In diesem Kursteil

- Modul : JavaIDL
  - Die Object Management Group
  - Die Objekt Management Architektur
  - Portable ORBs und iDL
  - IDL
    - Übersicht
    - IDL Konstrukte und Java Mapping
  - Zusammenfassung
- Kurs Zusammenfassung

Java in Verteilte Systeme

JavaIDL Interface Beschreibung.
und
CORBA

1.1. Kursübersicht

Diese Kurseinheit zum Thema ***Java in Verteilten Systemen*** ist eine Einführung in die Technoligien

- IDL - die Interface Definition Language der OMG (Object Management Group)
- CORBA - Common Object Broker Architecture der OMG

mit dem Ziel, Ihnen Wissen zu vermitteln, welche Sie befähigen wird, verteilte Anwendungen zu entwickeln. Dieser Kurs beschreibt Technologien, mit deren Hilfe Sie verteilte Anwendungen entwickeln können, basierend auf Java™ Application Programming Interfaces (API).

Voraussetzung für diesen Kurs sind Kenntnisse im Bereich *Java Programmierung, Objekt-Orientiertes Design und Analyse* und mindestens teilweise folgende praktischen Erfahrungen:

- Entwicklung von Java Applikationen
- Grundkenntnisse in Datenbanken

Sie sollten idealerweise auch bereits Erfahrungen in der objektorientierten Programmierung haben.

1.1.1. Lernziele

Nach dem Durcharbeiten dieser Kursunterlagen sollten Sie in der Lage sein

- unterschiedliche Technologien für die Programmierung verteilter Systeme in Java zu kennen und zu vergleichen
- einfache Java IDL Applikatione zu schreiben

1.2. Einführung in Java IDL

In diesem Modul

- Module JavaIDL
  - Einleitung
  - Object Management Group
  - Object Management Architecture
  - Portable ORB Core und JavaIDL
  - Wrapping von Legacy Code mit CORBA
  - Was ist IDL?
  - Wie funktioniert JavaIDL?
  - IDL Übersicht
  - IDL Grundlagen
  - Module Deklaration
  - Interface Deklaration
  - Operations und Parameter Deklarationen
  - Attribute Deklaration
  - Exceptions
  - Data Type Naming
  - IDL struct
  - Sequence
  - Array
  - enum und const
  - Praktische Übung - Java Interface Definition Language (JavaIDL)
  - Quiz
  - Zusammenfassung

1.2.1. Einleitung

JavaIDL ist eine Technologie zur Programmierung verteilter Systeme. JavaIDL wurde von Sun / JavaSoft entwickelt, um den Clients maximale Flexibilität bei der Integration in verteilte Systeme zu erlauben. Eines der Ziele ist es, den Client so flexibel zu bauen, dass die Technologie des Servers und der Kommunikation geändert werden kann - ohne dass der Client davon betroffen sein muss.

1.2.1.1. Lernziele

Nach dem Durcharbeiten dieser Unterlagen sollten Sie in der Lage sein:

- zu erklären, wer die OMG ist und in welchem Zusammenhang mit CORBA die OMG steht, sowie, welches typische Aufgaben der OMG sind.
- Stubs und Skeletons für eine CORBA Umgebung zu kreieren, mit Hilfe des `idlgen` Utilities.
- Java Objekte auf IDL abzubilden und mit Hilfe von JavaIDL (dem Java - IDL Mapping) zu beschreiben.

1.2.1.2. Referenzen

Teile dieses Moduls stammen teilweise oder ganz aus

- "A Note on Distributed Computing" `http://www.sunlabs.com/technical-reports/1994/abstract-29.html` (vollständig als PDF auf dem Server).
- ***Teach Yourself CORBA in 14 Days***
- Robert Orfali & Dan Harkey ***Client / Server Programming with JAVA and CORBA*** 2nd Edition (enthält sehr viele nette Bilder zum Thema).
- ***Design Patterns-Elements of Reusable Object-Oriented Software*** Gamma, Helm, Johnson und Vlissides (Addison-Wesley, 1995).

1.2.2. Die Object Management Group - OMG

Die Object Management Group (OMG) mit Hauptsitz in Framingham, Massachusetts, ausserhalb Boston (am berühmtesten Highway der Software Industrie: an dieser Umfahrungsautobahn von Boston finden Sie sogut wie alles was Rang und Namen in der Software Industrie hat) ist ein internationales nonprofit Konsortium, welches sich mit der Promotion der Theorie und der Praxis der Objekttechnologie (OT) für die Entwicklung verteilter Systeme. Ziel der OMG war ursprünglich, zu helfen, die Komplexität bei der Entwicklung solcher Systeme und deren Kosten zu reduzieren.

Innerhalb der Software-Standisierungsgremien spielt die OMG eine spezielle Rolle:

- es ist die grösste Standardisierungsgruppe der Welt, mit über 700 Mitgliedern weltweit.
- sie verkauft keine Software. Als nonprofit Organisation muss die OMG herstellerneutral sein. Natürlich versucht jeder Hersteller als Mitglied verschiedener Gremien, dominanten Einfluss zu gewinnen.
- Referenzmodelle und Architekturen, welche den Kern der OMG Standards bilden, umabhängig von irgend einer speziellen kommerziellen Implementation.

