OMG IDL
Interface Definition Language

IDL ist die Basis für den grundlegenden Abstraktionsmechanismus in CORBA:
Trennung von Interface und Implementation.

In IDL werden als Vertrag zwischen Client und Server *Typen und Schnittstellen* der Objekte der Anwendung definiert.

IDL ist *sprachunabhängig* und rein *deklarativ*.

Abbildungen auf konkrete Programmiersprachen werden in *Language Mappings* definiert.

Wichtigste Bestandteile:

- einfache Typen, Konstanten
- strukturierte Typen, Exceptions
- Interfaces mit Methoden und Attributen
- Module

Lexik

- Zeichensatz: ISO-Latin-1
- Leerzeichen, Tabulatoren, Zeilenwechsel nur für Layout
- IDL-Dateien werden durch einen C++-Präprozessor vorverarbeitet
  → Kommentare werden entfernt `/* .. */` `// ...`
  → `#include`, `#define`, `#ifdef`, etc.
- es gibt eine Reihe von Keywords: `interface`, `any`, `struct`, `short`, `Object`, `TRUE`, `oneway`, ...
- Bezeichner beginnen mit (lateinischen) Buchstaben, gefolgt von Buchstaben, Ziffern und Unterstrich (führender Unterstrich erlaubt ab CORBA 2.3 sogenannte *escaped identifier*)
- Groß- und Kleinschreibung wird beachtet, zwei Bezeichner dürfen sich jedoch nicht nur darin unterscheiden!
  (z.B. `SHORT` oder `false` sind ungültig)
  → ermöglicht das Mapping auf Sprachen wie Pascal

Basistypen

Typ	Bereich	Größe
`short`	-2^{15} bis $2^{15}-1$	≥ 16 Bit
`long`	-2^{31} bis $2^{31}-1$	≥ 32 Bit
`unsigned short`	0 bis $2^{16}-1$	≥ 16 Bit
`unsigned long`	0 bis $2^{32}-1$	≥ 32 Bit
`float`	IEEE single-precision	≥ 32 Bit
`double`	IEEE double-precision	≥ 64 Bit
`char`	ISO Latin-1	≥ 8 Bit
`string`	ISO Latin-1, ohne 0	variabel
`boolean`	`TRUE`, `FALSE`	k.A.
`octet`	0-255	≥ 8 Bit
`any`	beliebig	variabel

Darüber hinaus gibt es seit CORBA 2.2:
`wchar`, `wstring` (wide-Character Zeichensatz),
`long long`, `unsigned long long` (jeweils ≥ 64 Bit),
`long double` sowie `fixed`

Literale und Konstanten

Integer	`123, 014, 0x12, 0Xab`
Zeichen	`'x', '\007', '\x41', '\n', '\t'`
Strings	`"text", "hello \"world\"\n", "äöüß"`
Fließkomma	`3.14159, 5.0e-10, 1E10, .1`

```
const double e = 2.718281828;
```


Typedef, benannte Typen

```
typedef short YearType;
typedef short TempType;
```

Semantik, ob `YearType` und `TempType` austauschbar sind, ist sprachabhängig!
(in C++ und Java ist dies der Fall)

Aufzählungstypen

```
enum Farbe { blau, rot, braun };
```

→ Abbildung auf einen Typ mit mindestens 32 Bit
→ Zuordnung zu Ordinalzahlen nicht definiert
(damit Übertragung solcher Zahlen nicht portabel)
→ keine explizite Zuweisung von Ordinalzahlen möglich

```
enum Color { red = 0, green = 7 };   // Fehler
enum Ampel { rot, gelb, gruen };     // Fehler
```

→ `rot` bereits definiert (im umgebenen Scope)

Ab CORBA 2.3 existieren Aufzählungskonstanten.

Strukturen

```
struct Internals {
   short s;
   long val;
};

struct MyStruct {
   short s;
   Object o;
   Internals i;
};
```

inhaltlich das gleiche, aber syntaktisch unterschiedlich (und daher verschiedene Typen):

```
struct MyStruct {
   short s;
   Object o;
   struct Internals {
      short s;
      long val;
   } i;
};
```

Jede Struktur bildet einen eigenen Namensraum.

