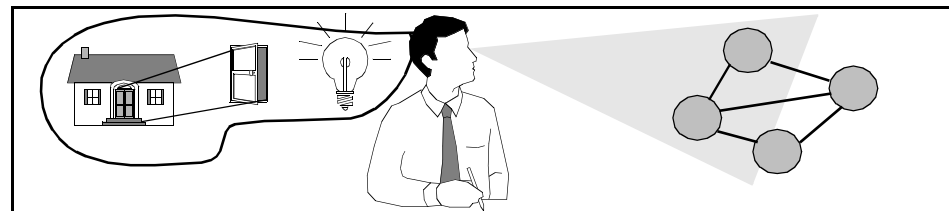
- Modulare Dekompositions-Fähigkeit



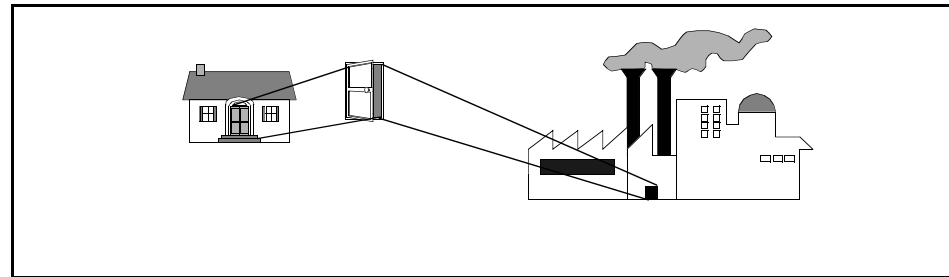
- Modulare Kombinierfähigkeit



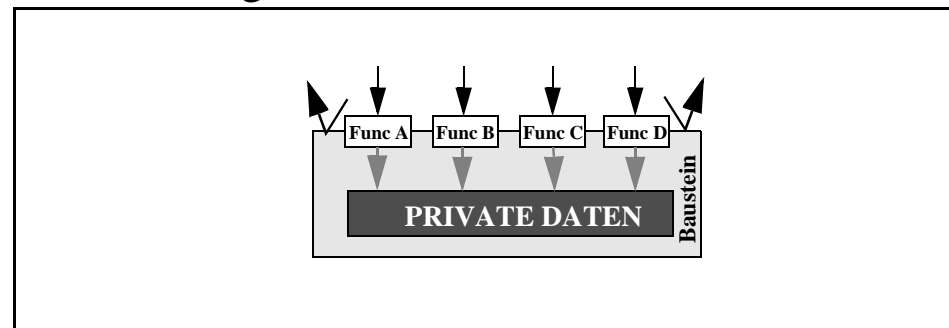
- Modulare Verständlichkeit



- Modulare Kontinuität



- Modulare Schutzfähigkeit



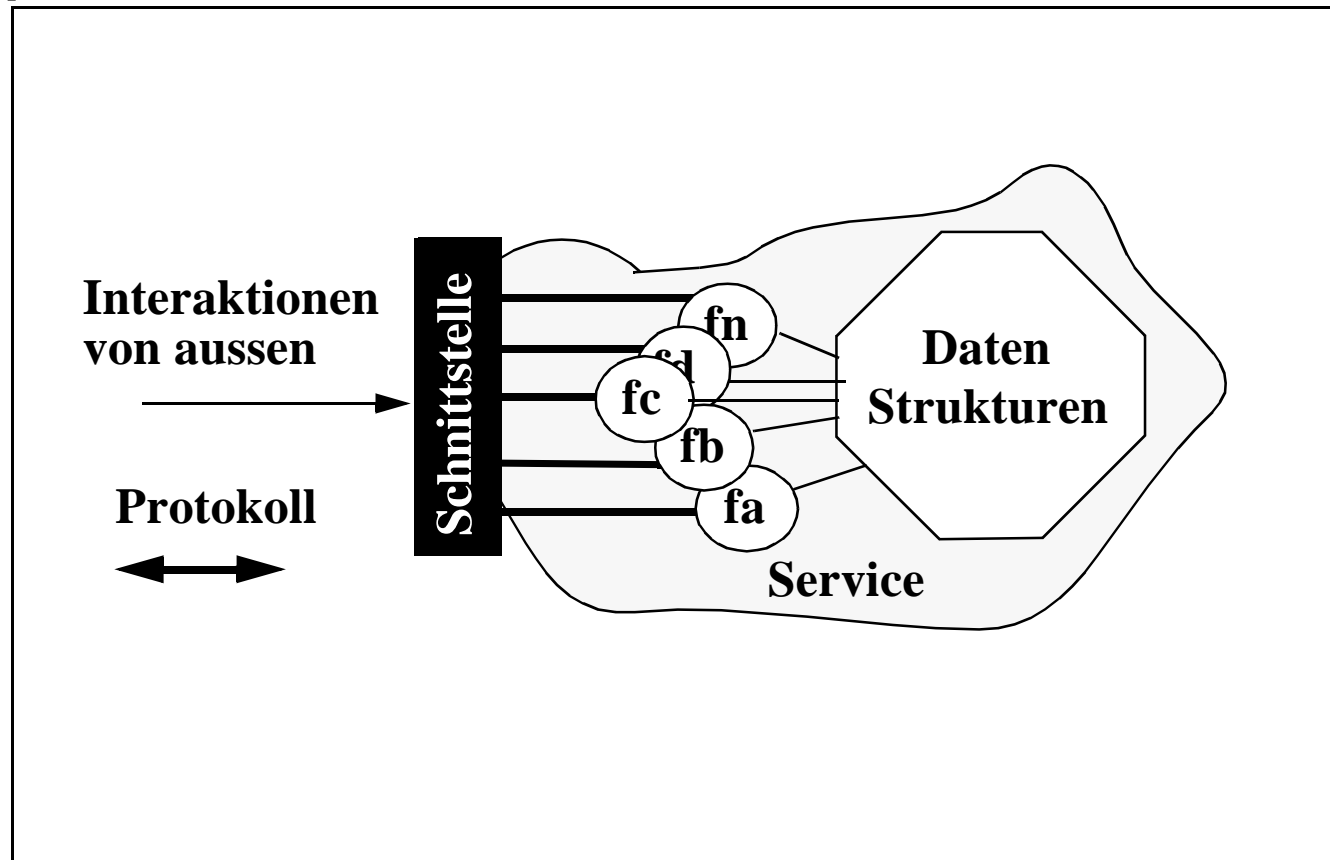
## Grundsätzlich

- Die Aufteilung muss vor allem zweckmässig sein, um eine Migration einer bestehenden System-Umgebung in eine flexiblere und leistungsfähigere zu überführen.
- Client/Server nicht nur der Mode willen, sondern als Strategie, um schrittweise und ohne Anwender-Aerger ein Re-Engineering in Modulen durchführen zu können.

### Bei der Aufteilung sollten berücksichtigt werden:

- Zweckmässigkeit (Mittel zum Zweck)
- Technische Realisierbarkeit
- Applikationsfunktionen und wo sie wie stark gebraucht werden
- Datenstrukturen (Datenbanken) und wo wieviele Datenzugriffe entstehen.
- Interkommunikations-Durchsatz-Menge (erwartet)
- Sicherheit (Datenschutz, Zugriffsschutz)

## Konzept der Schnittstelle



## Interface-Design: Schnittstellen-Definition



- Den Schnittstellen von Modulen sollte in der Entwurfsphase genügend Aufmerksamkeit und Sorgfalt gewidmet werden.
- Eine Schnittstelle während der Entwurfsphase ändern ist vertretbar, dies nach deren Inbetriebnahme zu tun ist fatal.
- Schnittstellen dürfen nur erweitert werden, dass eine Aufwärtskompatibilität sichergestellt ist.
- Meldungsformate entsprechen den Schnittstellen im meldungsorientierten Paradigma (Formate in XML)
- Prozedur-, Funktions- oder Methoden-Aufrufe eines Moduls resp. Objektes entsprechen den Schnittstellen im objektorientierten oder algorithmischen Paradigma (RPC).
- Die Abfolge der Benutzung der Schnittstellen entspricht dem Protokoll auf höherer Ebene (Applikations-Protokoll).
- Heute kann mit der CORBA-Interface Definition Language (CORBA-IDL) Schnittstellen definiert werden, die unabhängig von der im Service verwendeten Sprache ist.