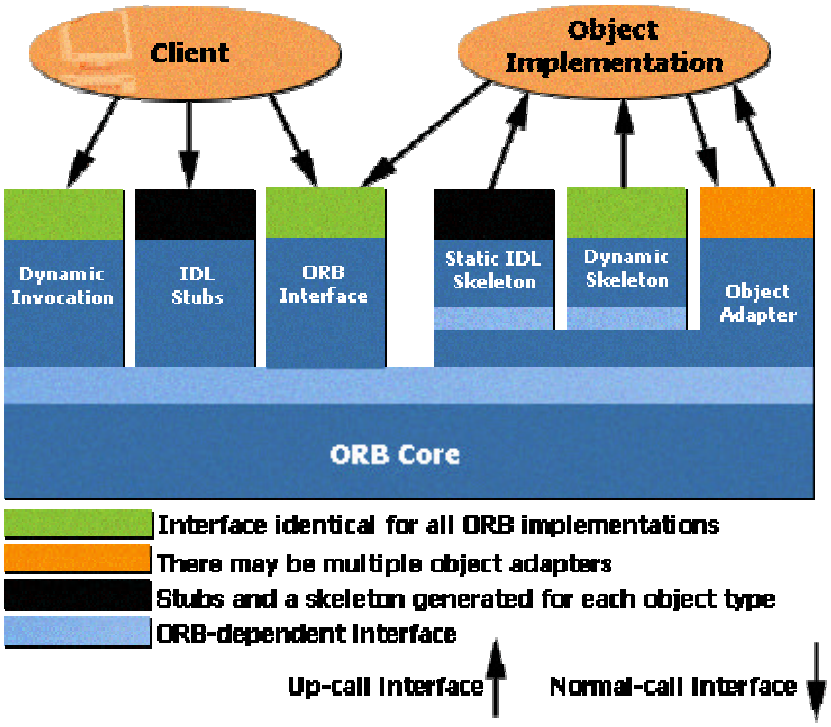
In der Startphase der OMG fokussierte sie sich darauf, die technische Infrastruktur aufzubauen und Werkzeuge zu definieren, welche die Interoperabilität garantieren: im Speziellen waren dies CORBA (Common Object Request Broker) und die OMG IDL (Interface Definition Language).

In den letzten Jahren befasste sich die OMG schwerpunktmässig mit der Verbreitung des Know Hows und der Technologien und deren Anwendungen in unterschiedlichen Anwendungsgebieten. Dies führte zu einer grösseren Reorganisation:
neben dem Platform Technical Committee wurde ein Domain Technical Commmittee gegründet.

1.2.3. Die Object Management Architektur

Der ORB ist das Kernstück des Referenzmodells. Er stellt die Kommunikationsinfrastruktr zur Verfügung, mit dessen Hilfe Objekte transparent Requests und Responses in einer verteilten Umgebung verarbeiten kann. Der ORB ist die Grundlage für den Bau von Applikationen mit verteilten Objekten und um die Interoperabilität zwischen Applikationen in heterogenen Umgebungen erreichen zu können.

Konsistent mit dem Fokus auf Interfaces spezifiziert CORBA nicht, wie der ORB implementiert wird: dies kann mit Hilfe eines Daemon Prozesses, einer Bibliothek oder mittels irgend einer Kombination geschehen.



Bemerkung
Welche Rolle der ORB konkret in einer Applikation spielt, wird im Standard nicht festgehalten. In einigen Applikationen kann es sein, dass der ORB nur bei der Initialisierung benötigt wird, um eine erste Verbindung aufzubauen. In anderen Fällen kann der ORB eine bleibende zentrale Rolle in der Applikation spielen.

Die Object Management Architektur wird bestimmt durch die folgenden Bestandteile, die Sie auch in der Skizze sehen können:

1.2.3.1. Static und Dynamic Invocation

CORBA definiert zwei Mechanismen für den Aufruf von Objektmethoden: statische und dynamische, Static und Dynamic Invocation.

Die statischen Aufrufe sind nur möglich, falls die Interfaces zur Zeit der Übersetzung der Applikationen bekannt sind.

Die dynamischen Aufrufe gestatten es dem Client die Interfaces zur Laufzeit durch Befragung des ORBs zu bestimmen. Diese Fähigkeit ist sehr mächtig und ist eine der vitalen Vorbedingungen, um flexible Systeme zu bauen, welche häufig verändert werden. Allerdings bezahlen Sie einen sehr hohen Overhead!

1.2.3.2. Interface Repository

Die Schlüsselkomponente dank dem die Dynamic Invocation möglich wird, ist das Interface Repository. In seiner einfachsten Form besteht das Repository aus einer Datenbank, oder beispielsweise einer Hashtabelle, in die der IDL Compiler die Interface Beschreibungen einträgt. Von der Architektur her gesehen ist das Interface Repository eine der kritischsten Komponenten in CORBA. Es enthält die Metadaten für die gesamte Objektföderation, prüft die Datentypen bei den Methodenaufrufen und überprüft die Konsistenz der Methodenaufrufe auch wenn diese mehrere ORBs betreffen.

In CORBA 2.0 wurden Standard Interfaces für den Zugriff auf das Interface Repository festgelegt.

1.2.3.3. Object Adapter

Im Allgemeinen gestattet ein Objektadapter Objekten mit inkompatiblen Interfaces miteinander zu kommunizieren. CORBA definiert den Objekt Adapter als eine ORB Komponente, welche den Objektimplementationen Aktivierung / Activation, Objektreferenzierung und zustandsabhängige Services zur Verfügung stellt.

Die folgenden Aktivitäten betrachtet man als zustandsabhängige Services:

- Registrierung von Implementationsobjekten / Serverobjekten beim ORB
- interpretieren und übersetzen von Objektreferenzen
- lokalisieren der Implementationsobjekte / Serverobjekte.
- aktivieren und deaktivieren der Implementationsobjekte
- Methodenaufrufe

Der Objektadapter unterhält einen Implementations- Repository für das Speichern von Informationen, welche die Objektimplementationen beschreiben. Irgendwo muss ja die Information über das Serverobjekt gespeichert werden!

Der ORB selber muss mindestens über einen Basic Object Adapter (BOA) verfügen, so wie er in CORBA definiert wird. Daneben kann der ORB aber auch noch weitere spezielle Objektadapter definieren. Beispielsweise könnte ein spezieller Objektadapter für die Handhabung der Persistenz definiert werden.