Unions

```
union Info switch (short) {
   case 1  : long l;
   case 2  :
   case 3  : string str;
   default : float f;
};
```

→ `short` ist der Diskriminatortyp (erlaubt sind `char`, `boolean`, Integer- und Aufzählungstypen)
→ Zugriff auf nicht aktive Member ist undefiniert

```
union U switch (Farbe) {
   case blau  : long l;
   case rot   : Object o;
   case braun : char c;
   default    : string msg;   // Fehler
};
```

→ kein Wert für den default-Zweig verfügbar

```
union OptFlag switch (boolean) {
   case TRUE : short flag;
};
```

→ zur Repräsentation optionaler Werte

Unions sollten nicht als Ersatz für (verbotene) überladene Methoden missbraucht werden.

Any

- universaler Container-Typ, der Werte beliebiger IDL-Typen aufnehmen kann
- enthaltener Typ kann zur Laufzeit erkannt werden (enthält Typ-Information als `CORBA::TypeCode`)
- Zugriff ist typsicher

```
interface UniversalStore {
   void put (in string name, in any value);
   any get (in string name);
};
```

Strings

gibt es bounded und unbounded:

```
typedef string Identifier;          // unbounded
typedef string<8> PasswordString;   // bounded
```


Wann verwende ich Unions, wann Anys?

Any:

- erlaubt die Aufnahme beliebiger Werte, auch solcher nutzerdefinierter Typen, die zum Programmierzeitpunkt noch nicht bekannt waren
- höherer Aufwand zum Erkennen des enthaltenen Wertes, Unterscheidung nur anhand des Typecodes möglich

Union:

- bessere Typsicherheit, erlaubt nur die Aufnahme zuvor definierter Typen
- spiegelt bei eingeschränktem Typbereich die gewünschte Semantik angemessener wider

Arrays

- ein- oder mehrdimensional
- alle Dimensionen müssen angegeben werden (keine offenen Arrays)
- Indizierung wird erst in der Zielsprache durch das Sprachmapping definiert (d.h. Übertragung eines Index ist i.a. keine gute Idee)
- Arrays müssen per `typedef` immer als benannter Typ definiert werden

```
typedef Farbe Palette[10];

typedef string Dict[20][2];
// typedef string Dtmp[2];
// typedef dtmp Dict[20];

typedef float T[];              // Fehler
struct S { long liste[5]; };    // Fehler
```


Sequenzen

- Vektoren variabler Länge beliebiger Elementtypen
- begrenzt (bounded) oder unbegrenzt (unbounded)
- können leer sein
- können (im Gegensatz zu Arrays) als anonyme Typen auftreten

```
typedef sequence <Farbe> Farben;
// -> unbounded

typedef sequence <string, 20> IDtable;
// -> bounded (maximal 20 Elemente)
```

Als anonymer Typ:

```
typedef sequence <sequence <long, 50> >
   NumVec_List;
struct S { sequence <char> vec; };
```

Solche anonymen Bestandteile können nicht separat behandelt (instanziiert, übertragen, etc.) werden.
Besser: immer benannte Typen verwenden

Wann verwende ich Sequenzen, wann Arrays?

Entscheidungshilfen:

Arrays:

- für eine feste Anzahl von Elementen, die alle übermittelt werden müssen

Sequenzen:

- für Listen variabler Länge
- für rekursive Strukturen
- für dünnbesetzte Arrays, d.h. Felder, bei denen die meisten Elemente unbesetzt sind
  (d.h. den Default-Wert haben, z.B. 0 in Matrizen)

  In diesem Fall kann es effizienter sein, nur die gesetzten Felder in einer Sequenz zu übermitteln

Beispiel:

```
typedef long Matrix[100][100];

interface MatrixProcessor {
   Matrix invertMatrix (in Matrix m);
};
```

führt zur Übertragung von jeweils 10 000 Zahlen hin und zurück (80 000 Bytes)!

