

CORBA

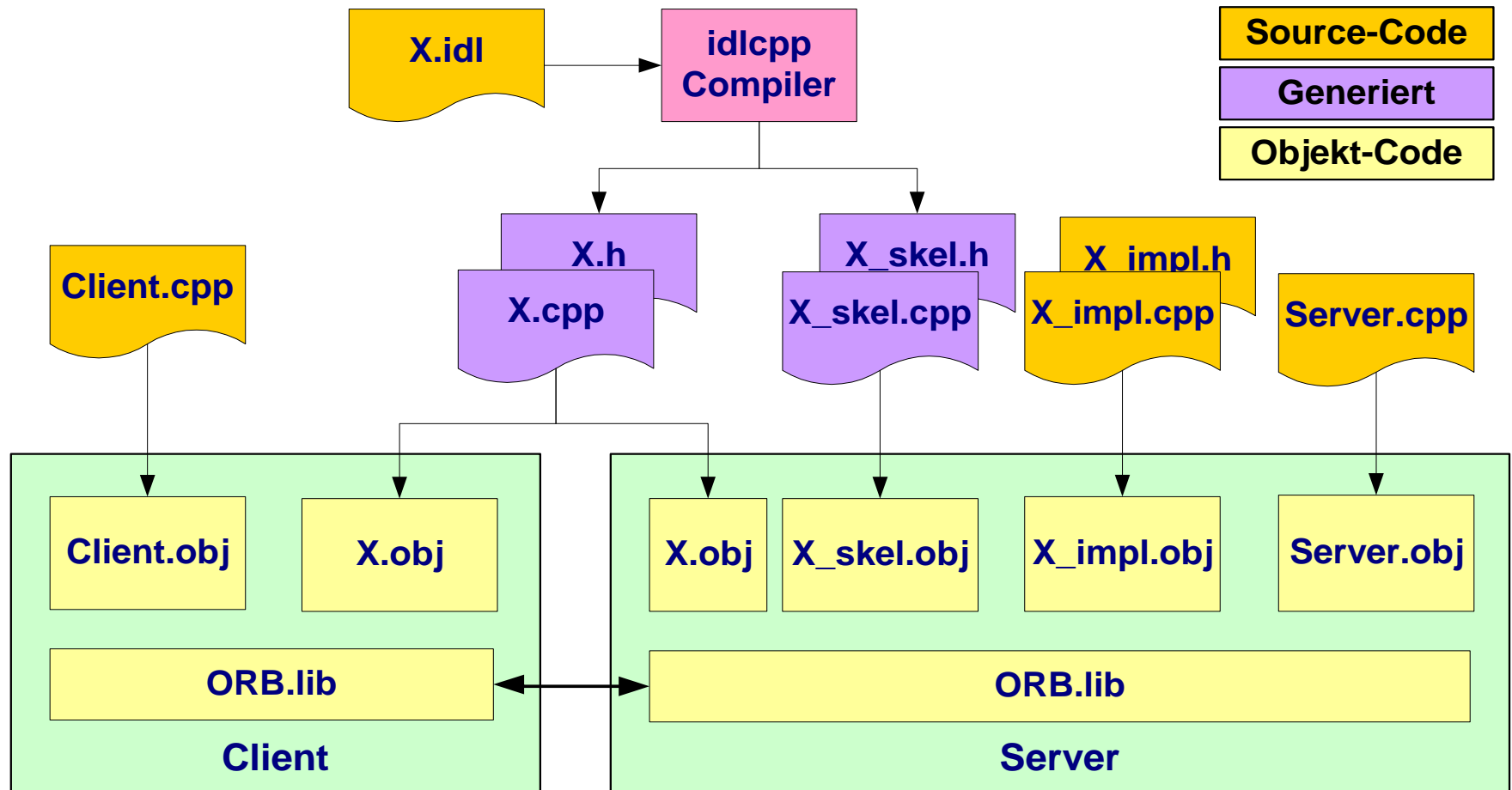
Interface Definition Language

© J. Heinzlreiter