## Wenige Schnittstellen

- Jedes Modul sollte mit möglichst wenigen andern Modulen kommunizieren.  
Beschränkte Gesamtzahl der Kommunikations-Kanäle zw. Modulen:
- ⇒ *Anzahl der Operationen (in interfaces) in Grenzen halten*

## Schmale Schnittstellen (Lose Kopplung)

- Wenn zwei Module überhaupt kommunizieren, sollten sie so wenig Information wie möglich austauschen.
- ⇒ *Anzahl der Parameter der Operationen beschränken ( $\leq 5$ )*

## die IDEaLe Schnittstelle

- Die Schnittstelle wird einmal definiert
- genau einmal
- und ändert während des ganzen Lebenszyklus des Systems nie mehr
  - ⇒ nicht realistisch und nicht voraussehbar

## Einfluss einer Schnittstellen-Veränderung

- Client und Server müssen beide mit den neuen IDL-Definitionen neu erstellt (generiert, konstruiert, “built”) werden ⇒ Build-Time Effect
- Entweder Client oder Server muss nicht neu erstellt werden ⇒ Runtime

## Level 1

- Kein Einfluss auf den Erstellungs-Prozess (Build)

## Level 2

- Re-Compilation und Re-Link notwendig

## Level 3

- Code muss verändert werden (Broken Code) ⇒ sollte nicht toleriert werden

## Regel:

Wenn immer Sie eine ausgebreitete IDL-Definition verändern, denken Sie an die Möglichkeit, dass irgendwo "da draussen" ein (oder viele) Clients mit einer älteren IDL existiert und operativ ist.

## IDL-Design: Was vermieden werden soll



- **Konstanten im IDL-File**
  - ⇒ bei Aenderungen ⇒ Level 2
- **Arrays ⇒ Performance**
  - ⇒ fixe Grösse ⇒ evtl. Level 2
- **Strings mit fixen Grössen**
  - (string<n>) ⇒ bei Aenderungen Level2
- **Bounded Sequences**
  - ⇒ bei Aenderungen ⇒ Level2
- **Enumerations**
  - ⇒ Level 2
- **Attribute**
  - ⇒ Bestes Beispiel für schlechte Performance
- **Applikatorische Schnittstellen mit Strukturen**
  - ⇒ können ändern

## IDL-Design: Flexibilität und Performance



### Performance

- Flexibilität und Performance schliessen sich fast gegenseitig aus
- Man braucht beides und muss optimieren
- Netzwerk-Calls müssen berücksichtigt werden für Performance
- Verarbeitungszeit muss berücksichtigt werden (Millionen von Elementen)
- Kardinalität in den Duzenden kann vernachlässigt werden, aber im Bereich von Tausenden oder Millionen wird ihr System “ruiniert”.

### Flexibilität

- Verwendung von NamedValue (any!)
- Unbounded sequences
- Schnittstellen Trennung nach Bereich
- Mischen zwischen schnellen und flexiblen Mechanismen
- **XML** kombiniert mit **CORBA** für die Zukunft als flexible Lösung

## Praxis-Beispiele (1):

## Flexibilität



```
typedef long      t_RetCode ;

exception IF_ABException
{
    long      fMinorCode ;
    string    fInfo ;
};

struct NVField
{
    string    fFldId ;
    string    fFldValue ;
};

typedef sequence<NVField> FieldRow ;
```

## Praxis-Beispiele (1):

## Flexibilität



```
union FieldValue switch (short)
{
    // Statt any
    case 1: string      fFldValue;
    case 2: FieldRow   fFldList;
};
```

```
struct AccessField
```

```
{
    string      fFldId;
    FieldValue fValue;
    short      fIndex;
};
```

```
typedef sequence<NVField> AccessKeyList;
```

```
typedef sequence<AccessField> ArgumentList;
```

```
interface GeneralAccess
{
    t_RetCode process
    (
        // Function Name to perform
        in string      pFuncName ,
        // Search keys may be empty
        in  AccessKeyList pKeys ,
        // Names to be read/written
        inout ArgumentList pArguments ,
        // Signals various states.
        out long  pFlag
    )
    raises ( ABException );
}
```

```
struct MsgRecord
{
    long          fRecIndx;
    string       fData; // Effective Data in XML
};

// List of messages sent.
typedef sequence <MsgRecord>MsgRecList;
// List of simple Data
typedef sequence <string>DataList;
```

```
interface OrderAgent
{
// Send Data to the Provider to be processed
RetCode process
( in ApplID_t pApplId ,
  in Command pCmd ,
  in MsgRecList pMsgList ) ;
// Get Data after processing (by Notification)
RetCode getData
( in ApplID_t pApplId ,
  in long pMaxRecords ,
  out MsgRecList pData ) } ;
in long pMaxRecords ,
out MsgRecList pData ) ;
```