1.2.3.4. CORBA Services

Die CORBA Services unterstützen grundlegenden Funktionen, die bei der Implementation und dem Betrieb verteilter Applikationen benötigt werden, unabhängig von der spezifischen Anwendung.

Beispielsweise definiert der Lifecycle Service Interfaces, mit deren Hilfe Objekte kreiert, gelöscht, kopiert oder verschoben werden können. Wie die Objekte in der Applikation implementiert werden müssen, bleibt dabei offen.

Die Details dieses und weiterer Dienste wurde in einem OMG Dokument *CORBAservices* festgehalten.

1.2.3.5. CORBA Facilities

CORBA Facilities sind eine Sammlung von Diensten, die gemeinsam genutzt werden können. In der Architektur sind sie auf einer höheren als die Grunddienste angeordnet.

In der Regel geht es bei horizontalen Facilities um solche, die von mehreren Applikationen genutzt werden können. Beispiele wären: das Drucken oder electronic mail Dienste.

Vertikale Facilities betreffen spezielle Anwendungsbereiche, beispielsweise Finanzapplikationen oder Produktionsplanungsfacilities.

Mehrere Facilities werden in einem OMG Dokument CORBAfacilities festgehalten.

Zu den horizontalen oder generischen Facilities gehören folgende:

- **User Interface Facilities**
  Diese umfassen das Desktop Management, Skripting, Darstellung der Komponenten und User Support Facilities.

- **Information Management Facilities**
  Austausch von Daten, Informationen, Informationsmodellierung, Informationsspeicherung und Retrieval, Datenverschlüsselung, Zeitkoordination.

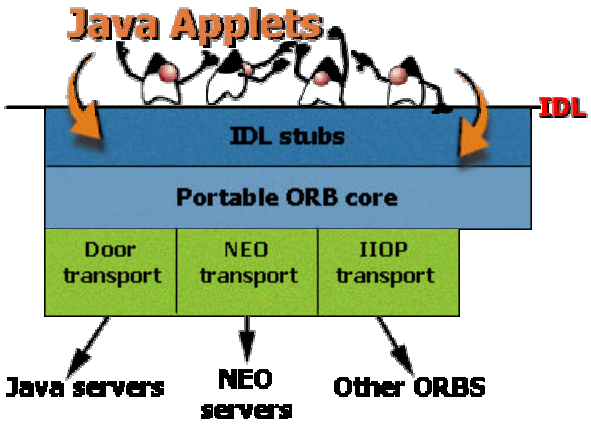
- **System Management Facilities**
  Event Management, Policy Management, Quality of Service Management, Scheduling Management, Security Facilities und Konsistenz der Daten.

- **Task Management Facilities**
  Workflow Management, Automation, OMG CORBA Agenten

Zu den vertikalen CORBAfacilities gehören zur Zeit einige wenige durch die OMG beschriebene Anwendungsbereiche, konkret handelt es sich dabei um: Rechnungswesen, Anwendungsentwicklung, CIM Computer Intergrated Management, Währungen, verteilte Simulation, Information Superhighways, Internationalisierung, Geo-Mapping, Oel und Gas Exploration und Produktion, Security und Telekommunikation.

1.2.4. Portable ORB Core und JavaIDL

JavaIDL ist eine CORBA ähnliche Implementation, bei der zusätzliche Möglichkeiten von Java, vom Java Objektmodell, berücksichtigt wurden. Die JavaIDL Architektur übersetzt IDL Dateien in flexiblen und portablen Programmcode, der mit dem Kernsystem, dem Core, zusammenarbeitet. Die Coredateien werden anschliessend auf mögliche Transportsysteme (IIOP, ...) abgebildet.



Sie sollten beachten, dass JavaIDL nicht für "Java Interface Definition Language" steht - JavaIDL ist der Name eines Produkts! NEO ist der Name der CORBA Implementation auf Sun Solaris. JOE steht für die CORBA Implementation von Sun in Java.

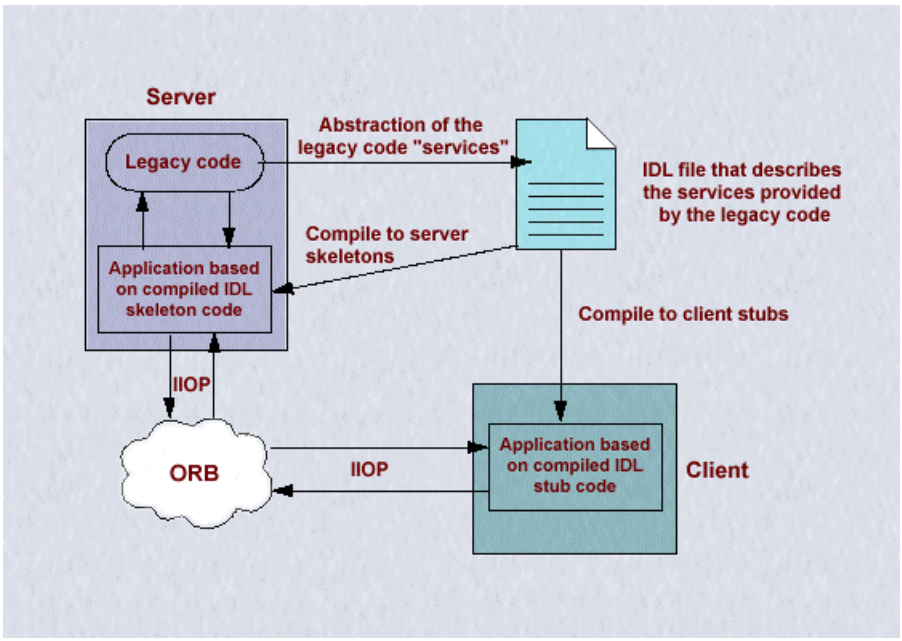
Verschiedene, eigentlich die meisten, Produkte im CORBA Umfeld unterstützen auch das Internet- Inter ORB Protokoll (IIOP).