Wenn mehr als die Hälfte der Matrix-Elemente 0 sind, lohnt sich statt dessen folgende Darstellung einer Matrix:

```
struct MatrixElement {
   unsigned short row;   // Zeile
   unsigned short col;   // Spalte
   long val;             // Wert
};
typedef sequence <MatrixElement> Matrix;

interface MatrixProcessor {
   Matrix invertMatrix (in Matrix m);
};
```

Rekursive Typen

- nur innerhalb von Strukturen und Unions
- mittels anonymer Sequenzen des (noch unvollständigen) zu definierenden Typs

```
struct Node {
   long value;
   sequence <Node, 2> children;
   // Node children[2];   illegal, keine Sequenz
};
```

→ binärer Baum
(Blätter sind Nodes mit leerer `children`-Sequenz)

Achtung!
Der Typ ist rekursiv, aber **nicht** die enthaltenen Daten. Zyklische Graphen oder doppelt verkettete Listen lassen sich so nicht ausdrücken.

Elemente von Sequenzen sind immer Werte (Values) und niemals Zeiger.

Rekursion kann über mehrere Stufen erfolgen:

```
struct Outer {
   string id;
   struct Inner {
      long value;
      sequence <Outer> children;
   } data;
};
```

Gegenseitige Rekursion zwischen verschiedenen Strukturen ist nicht möglich:

```
struct A {
   long data;
   sequence <B, 1> nested;  // B nicht definiert
};
struct B {
   long data;
   sequence <A, 1> nested;
};
```

Forward-Deklarationen für Strukturen sind in IDL nicht erlaubt.

Mögliche Lösung:
Rekursives Union, das entweder A- oder B-Elemente enthält.

Konstantenausdrücke

Arithmetische Operatoren: `+ - * / %`

- `%` (Modulo-Operator) nur für Integer-Ausdrücke
- keine gemischten Ausdrücke Float/Integer
  → `3.14 * 2` ist illegal
- Integer-Konstanten werden immer als `long` ausgewertet und das Ergebnis in den Resultatstyp umgewandelt

Bitweise Operatoren: `| & ^ << >> ~`

- nur für Integer-Ausdrücke
- right shift (>>) ist immer logisch (ohne Interpretation des Vorzeichens)
  (entspricht >>> in Java)

```
const long ALL_ONES = -1;              // 0xffffffff

const long LHW_MASK = ALL_ONES << 16; // 0xffff0000
const long RHW_MASK = ALL_ONES >> 16; // 0x0000ffff
```


Interface

- definiert die Schnittstelle eines CORBA-Objektes, jedes CORBA-Objekt besitzt genau ein Interface
- bildet einen eigenen Namensraum
- alle Deklarationen sind öffentlich (quasi `public`), es existiert keine Entsprechung zu `private` oder `protected`
- Forward-Deklarationen sind möglich

```
interface foo;   // Forward-Deklaration

interface bar {
   foo getFoo (in bar obj);
};

interface foo {
   bar getBar (in foo obj);
};
```

Interfaces können enthalten:

- Konstantendefinitionen
- Typedefinitionen (Typedefs, strukturierte Typen)
- Definitionen von Ausnahmen
- Attribute
- Operationen
- insbesondere **keine** weiteren Interfaces! (innere Klassen in C++/Java)

```
interface Heuhaufen {
   const short MAX_LAENGE = 10;

   typedef long Nadel;

   exception NichtGefunden { };

   attribute unsigned long groesse;

   void finde (in Nadel n)
      raises (NichtGefunden);
};
```


Interface-Semantik

- Die Deklaration eines Interface definiert einen neuen Typbezeichner
- Werte dieses Typs sind Objektreferenzen

```
interface Scheune {
   void einlagern (in Heuhaufen h);

   Heuhaufen finde (in Heuhaufen::Nadel n)
      raises (Heuhaufen::NichtGefunden);

   void verstecke (in Heuhaufen h,
                   in Heuhaufen::Nadel n);
};
```

- `Heuhaufen` und `Scheune` können völlig unabhängig voneinander implementiert sein
- der `einlagern`-Operation in `Scheune` können nur `Heuhaufen`-Objekte (oder abgeleitete sowie `nil`) übergeben werden
- *Wie kommt die Nadel in den Heuhaufen?*
  → nur über versteckte Kommunikation zwischen den Objekten (außerhalb von CORBA)!