WS 2003/04

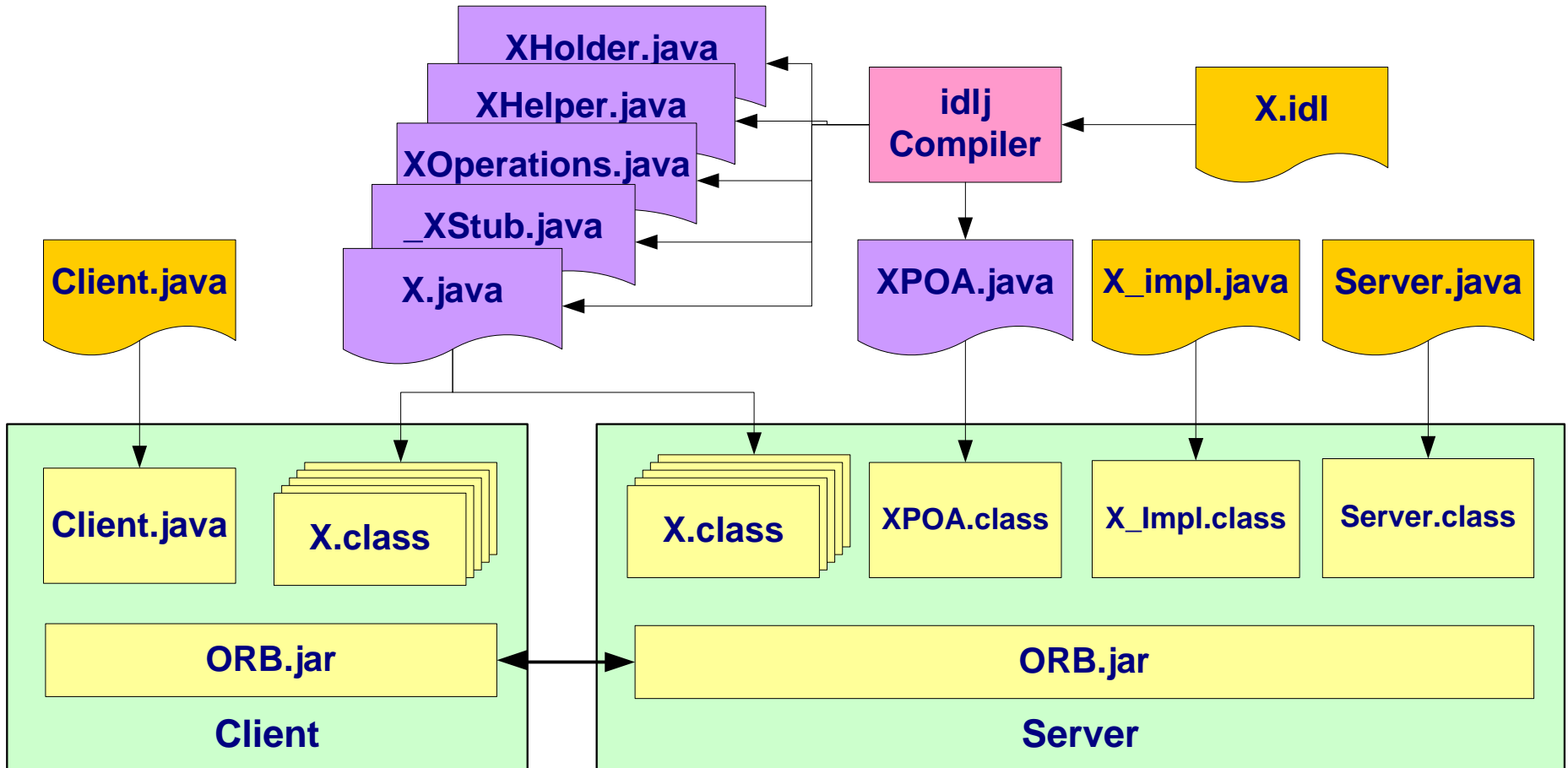
Interface Definition Language (IDL)