Das obige Diagramm sollten Sie nicht überinterpretieren. Es soll lediglich veranschaulichen, dass unterschiedliche Transportprotokolle eingesetzt werden können. Dies wird durch die Aufteilung der Aufgaben in Transport Layer und Stub und Skeleton Layer möglich.

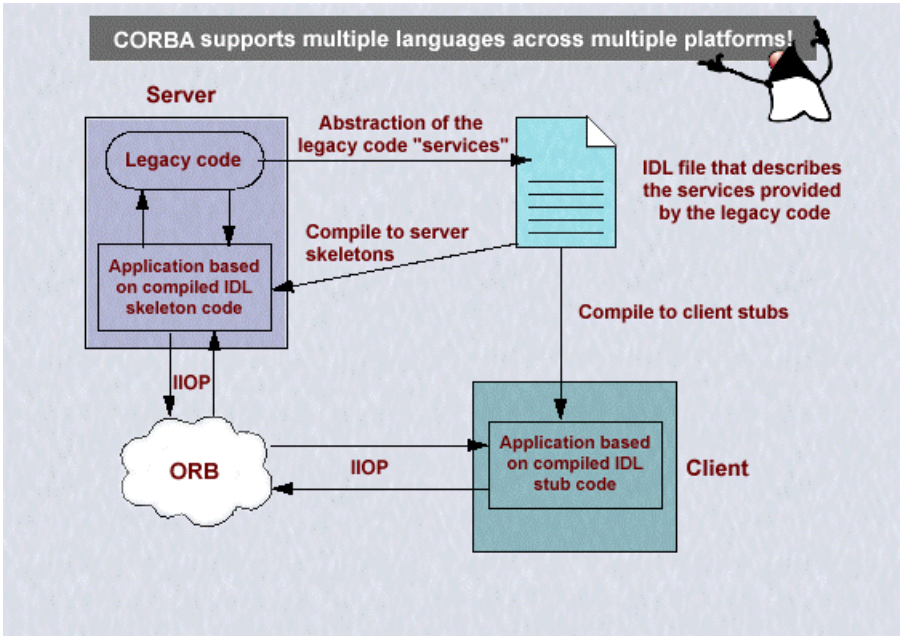
Sinnvollerweise werden Sie in Ihren Applikationen einen Standard wie IIOP einsetzen, um mehrere ORBs verbinden zu können.

1.2.5. Wrappen von Legacy Code mit CORBA

Eine der wichtigsten Fähigkeiten von CORBA ist die Unterstützung der unterschiedlichsten Programmiersprachen auf unterschiedlichsten Plattformen. Das folgende Schema zeigt, wie Legacy Code "gwrapped" werden kann, also in ein CORBA Umfeld integriert werden kann:

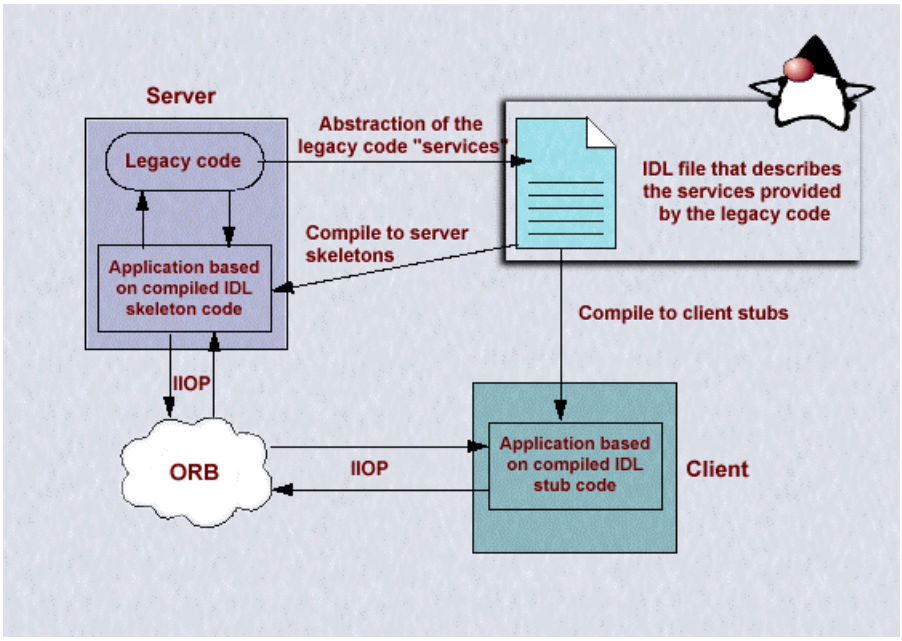


CORBA Anbieter kreierten die unterschiedlichsten Compiler für die unterschiedlichsten Plattformen und Programmiersprachen. Viele dieser Systeme dienen dem Wrappen, also demeinkapseln bestehender Anwendungen, der sog. Legacy Systeme, in die OT Welt.