## Beim Erstellen

- Schnittstelle so einfach wie möglich entwickeln
- Keine zu hohe Granularität wählen
- Einfachheit ist auch hier der Schlüssel zum Erfolg.
- Nur den unmittelbaren Bedarf abdecken
- Spätere Erweiterung ist leichter möglich

## Beim Aendern & Erweitern

Wenn eine Schnittstelle erweitert wird, dann:

- neue Strukturen einfügen, statt bestehende ändern
- neue Operationen, statt bestehende ändern
- durch “Vererbung” erweitern, damit bestehende Schnittstellen beibehalten werden können

## Interface-Design:

## Tips (2)



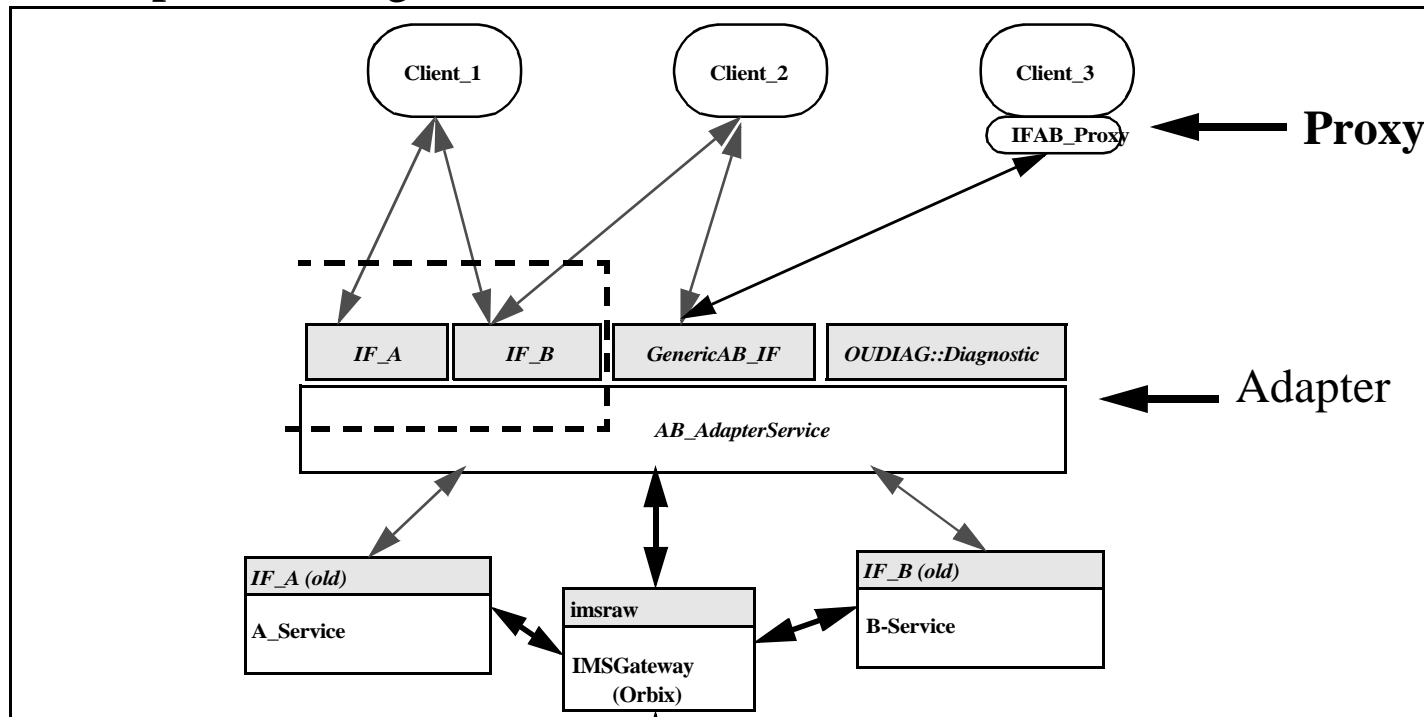
- Wenn Schnittstellen festgelegt, je einen Test-Client und einen Prototypen-Service (mit fiktiven Rückgabewerten) erstellen und überprüfen, ob die definierte Schnittstelle handlich, einfach benutzbar und natürlich zweckmässig ist.
- Test-Client und -Prototypen-Server können dann auch während der Entwicklungsphase gegenseitig genutzt werden.
- Test-Clients sollten so ausgebaut werden, dass sie Testsequenzen (Regressionstest, Massentest) ab einem Speichermedium ablaufen lassen können und die Ergebnisse überprüfen und protokollieren können.
- Test-Clients sollten auch Zeiten messen können.
- Zeitliche Optimierung erst vornehmen, wenn System funktionell stabil läuft (z.B. dann Einbau Smart-Proxy).