- Abstraktionsmechanismus zur Trennung von *sprachunabhängigen Schnittstellen* von *sprachabhängigen Implementierungen*.
- IDL ist ausschließlich *deklarativ*.
- IDL-Spezifikationen sind analog zu *Java-Interfaces*.
- *IDL-Compiler* übersetzen IDL-Spezifikation in sprachabhängige Datentypen und APIs.
- *Language mapping*: Abbildung der IDL auf Konstrukte einer konkreten Programmiersprache.
- Daten können nur über Datentypen ausgetauscht werden, die in IDL definiert sind.

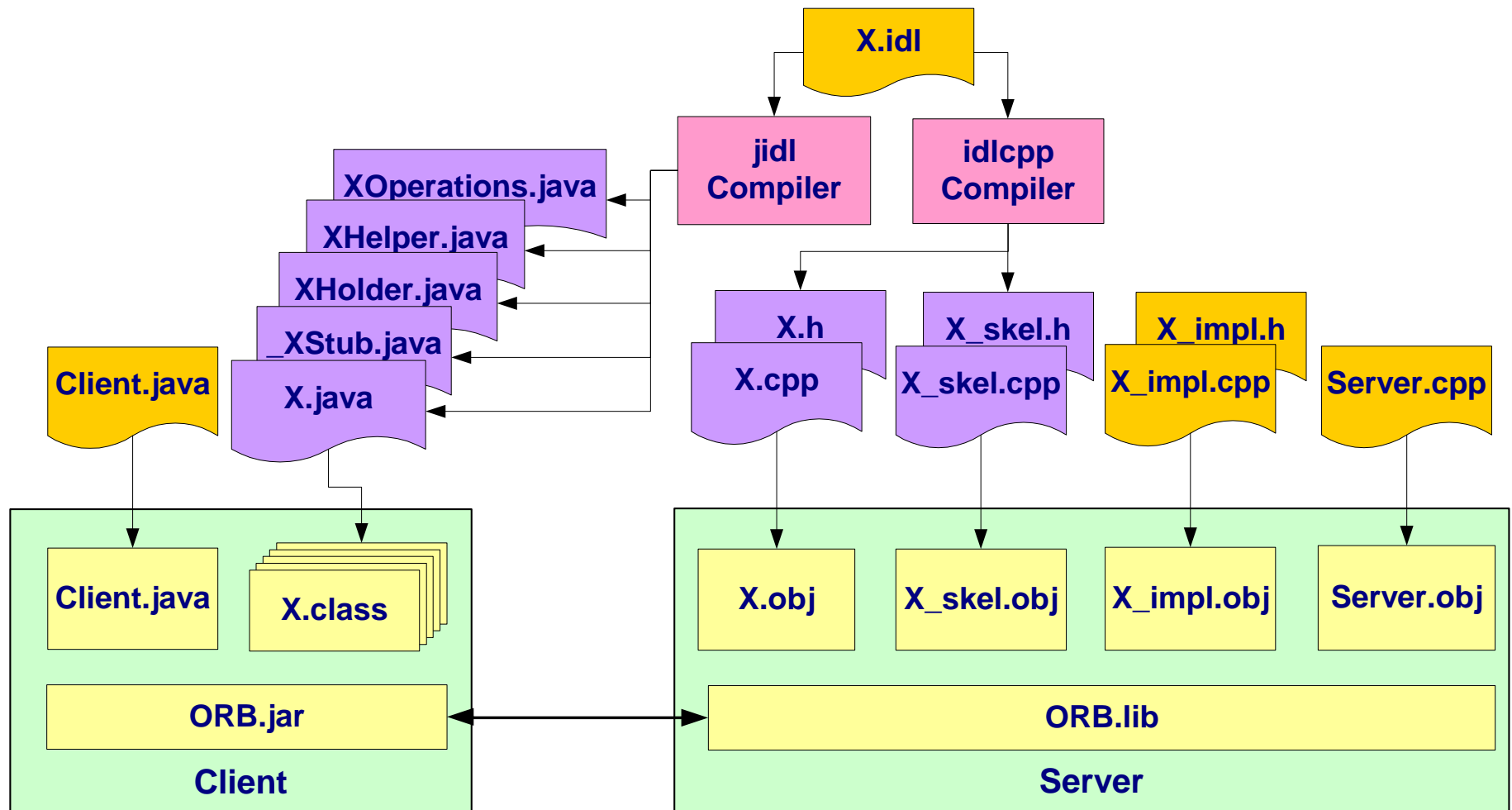
IDL-Compiler für C++



IDL-Compiler für Java



Mixed Mode: C++-Server/Java-Client



Lexikalische Anforderungen

- Vorgeschriebene Extension: `.idl`.
- Freies Format.
- IDL-Files werden von einem *C++-Präprozessor* vorverarbeitet.
- *C/C++-Kommentare*.
- Symbole müssen vor erster Verwendung definiert werden.
- IDL-*Schlüsselwörter* werden kleingeschrieben.
 - Ausnahme: TRUE, FALSE, Object, ValueBase.
- Bezeichner:
 - Syntax: (`"_"` | *letter*) { `"_"` | *letter* | *digit* }
 - Groß/Kleinschreibung ist relevant,
 - Bezeichner, die sich nur in Groß/Kleinschreibung unterscheiden, sind nicht erlaubt (z.B.: `identifier`, `Identifier`).

Standard-Datentypen (1)

Datentyp	Größe
short	≥ 16 Bit
unsigned short	≥ 16 Bit
long	≥ 32 Bit
unsigned long	≥ 32 Bit
long long (*)	≥ 64 Bit
unsigned long long (*)	≥ 64 Bit
float	≥ 32 Bit
double	≥ 64 Bit
long double (*)	≥ 79 Bit