Operationen

- immer innerhalb eines Interface
- obligatorisch: Ergebnistyp und eindeutiger Name (keine überladenen Operationen)
- enthält 0 oder mehr Parameter-Deklarationen (IDL ähnelt in dieser Hinsicht eher Java als C++)
- optional: Exceptions, Context
- jeder Parameter besitzt ein Richtungsattribut: `in`, `out` oder `inout`

```
interface Simple {
   void op1();
   boolean checkPasswd (in string login,
                        in string password);
   void getIterator (in long maxSize,
                     out long size,
                     out Iterator i);
   boolean crypt (inout string text);
};
```

Ergebnis- und Parametertypen müssen immer benannt sein.

Wie gebe ich Ergebnisse von Operationen zurück?

Prinzipiell gibt es drei Möglichkeiten, das Resultat eines Operationsaufrufes mitzuteilen:

1. als normaler Return-Wert einer Operation
2. per `out`- oder `inout`-Parameter
3. über eine Exception

Entscheidungshilfen:

- Operationen, die ein einzelnes Ergebnis liefern, sollten dieses als Return-Wert zurückgeben
- Operationen, die mehrere gleichwichtige Ergebnisse zurückgeben, sollten dies per `out`-Parameter tun und den Ergebnistyp `void` besitzen
- gibt es einen speziellen Rückgabewert unter mehreren, sollte dieser der Return-Wert sein
  Beispiel: Iterator

  ```
  boolean get_next (out ValType val);
  ```

- `inout`-Parameter sind dann sinnvoll, wenn zur Modifikation sehr großer Datenmengen nur einmal Speicherplatz im Klienten oder im Servant angelegt werden soll

Oneway-Operationen

- gekennzeichnet durch Schlüsselwort `oneway`
- Operation muss den Ergebnistyp `void` besitzen
- es dürfen keine `out`- oder `inout`-Parameter auftreten
- es können keine Exceptions deklariert werden

```
interface Channel {
   oneway void send (in string event);
};
```

Intention: asynchron Nachrichten versenden
Semantik: *best effort*

→ keine gesicherte Übermittlung
→ keine Zusicherung über die Reihenfolge der Aufrufe
→ ebenso: keine Zusicherung der Asynchronität

User-Exceptions

- Mittel, um Fehlerzustände in der Anwendung anzuzeigen
- Definition ähnelt stark der von Strukturen
- Exceptiondefinitionen bilden keinen Datentyp, keine Vererbung möglich
- werden durch `raises` bei Operationen deklariert

```
interface Unreliable {

   exception Failed { };

   exception RangeError {
      unsigned long supplied_val;
      unsigned long min_permitted_val;
      unsigned long max_permitted_val;
   };

   void can_fail (in long value)
      raises (Failed, RangeError);
};
```

System-Exceptions

- können jederzeit aufgrund von Fehlern bei der Requestbearbeitung auftreten (auch bei `oneway`-Operationen)
- dürfen nicht mittels `raises` bei Operationen deklariert werden

Grundaufbau:

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

exception <Name> {
   unsigned long minor;
   completion_status completed;
};
```

Die Bedeutung des Minor-Codes ist durch CORBA nicht definiert, d.h. ORB-abhängig.

System-Exceptions (Beispiele)

CORBA 2.3 kennt insgesamt 29 System-Exceptions (definiert im Modul `CORBA`)

`BAD_OPERATION`	ungültige Operation
`BAD_PARAM`	falscher Parameter
`COMM_FAILURE`	Kommunikationsfehler
`INTF_REPOS`	Interface Repository nicht verfügbar
`INV_OBJ`	ungültige Objektreferenz
`MARSHAL`	Fehler beim Marshaling
`NO_IMPLEMENT`	Implementation nicht verfügbar
`OBJECT_NOT_EXIST`	Objekt existiert nicht
`TRANSIENT`	vorübergehender Fehler
`UNKNOWN`	unbekannte Exception aufgetreten

Die Bedeutung einiger System-Exceptions ist nicht eindeutig genug definiert. Es können daher Abweichungen von ORB zu ORB auftreten.

System-Exceptions sollten niemals zur Anzeige eines anwendungsspezifischen Fehlers ausgelöst werden.

Verwendung von Exceptions:

- Exceptions sollten auch nur in *Ausnahmefällen* ausgelöst werden, niemals für erwartete Ergebnisse

  schlecht:

```
interface Database {
   typedef sequence <Record> RecordSeq;
   exception NotFound { };

   RecordSeq lookup (in string query)
      raises (NotFound);
};
```

  → eine leere Sequenz kann anzeigen, dass nichts passendes gefunden wurde

- andererseits: Fehler sollten nicht als normales Ergebnis zurückgegeben werden

  schlecht:

```
interface FileSystem {
   long getSize (in string filename);
};
```

  → statt negative Werte für unterschiedliche Fehlerzustände zurückzugeben, sollten besser Exceptions ausgelöst werden

- eine Exception muss sinnvolle, genaue und vollständige Informationen über die Ursache des aufgetretenen Fehlers enthalten

  schlecht:

```
exception SyntaxError { string query; };
```

  → die Zeichenkette `query` ist dem Aufrufer bekannt, außerdem erfährt er nichts darüber, welche Stelle seiner Frage syntaktisch fehlerhaft ist

  besser:

```
exception SyntaxError { short pos; };
```

- verschiedene Fehlerzustände sollten durch unterschiedliche Exceptiontypen, und nicht durch verschiedene Daten als Bestandteil eines Exceptiontyps signalisiert werden

  schlecht:

```
exception Failure {
   short code;
   long additionalInfo;
   string reason;
};
```

Attribute

- erscheinen wie Member-Variablen einer Klasse
- definieren in Wirklichkeit ein Paar von Lese- und Schreiboperationen
- können `readonly` deklariert werden
- können keine User-Exceptions werfen, aber durchaus System-Exceptions

```
interface I {
   attribute short a;
   readonly attribute long b;
};
```

entspricht semantisch:

```
interface I {
   short get_a();
   void set_a (in short value);
   long get_b();
};
```


Wenn möglich, sollte auf Attribute verzichtet werden!

- Ein Zugriff könnte Nebeneffekte haben.
- Werte können sich willkürlich im Objekt ändern.
- Falsche Werte können nicht behandelt werden.

```
struct Date {
   short day;
   short month;
   short year;
};
```

Variante 1 mit Attribut (ungünstig):

```
interface Planer {
   attribute Date ultimo;
};
```

Variante 2 mit Operationen (besser):

```
interface Planer {
   exception IllegalDate { };
   exception NotSet { };

   void setUltimo (in Date ultimo)
      raises (IllegalDate);
   Date getUltimo()
      raises (NotSet);
};
```

Interface-Vererbung

→ wie Vererbung in C++ (bzw. `extends` in Java)

```
interface A { void myOp(); };
interface B { long myOp(); };
interface C { short yourOp(); };

interface D : A { void op(); };
// ok

interface E : A, B { };
// Fehler, Namenskollision bei myOp

interface F : A, C { };
// ok

interface G : D, F { };
// ok

interface H : A { void myOp(); };
// Fehler, Namenskollision bei myOp

interface I;  // Forward-Deklaration
interface J : I { };
interface I { };
// Fehler, Basisinterface noch nicht definiert
```


Alle CORBA-Objekte erben implizit von `Object`.

```
interface Counter : Object {  // Fehler
   ...
};

interface SimpleNameService {
   void bind (in string n, Object o);

   Object resolve (in string n);
};
```

Leere Interfaces sind möglich –
als Ersatz für abstrakte Basis-Interfaces.

Fraglich: gute Modellierung –
ein leeres Interface definiert keinerlei Eigenschaften.