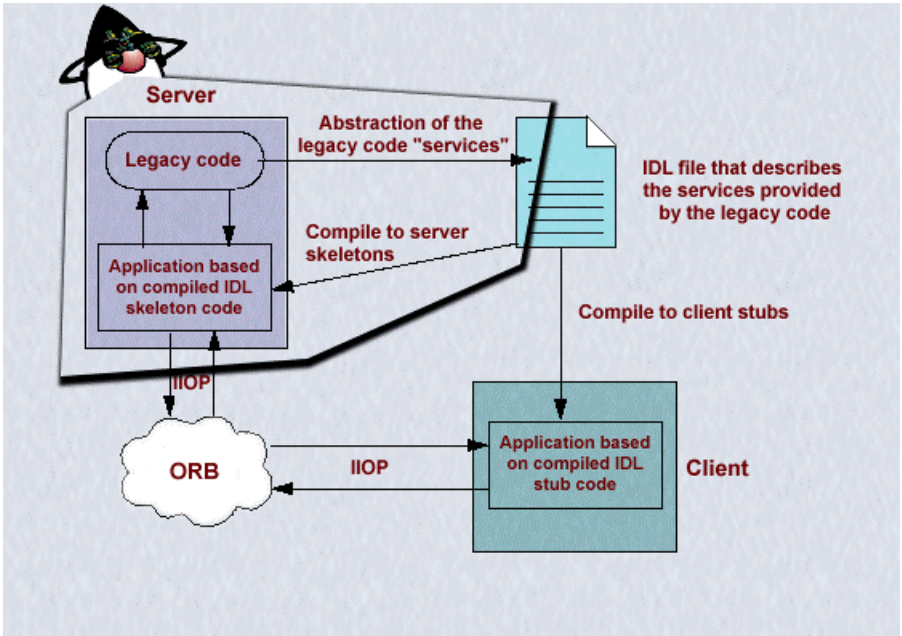


Falls Sie ein IDL Modell des Legacy Codes erstellen, stellt dies einen "Contract", eine Definition der möglichen Services der Serverimplementation (dem Legacy Code). In der

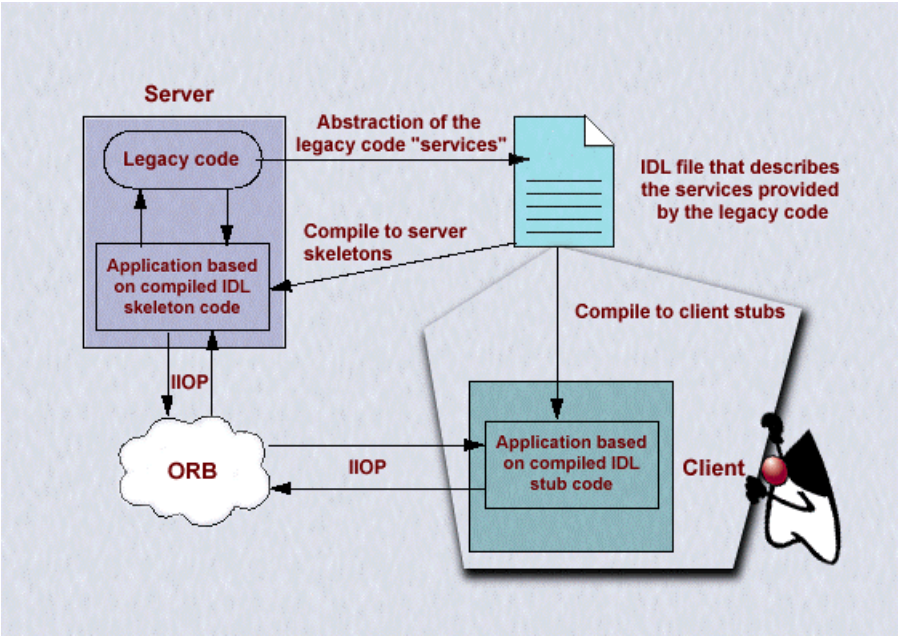
Regel werden Sie Kompromisse machen müssen; aber die Grundfunktionalität wertden Sie sicher mittels IDL anbieten können.



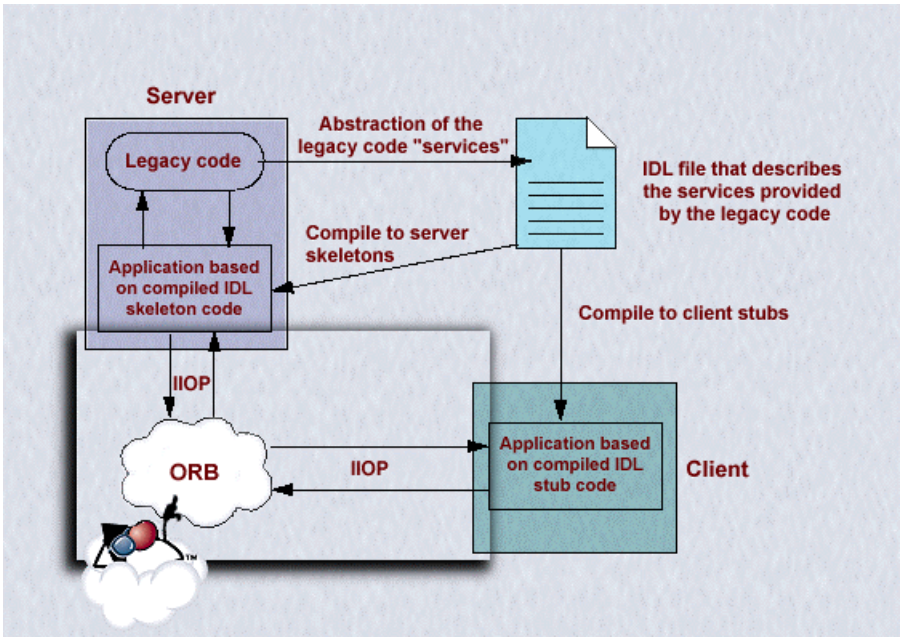
Auf der Serverseite wird IDL übersetzt und mit dem Legacy Code verbunden. Dies kann auf unterschiedliche Art und Weise geschehen: indem Legacy Bibliotheken aufgerufen werden, oder indem der Legacy Code direkt in die Skeletons, welche vom IDL Compiler generiert werden, eingebaut wird.



Auf der Client Seite wird die IDL Beschreibung mit der gewünschten Zielsprache übersetzt, beispielsweise für Java, und ab dann kann die Clientapplikation die in IDL definierten Methoden des Legacy Codes aufrufen.

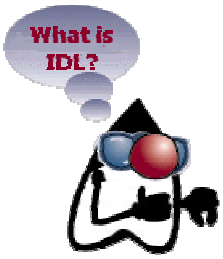


Auf beiden Seiten (Client und Server) ist die Kommunikation zwischen Stub Code und Skeleton Code und dem ORB transparent.