(*) nicht auf allen Plattformen unterstützt.

Standard-Datentypen (2)

- Festkommazahlen: `fixed<size, precision>`
 - `typedef fixed<9, 2> Amount;`
 - 9 signifikante Stellen, 2 Nachkommastellen,
 - größter Wert: 9.999.999,99.
 - maximal sind 31 Stellen möglich.
 - intern wird mit 62 Stellen gerechnet.
- `char`: 8-Bit ASCII-Zeichen.
- `wchar`: 16-Bit Unicode-Zeichen.
- `string/wstring`: Zeichenketten
 - unbeschränkt: `typedef string Address;`
 - beschränkt: `typedef string<20> Name;`

Standard-Datentypen (3)

- **octet**: 8-Bit Datentyp zur Repräsentierung von *Binärdaten*.
 - Wert bleibt bei Übertragung unverändert.
- **boolean**: Datentyp mit den Werten **TRUE** und **FALSE**.
- **any**: Datentyp, der beliebige andere Datentypen aufnehmen kann (Standard- und benutzerdefinierte Datentypen).
 - Werte sind *typsicher*.
 - *Metainformation*: Tatsächlicher Typ kann zur Laufzeit ermittelt werden.

Typdefinitionen und Aufzählungen

- Mit `typedef` können *neue Typen* erzeugt werden.
 - Beispiele:
 - `typedef short AgeType;`
 - `typedef wstring<30> AddressType;`
 - `typedef octet BinaryData[1024];`
- *Aufzählungen* können mit `enum` erzeugt werden.
 - Beispiel:
 - `enum Color { red, green, blue, white, black };`
 - Typname muss angegeben werden.
 - Wert der Aufzählungskonstanten kann *nicht explizit definiert* werden.
 - Aufzählungskonstanten dürfen in einem Gültigkeitsbereich *nur einmal verwendet* werden.