```
interface Fahrzeug { };
interface Auto : Fahrzeug { ... };
interface Roller : Fahrzeug { ... }:

interface Garage {
   void parke (in Fahrzeug f);
};
```

Welche Konzepte lassen sich durch Objekte, welche besser durch Strukturen modellieren?

Interfaces, die nach dem ersten Entwurf nur `readonly`-Attribute enthalten, lassen sich eventuell besser durch entsprechende Strukturen beschreiben.

Objekte

- besitzen einen Zustand und „leben“ innerhalb eines Servers
- werden *per reference* übermittelt, mehrere Klienten besitzen möglicherweise eine Referenz auf das gleiche Objekt
- sind in der Regel entfernt, der Zugriff auf Daten des Objekts bewirkt einen entfernten Funktionsaufruf
- Änderungen des Zustandes (der Attribute) werden für alle Besitzer der Referenz wirksam
- ermöglichen die Modellierung von Vererbungsbeziehungen

Strukturen

- sind reine Daten und werden *per value* übermittelt
- ermöglichen damit den lokalen und schnellen Zugriff auf enthaltene Informationen
- erlauben keine Vererbung

Beispiel: Einträge einer Datenbank

```
interface Database {
   typedef sequence <Record> RecordSeq;

   RecordSeq lookup (in string query);
};
```

Wie sollte `Record` definiert sein?

über ein Interface, falls

- Vererbungsbeziehungen ausgedrückt werden sollen
- mehrere und große Datenmengen bereitgestellt werden, von denen aber jeweils nur Teile wirklich benötigt werden
- die Daten zustandsbehaftet sind und sich dies über den Wert der Attribute bemerkbar macht

als Struktur, falls

- es sich um statische Daten handelt
- alle enthaltenen Daten benötigt werden

Gegebenenfalls kann es günstig sein, sowohl einen Zugriff über eine Objektreferenz als auch über eine Datenstruktur parallel anzubieten.

Modul

- definieren einen Namensraum für Bezeichner
- können ineinander verschachtelt sein
- können wiedereröffnet werden
- können enthalten: Typdefinitionen, Konstanten, Exceptions, Interfaces, Module

```
module DomainA {
   interface I { void op(); };
};

module DomainB {
   module SubdomainC {
      interface I : DomainA::I { };
      interface J : ::DomainA::I { };  // dito
      interface K { /* ... */ };
   };
};

module DomainA {
   // weitere Definitionen
};
```


Alle Bezeichner des übergeordneten Namensraums sind sichtbar:

```
module A {
   typedef long TypeL;

   interface I {
      TypeL op1();
   };
};

module B {
   interface J : A::I {
      void op2 (in TypeL arg);  // Fehler
   };
};
```

→ `TypeL` ist zwar in `A::I` sichtbar, aber nicht in `B::J`

Zusammengehörige Definitionen und Interface sollten innerhalb eines Moduls gruppiert werden.

Interface-spezifische Definitionen gehören in dieses Interface.

Beispiel 1: Gültigkeitsbereich zu groß:

```
exception CantDoA { };
exception CantDoB { };

interface A {
   void doA () raises (CantDoA);
};

interface B {
   void doB () raises (CantDoB);
};
```

besser:

```
interface A {
   exception CantDoA { };

   void doA () raises (CantDoA);
};

interface B {
   exception CantDoB { };

   void doB () raises (CantDoB);
};
```


Beispiel 2: Gültigkeitsbereich zu eng

```
interface A {
   typedef long Length;
   exception NoResources { };

   void doA (in Length l) raises (NoResources);
};

interface B {
   A::Length doB () raises (A::NoResources);
};
```

besser:

```
typedef long Length;
exception NoResources { };

interface A {
   void doA (in Length l) raises (NoResources);
};

interface B {
   Length doB () raises (NoResources);
};
```

Das Interface Object

Jedes CORBA-Objekt unterstützt folgende Operationen:

```
interface Object {            // Pseudo IDL
   boolean is_nil();

   Object duplicate();
   void release();
   // -> beziehen sich nur auf Objektreferenzen,
   //    nicht auf die Objektimplementation

   boolean non_existent();

   boolean is_equivalent (in Object otherObj);

   InterfaceDef get_interface();

   boolean is_a (in string type_id);

   // und weitere ...
};
```

Die konkrete Benutzung ist in den jeweiligen Sprachmappings geregelt (deswegen „Pseudo-IDL“).
→ z.B. sind in C++ einige der Operationen durch Klassenmethoden realisiert

Das Interface ORB

Der ORB ist *kein* CORBA-Objekt, seine Schnittstelle wird jedoch ebenfalls in (Pseudo)-IDL beschrieben:

```
interface ORB {               // Pseudo IDL
   typedef string ObjectId;
   typedef sequence <ObjectId> ObjectIdList;
   exception InvalidName { };

   ObjectIdList list_initial_services();
   Object resolve_initial_references
      (in ObjectId identifier);
      raises (InvalidName);

   string object_to_string (in Object obj);
   Object string_to_object (in string str);

   void run();
   boolean work_pending();
   void perform_work();
   void shutdown
      (in boolean wait_for_completion);

   void destroy();

   // und viele weitere ...
};
```

Interface-Entwurf

Gegeben: Beschreibung einer komplexen Situation

Gesucht: Menge von CORBA-Objekten, die die Objekte der realen Welt angemessen repräsentieren

Grundregeln:

- Entwurf aus der Sicht des Klienten, d.h. welche Funktionalität möchte ich vorfinden
- strikte Trennung von Implementation und Interface
- Implementationsaspekte (private Daten) gehören nicht ins Interface
- Operationen und Attribute sind immer public, Zugriffsbeschränkungen müssen durch Aufteilung in verschiedene Interfaces modelliert werden
- geeignete Gruppierung von Typen und Interfaces in Modulen
- „sprechende“ Bezeichner, verbale Beschreibung der Semantik

Operationen, die sich auf ein bestimmtes Objekt beziehen, sollten (im Sinne der Objektorientierung) im Interface dieses Objektes implementiert sein:

```
interface Counter {
   // ... Counter-Funktionalitaet

   void destroy();
};
```

statt

```
interface CounterFactory {
   Counter create();
   void destroy (in Counter obj);
};
```

Dies verhindert die missbräuchliche Benutzung mit falschen Argumenten.

(hier: Übergabe eines Counter-Objektes, das von einer anderen Factory erzeugt wurde)

Wichtiger Designaspekt:
kleinste Einheit ist nicht ein einzelnes Interface, sondern eine Menge zusammengehörender Interfaces (Komponente, Modul)

(Prinzip der Abstraktion von der Implementierung verwischt hier ein wenig)

- zusammengehörende Objekte einer Komponente können lokal Daten austauschen
- deren Kommunikation wird nicht im IDL-Interface sichtbar

Beispiel:
CounterFactory besitze ein Attribut über die Anzahl existierender Counter-Objekte:

```
interface CounterFactory {
   readonly attribute long counterNumber;

   Counter create();
};
```

Jeder Aufruf von `CounterFactory::create()` erhöht den Wert des Attributes `counterNumber`, jeder Aufruf von `Counter::destroy()` vermindert ihn wieder.

Verschiedene Sichten und Zugriffsebenen können in separaten und unabhängigen Interfaces modelliert werden.

Beispiel:
ein spezielles Admin-Interface zur `CounterFactory` soll Informationen darüber bereitstellen, wie lange deren Counter-Objekte nicht mehr benutzt worden sind:

```
module CountAdmin {
   struct Idle {
      Count::Counter obj;
      long days;
   };
   typedef sequence <Idle> IdleSeq;

   interface FactoryAdmin {
      readonly attribute IdleSeq createdObjects;
   };
};
```

Es gibt keine in IDL sichtbare Verbindung zwischen `Count::CounterFactory` und `CountAdmin::FactoryAdmin`

Über die Verteilung der Objektreferenzen wird gesteuert, wer welches Interface benutzen darf.

Was ist von folgender Definition zu halten?

```
module CountAdmin {
   // Typdefinition von IdleSeq

   interface FactoryAdmin :
             Count::CounterFactory {
      readonly attribute IdleSeq createdObjects;
   };
};
```

Idee:
Es gibt nur eine Implementation, Klienten erhalten eine Referenz auf das Interface `Count::CounterFactory`

Aber:
Klienten, die eine Objektreferenz eines solchen Objekts besitzen, können natürlich nach einem `narrow` auf `CountAdmin::FactoryAdmin` auch die Operationen dieses Interface nutzen.