Die meisten ORB Implementationen offerieren einen direkten Verbindungsservice: der Client verlangt einen *'named service*' und der ORB kümmert sich um die Verbindung zum Server und das '*marshalling*' und '*unmarshalling*' des Requests.Nach dem erfolgreichen Aufbau der Verbindung, tritt der ORB zurück und "gibt" die Verbindung dem Client, so dass der Client direkt mit dem Server sprechen kann.

1.2.6. Was ist IDL?



JavaIDL ist ein System, mit dem Sie ein Set von remote Interfaces mit Hilfe der CORBA Interface Definition Language (IDL) beschreiben können. Der IDL Standard ist eine einfache Sprache, welche es Ihnen erlaubt, Interfaces für Objektsysteme zu deklarieren, ohne sich um Details der Implementation zu kümmern.

Beispiel:

```
interface MeinService {
      void operation ();
};
```

Diese Interface Definition wird an den IDL Compiler übergeben. Dieser generiert:

- Code für den Client, um einen Proxy zu kreieren, welcher eine Objektreferenz zum aktuellen `MeinService` Objekt enthält. Der Proxy Code enthält auch Stubs, mit deren Hilfe die Methode mit einer lokalen Semantik aufgerufen werden kann: es erscheint so, als ob das Objekt lokal vorhanden wäre.

- Code für die Serverseite (Skeletons), mit deren Hilfe die Interaktion zwischen Client Proxies und der aktuellen Implementation gemanaged werden kann.

Alle Programmteile, die für die Kommunikation benötigt werden, werden automatisch und vollständig durch den IDL Compiler generiert. Client und Server Code können jeweils unabhängig für eine bestimmte Zielsprache generiert werden, sofern der ORB Hersteller die entsprechende Programmiersprache und deren IDL Mapping unterstützt.

Zur Zeit sind verschiedene Sprachmappings implementiert, unter anderem für C, C++, Smalltalk und Ada '95. Die meisten Produkte unterstützen die Generierung des Codes für Server und Client Seite.

1.2.7. Wie funktioniert JavaIDL?

Ab JDK 1.4 wird ein neuer IDL Compiler mitgeliefert. Die folgende Beschreibung wird dann überholt sein!

JavaSoft's JavaIDL kreiert Stub und Skeleton Code aus einer IDL Beschreibung. Dazu wird der IDL zu Java Compiler verwendet, `idlgen`.

Synopsis
`idlgen [ Optionen ] Dateiname ...`

Beschreibung
Das `idlgen` Kommando übersetzt IDL Programmcode in Java Programmcode. Anschliesend wird mit dem `javac` Compiler dieser plus zusätzlicher Applikationscode in den Bytecode übersetzt.

Die IDL Deklarationen werden von IDL in Java übersetzt, mit Hilfe des IDL-2-Java Mappings. Der IDL Compiler generiert insbesondere Stubs und Skeletons mit deren Hilfe viele Funktionen, unabhängig vom Transport, implementiert und verwendet werden können:

- gemeinsame Dienste mit deren Hilfe Objekte für den Transport verpackt und nach dem Transport entpackt werden können (marshalling, unmarshalling).
- gemeinsame Dienste, mit deren Hilfe die aktuellen Methodenaufrufe ausgeführt werden können.
- gemeinsame Dienste, mit deren Hilfe die Resultate wieder passend eingepackt und ausgepackt werden können und dies speziell für die beim Client verwendete Programmiersprache.

`idlgen` kennt viele Optionen und Umgebungsvariablen, die Sie kontrollieren können.
Auf den folgenden Seiten finden Sie eine Zusammenstellung einiger Optionen und Steuermöglichkeiten für die Generierung der Stubs und Skeletons.

1.2.7.1. Umgebungsvariablen für `idlgen`

Der `idlgen` Compiler kann die folgenden Umgebungsvariablen auch direkt aus einem sogenannten Environment File lesen. Die Syntax für diese Datei ergibt sich aus der folgenden Beschreibung. Die Datei besteht aus einer Sequenz von Anweisungen in der Form:

```
idlSymbol ':' { attribute [ '=' value ] }* ';'
```

Eine Anweisung besteht also aus einem IDL Symbol, einem Doppelpunkt und einer beliebigen Anzahl Attribute und deren Werte.

Einige der gängigen Attribute:

`RepositoryIdentifier="`*identifier*"
: Der Identifier charakterisiert das Repository.

`RepositoryPrefix="`*prefix*"
: Requests können "prefix" verwenden, um damit ein bestimmtes Repository auszuwählen. Der Standard Präfix kann auch mit
`#pragma pntifier refix "requested prefix"`
in der IDL Spezifikation angegeben werden.

`RepositoryVersion="`*major.minor*"
: Spezifiziert major.minor als Repository Version.

`serverless`
: Diese Anweisung verlangt, dass das Objekt als "serverless" Objekt behandelt wird. Serverless Objekte werden als IDL Interfaces spezifiziert und übersetzt, ausser der Compiler kann kein Stubs und Skeletons generieren. Sie können die selbe Angabe mit
`#pragma serverless interfaceName`
am Anfang in der IDL Spezifikation machen.

`builtin`
: Diese Anweisung verlangt, dasss das Objekt als built-in Objekt behandelt wird. Built-in Objekte werden als Pseudo-Objekte behandelt: der Compiler generiert keinen Code dafür. Sie können die selben Angaben mit
`#pragma builtin interfaceName`
am Anfang in der IDL Spezifikation machen.

`javaClass="`*className*"
: Diese wird nur von built-in Objekten interpretiert. Sie bedeutet, dass das Objekt so behandelt wird, als ob es sich dabei um eine Java Klasse mit dem Klassennamen *className* handle. Die Java Klasse kann man auch mit:
`#pragma builtin interfaceName javaClass = className`
am Anfang der IDL Datei angeben.