Strukturen und Unions

- **struct** fasst eine *Menge benannter Komponenten* zu einer Einheit zusammen.
 - Beispiel:

```
struct Time {  
    short hour;  
    short minute;  
    short second;  
}
```
- **union** fasst eine *Menge alternativer benannter Komponenten* zu einer Einheit zusammen.
 - Beispiel:

```
union MonetaryValue switch (MonetaryFormat) {  
    case floatFormat:  
        float floatValue;  
    case fixedFormat:  
        fixed<15, 2> fixedValue;  
}
```
 - Nach Möglichkeit Unions vermeiden.
 - Typ des Diskriminators: **char**, **boolean**, Integer-Datentypen, Aufzählungen.
- Strukturen und Unions werden „*by Value*“ übergeben.
- Keine **typedefs** zur Definition von Strukturen und Unions verwenden.

Felder

- Die IDL unterstützt ein- und mehrdimensionale Felder mit beliebigen Elementtypen.
 - Beispiele: `typedef Color ColorVector[10];`
`typedef Field ChessBoard[8][8];`
- Zur Typdefinition muss `typedef` verwendet werden.
 - `Color ColorVector[10]; // Fehler!`
- Sämtliche Arraydimensionen müssen definiert werden.
 - `typedef Long OpenArray[][10]; // Fehler!`
- Index des ersten Elements ist sprachabhängig.

Sequenzen

- Mit `sequence` können *variabel lange Felder* definiert werden.
 - unlimitierte Sequenzen (*unbound*):
`typedef sequence<Color> ColorSequence;`
 - limitierte Sequenzen (*bound*):
`typedef sequence<short, 100> Grades;`
- Zur Definition muss `typedef` verwendet werden.
- Beliebige Elementtypen sind zulässig.
`typedef sequence<Edge> EdgeList;`
`typedef sequence<EdgeList> Graph;`
- Verschachtelte Schreibweise ist künftig unzulässig.
`typedef sequence<sequence<Edge> > Graph;`

Interfaces

- Wie Java-Interfaces sind IDL-Interfaces eine Sammlung von *Methoden-Signaturen*.

- Beispiel:

```
interface Article {  
    void    setPrice(in double price);  
    double getPrice();  
};
```

- Interfaces dürfen *keine Datenkomponenten* enthalten.
- Alle Methoden sind *öffentlich*.
- Bei Aufruf einer Interface-Methode wird ein *Request* an den (entfernten) Server geschickt.

Syntax von Interfaces

- Interfaces umfassen
 - Konstantendefinitionen,
 - Attributdefinitionen,
 - Typdefinitionen,
 - Methodendefinitionen (Operationen).
- Beispiel:

```
interface Article {  
    const string prefix = "A_";  
    typedef unsigned long QuantityType;  
    exception InvalidPrice { double price; }  
    readonly attribute QuantityType minStock;  
    void setPrice(in double price)  
        raises (InvalidPrice);  
    double getPrice();  
};
```

Semantik von Interfaces

- Interfaces können als Parameter-Typen verwendet werden:

```
interface Store {  
    void addArticle(in Article newArticle);  
    Article findArticle(in string id);  
}
```

- Ist der Parametertyp ein IDL-Interface, wird beim Aufruf eine *Referenz* übergeben.
- Aufrufe mit dieser Referenz bewirken einen entfernten Methodenaufruf (Request).
- Referenzen können auch *nil* sein.

Syntax von Wertetypen (val uetype)

- Wertetypen umfassen
 - Komponenten von Interfaces (Konstanten, Attribute, Typen, Methoden),
 - Datenkomponenten, die Zustand definieren
 - Sichtbarkeits-Attribute (**public**, **private**) legen fest, welche Methoden auf Datenkomponenten Zugriff haben.
- Beispiel:

```
val uetype ArticleVal {
    public string id;
    private double price;
    readonly attribute QuantityType minStock;
    void setPrice(in double price)
        raises (InvalidPrice);
    double getPrice();
};
```

Semantik von Wertetypen (val uetype)

- Wertetypen können wie Interfaces als Parameter von Methoden verwendet werden.

```
interface Store {  
    void addArticle(in ArticleVal newArticle);  
    ArticleVal findArticle(in string id);  
}
```

- Wertetypen werden (wie Strukturen, aber im Gegensatz zu Interfaces) „*by Value*“ übergeben.
- Empfänger erhält eine *Kopie*.
- Wertetypen können (anders als Strukturen) auch *nil* sein.
- Mit rekursiv definierten Wertetypen können komplexe Datenstrukturen (Bäume, ...) „*by Value*“ übergeben werden.