## Interface-Design:

## Typische Pattern



- **Facade:** Wrapping von bestehenden Applikationen
- **Bridge, Mediator:** Zum Beispiel der Event-Service
- **Proxy:** die Client-Seite eines verteilten Objektes
- **Adapter:** für Migration von Schnittstellen

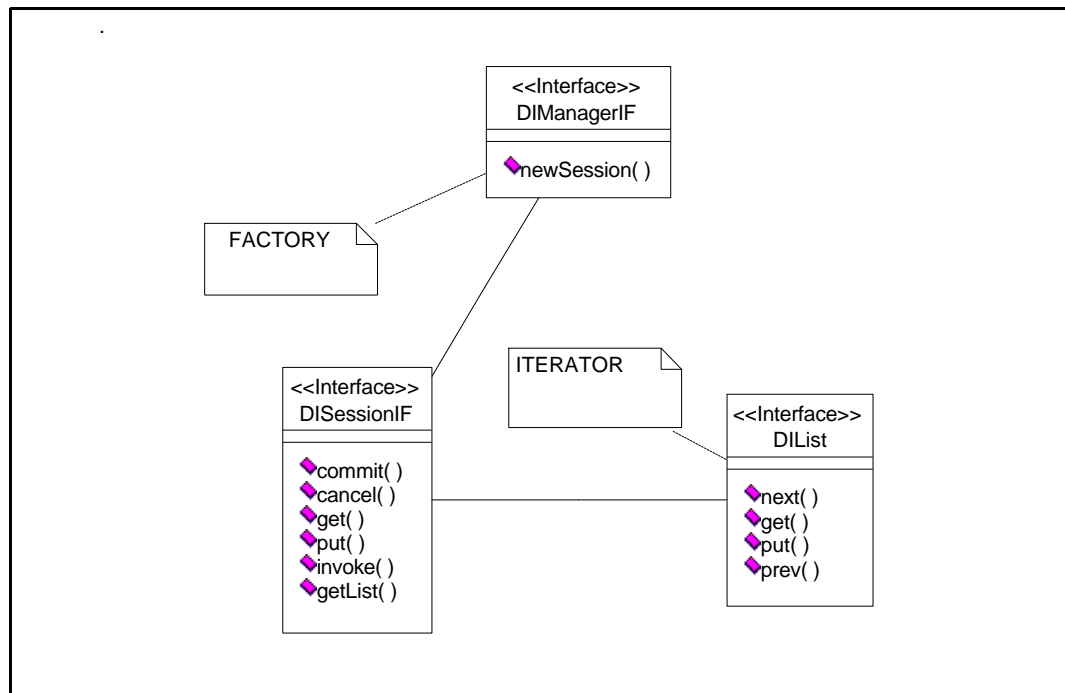


## Interface-Design:

## Factory-Pattern



- **Factory:** Häufig gibt es ein Manager-Objekt, welches die Erstellung von Objekten übernimmt und evtl. diese auch verwaltet.
- **Iterator:** Für das Durchlaufen von entfernten Listen ohne dabei den Status des Servers zu verändern. z.B. Naming, u.a.



## Zusammenfassung

---



- Schnittstellen sind in der Realisation (ausprogrammiert) um Faktoren komplexer, als es die Schnittstellen-Definition erkennen lässt.
- Schnittstellen-Design muss in durch die Architektur gegeben sein.
- Auch wenn CORBA-IDL-Sprache nicht umfangreich ist, muss man trotzdem nicht alle Möglichkeiten davon nützen (Weniger ist oft mehr).
- Performance ist in der Regel gegensätzlich zu Flexibilität
- **attribute** und **any** sind üblicherweise Performance-Killer
- Nachträgliches Aendern von Schnittstellen, kann zu “broken Code” führen.
- Aufteilen und “Vererben” von Schnittstellen bringt Design-Flexibilität.
- Generische Schnittstellen erfordern teils aufwendige Interpretationen auf Client wie Service-Seite.
- XML kann wesentlich zur Flexibilität beitragen ⇨ *Message Paradigma*
- Die IDEaLe Schnittstelle existiert nicht, aber sie kann angestrebt werden.