1.2.7.2. idlgen Optionen

- `-j` *javaDirectory*- Spezifiziert, dass die generierten Java Dateien in das bezeichnete Verzeichnis geschrieben werden sollen. Dieses Verzeichnis ist unabhängig von der im Folgenden beschriebenen -p Option, falls diese überhaupt eingesetzt wird.

- `-p` Package - Spezifiziert, dass die generierten Java Symbole im angegebenen Package eingetragen werden sollen. Sie können als Trennzeichen entweder als Java Symbol spezifizieren (beispielsweise mit "." als Trennzeichen) oder aber als IDL Symbol (beispielsweise mit dem "::" Separator) oder als Verzeichnisname (mit dem "/" Trennzeichen). Package Angaben sind unabhängig von den Verzeichnisangaben in der obigen Option.

- `-J` Dateiname - Spezifiziert, dass eine Liste generierter Java Dateien in eine bestimmte Datei geschrieben werden soll. Diese Datei kann eingesetzt werden, um die generierten Dateien besser kontrollieren zu können. Die Datei kann auch eingesetzt werden, um effizient aufräumen zu können, da es eine Liste der generierten Dateien enthält.

- `-e` envfile - Spezifiziert eine Environment Datei, als Alternative zur `pragma` Anweisung.

- `-I` directory - Spezifiziert ein Verzeichnis, in dem nach `included` gesucht wird.

- `-D` symbol - Definiert ein Symbol, welches an den Preprozessor weitergeleitet wird..

- `-U` symbol - Definiert ein Symbol, welches gelöscht werden soll.

Die folgenden Optionen können ein- und ausgeschaltet werden. Ausgeschaltet werden die Optionen, indem Sie einfach ein `no` davor setzen.

- `-flist-flags`- Verlangt, dass alle `-f` Flags angezeigt werden. Standardwert ist `off`.

- `-fcpp`- Verlangt, dass der C++ Preprozessor benutzt wird. Standardwert ist `on`.

- `-fclient`- Verlangt, dass die Client Seite (Stub) generiert wird. Standardwert: `off`.

- `-fserver`- Verlangt, dass die Server Seite (Skeleton) generiert wird. Standardwert: `off`.

- `-fwrite-files`- Verlangt, dass die Java Dateien erzeiugt werden. Standardwert: `on`.

- `-fcaseless`- Verlangt, dass Identifiers ohne Klein / Grossschreibung verarbeitet werden.Standardwert: `off`.

- `-fverbose`- Ausgabe von Zusatzinformationen während der Übersetzung. Standardwert: `off`.

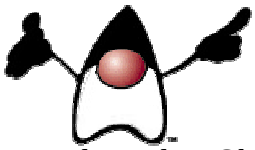
- `-fversion`- Anzeige der Version.Standardwert: `off`.

1.2.8. IDL Übersicht

Die Interface Definition Language (IDL) kann man sich als ein Werkzeug vorstellen, mit dem Kontrakte, Verträge zwischen Client und Server definiert werden. IDL ist aber, trotz aller Ähnlichkeit mit Standard Programmiersprachen, keine Sprache zur *Implementation* von Systemen. Schauen Sie sich das folgende Beispiel an:

```
module bank
{
      exception UngültigerBetrag
      {
            string Grund;
      };
      interface Konto
      {
            attribute string Name;
            attribute string KontoNummer;
            attribute double Kontostand;
            void Einzahlung(in double Betrag)
                  raises (UngültigerBetrag);
            void Abheben(in double Betrag)
                  raises (UngültigerBetrag);
      };
      interface Kontoabfrage: Konto
      {
            attribute double MonatsGebühr;
      };
};
```

Ein Modul definiert einen Betrachtungsbereich, einen semantischen Gültigkeitsbereich, ähnlich wie beim Konzept eines Subsystems. Ein Modul kann mehrere Interfaces enthalten. Ein Interface kann von anderen Interfaces erben (ähnlich wie beim Klassenkonzept in Java). Ein Interface kann Attribute , Ausnahmen und Operationen definieren. Attribute, wie auch Rückgabewerte sind von einem bestimmten Typus.



Attributtypen können Standard IDL Typen oder andere Objekte (inklusive Interfaces) sein,welche in IDL definiert werden. Eine Operation besitzt einen Rückgabewert, einen Namen (unter dem sie aufgerufen werden kann) und kann Ausnahmen werfen.

Jedes Argument wird mit einer

- Richtungsangabe versehen:
  - `in` für Eingabewerte, `out` für Ausgabewerte,`inout` für Wert- Resultat Argumente (Eingabe und Ausgabe)
- Typ und
- Name.

wobei weitere IDL Schlüsselworte möglich sind.
Die IDL Deklaration wird mittels idlgen in Java Interfaces übersetzt. Mit dazugehörigen Klassen werden dann die Applikationen (Client, Server) implementiert.

1.2.9. IDL Grundlagen

Das Format von IDL wurde durch die OMG festgelegt (Section 3 der Common Object Request Broker Architektur und Spezifikation). Die Abbildung auf Java finden Sie bei Sun Microsystems, zusammen mit der Spezifikation von JavaIDL:
http://java.sun.com/products/jdk/idl/index.html *IDL to Java Language Mapping Specification*oder dort in der Nähe, falls Sun ihren Web Site wieder mal reorganisiert hat.

IDL besitzt folgende Charakteristiken:

- die gesamte IDL Datei bildet einen Namensraum, in dem jedes Element eine bestimmte Rolle spielt.
- Bezeichner, Identifier, dürfen innerhalb eines Namensraumes lediglich einmal definiert werden
- IDL Identifier können in Grossbuchstaben oder Kleinbuchstaben speziffiziert werden (es wird nicht zwischen Gross- und Kleinschreibung unterschieden).
- Operationen innehalbe eines Namensraumes können nicht überschrieben werden, da es Programmiersprachen gibt, welche das Überschreiben von Methoden / Prozeduren verbieten.