Methoden (Operationen)

- Beispiel:

```
interface Math {  
    double sqrt1(in double x);  
    void    sqrt2(in double x, out double res);  
    void    sqrt3(inout double x);  
}
```

- Parameter-Attribute („*directional attributes*“)
 - **in**: Parameter wird von Client an Server geschickt.
 - **out**: Parameter wird von Server an Client geschickt.
 - **inout**: Parameter wird in beide Richtungen geschickt.
- Attribute haben Einfluss auf die Größe eines Requests.
- Überladen von Methoden ist nicht erlaubt.
- Attribute haben Auswirkungen auf Speicherverwaltung.

Benutzerdefinierte Exceptions

- Beispiele:

- `exception Failure {};`
- `exception RangeError {
 unsigned long minVal;
 unsigned long maxVal;
};`
- `interface Unreliable {
 void canFail() raises (Failure);
 void canFailToo() raises (Failure, RangeError);
};`

System-Exceptions

- CORBA definiert 35 System-Exceptions
- System-Exceptions müssen nicht in `raises`-Klause angegeben werden.
- System-Exceptions haben folgende Struktur

```
enum completion_status {  
    COMPLETED_YES,  
    COMPLETED_NO,  
    COMPLETED_MAYBE  
};  
exception SystemExceptionName  
    unsigned long        minor;  
    completion_status    completed;  
}
```

Attribute

- Attribute definieren Merkmale eines Objekts.

```
interface Article {  
    enum CategoryType { business, ... };  
    readonly attribute Long minStock;  
    attribute CategoryType category;  
};
```

- IDL-Compiler generiert eine Methode zum Setzen (außer bei **readonly**) und zum Lesen des Attributwerts.
- Attribute definieren *keine Datenkomponenten*.
- Attribute entsprechen *Properties*.
- Attribut-Methoden können *keine Exceptions* werfen.
- Möglichst nur **readonly**-Attribute verwenden.

Module

- Module definieren einen eigenen *Gültigkeitsbereich* und sind mit C++-Namensräumen vergleichbar.
- Beispiel:

```
module Shop {  
    interface Article {  
        enum CategoryType { business, ... };  
    };  
    interface Store {  
        typedef sequence<Article> ArticleList;  
        long findByCategory(  
            in Article::CategoryType cat,  
            out ArticleList articles);  
    };  
};
```

- Mit Scope-Operator `::` kann auf andere Namensräume (Module, Interfaces, Strukturen, Unions) zugegriffen werden.

IDL: Beispiel (1)

```
module Shop {  
    typedef unsigned long QuantityType;  
    typedef string<10> IdType;  
  
    struct SupplierType {  
        IdType id;  
        string name;  
    };  
  
    interface Article { ... };  
  
    interface Store { ... };  
  
};
```


IDL: Beispiel (2)

```
interface Article {
    exception InvalidPrice {
        double price;
    };

    readonly attribute IdType      id;
    readonly attribute string      name;
    readonly attribute SupplierType supplier;

    void      setStock(in QuantityType value);
    QuantityType getStock();

    boolean sell (in QuantityType quantity);
    void      buy  (in QuantityType quantity);

    void      setPrice(in double price) raises (InvalidPrice);
    double getPrice();
};
```

IDL: Beispiel (3)

```
interface Store {
    typedef sequence<Article> ArticleList;

    ArticleList getAllArticles();

    Article findById(in string id);
    long findOutOfStock(out ArticleList articles);
    long findBySupplier(in IdType supplierId,
                       out ArticleList articles);

    void increasePrice(in ArticleList articles, in double perc);
    void decreasePrice(in ArticleList articles, in double perc);
};
```