Bemerkung
Das Java Language Mapping wurde von Sun, nicht von der OMG definiert. Sun hat das Mapping eingereicht, aber es liegt bei der OMG ab wann und ob überhaupt, dieses Mapping ein offizieller Bestandteil von CORBA ist.

Diese Anerkennung kann jederzeit passieren: Sie können sich direkt bei der OMG informieren: http://www.omg.org (wobei der Server oft nicht funktioniert).

Unabhängig davon halten wir uns an die von Sun publizierte Version von JavaIDL. Sollten Sie eine Anbindung von C/C++ an CORBA im Auge haben, sollten Sie auf die OMG Literatur zurückgreifen.

1.2.10. Module Deklaration

Das Modul Konstrukt wird benutzt, um die Gültigkeitsbereiche der IDL Identifiers zu definieren. Im folgenden Beispiel ist Bank der definierende Bereich:

```
1  module Bank {
2     interface Konto {
3     ...
4     };
5  };
```

In Java wird ein Modul auf ein Package abgebildet. Im Falle des obigen Beispiels wird Konto zu Bank.Konto.

1.2.11. Interface Deklaration

Ein Interface definiert einen IDL Grunddienst. Interfaces bestehen aus Attributen, Ausnahmen und Operationen, welche von einem Object durch einen Client verlangt werden. JavaIDL bildet IDL Interfaces auf Java Interfaces ab. Der Punkt ist, dass IDL Mehrfachvererbung von Interfaces unterstützt. In Java trifft dies für Interfaces ebenfalls noch zu; allerdings nicht bei Java Klassen.

Diese Abbildung reicht noch nicht aus, um dem Client die volle Proxy Funktionalität zur Verfügung zu stellen.

Der IDL Compiler generiert für jedes IDL Interface:

1. ein Java Interface, welches die Client Sicht des IDL Interfaces definiert und die Funktionalität des Client Interfaces festhält.
2. eine Java Klasse, welche dieses Interface implementiert und die Proxy Funktionalität zur Verfügung stellt.

Schauen wir und ein Beispiel an:

```
module Bank {
  exception UngenuegenderKontostand {
    float aktuellerKontostand;
  };

  interface Konto {
    attribute wstring name;
    readonly attribute unsigned long ssn;
    readonly attribute float kontostand;
    void abheben (in float betrag) raises (UngenuegenderKontostand);
    void einzahlen (in float betrag);
  };
};
```

Diese IDL Beschreibung generiert mit folgendem Befehl:

```
@echo off
%JAVA_HOME%bin\idlj -v -fall Bank.idl
Rem @echo Die generierten Dateien stehen im Unterverzeichnis Bank
```

folgende Java Interface Beschreibungen:

```
package Bank;

/**
* Bank/Konto.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Bank.idl
* Mittwoch, 18. April 2001 15.19 Uhr GMT+02:00
*/

public interface Konto extends KontoOperations, org.omg.CORBA.Object,
org.omg.CORBA.portable.IDLEntity
{
} // interface Konto
```

sowie die Interface Beschreibung der Operationen:

```
package Bank;


/**
* Bank/KontoOperations.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Bank.idl
* Mittwoch, 18. April 2001 15.19 Uhr GMT+02:00
*/

public interface KontoOperations
{
  String name ();
  void name (String newName);
  int ssn ();
  float kontostand ();
  void abheben (float betrag) throws Bank.UngenuegenderKontostand;
  void einzahlen (float betrag);
} // interface KontoOperations
```

und die Beschreibung der Ausnahme als finale Klasse:

```
package Bank;


/**
* Bank/UngenuegenderKontostand.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Bank.idl
* Mittwoch, 18. April 2001 15.19 Uhr GMT+02:00
*/

public final class UngenuegenderKontostand extends
org.omg.CORBA.UserException implements org.omg.CORBA.portable.IDLEntity
{
  public float aktuellerKontostand = (float)0;

  public UngenuegenderKontostand ()
  {
  } // ctor

  public UngenuegenderKontostand (float _aktuellerKontostand)
  {
    aktuellerKontostand = _aktuellerKontostand;
  } // ctor

} // class UngenuegenderKontostand
```

und die folgende Java Implementation (Stub):

```
package Bank;


/**
* Bank/_KontoStub.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Bank.idl
* Mittwoch, 18. April 2001 15.19 Uhr GMT+02:00
*/
```

```
public class _KontoStub extends org.omg.CORBA.portable.ObjectImpl
implements Bank.Konto
{
  // Constructors
  // NOTE:  If the default constructor is used, the
  //        object is useless until _set_delegate (...)
  //        is called.
  public _KontoStub ()
  {
    super ();
  }

  public _KontoStub (org.omg.CORBA.portable.Delegate delegate)
  {
    super ();
    _set_delegate (delegate);
  }

  public String name ()
  {
    org.omg.CORBA.portable.InputStream _in = null;
    try {
       org.omg.CORBA.portable.OutputStream _out = _request ("_get_name",
true);
       _in = _invoke (_out);
       String __result = _in.read_wstring ();
       return __result;
    } catch (org.omg.CORBA.portable.ApplicationException _ex) {
       _in = _ex.getInputStream ();
       String _id = _ex.getId ();
       throw new org.omg.CORBA.MARSHAL (_id);
    } catch (org.omg.CORBA.portable.RemarshalException _rm) {
       return name ();
    } finally {
        _releaseReply (_in);
    }
  } // name

  public void name (String newName)
  {
    org.omg.CORBA.portable.InputStream _in = null;
    try {
       org.omg.CORBA.portable.OutputStream _out = _request ("_set_name",
true);
       _out.write_wstring (newName);
       _in = _invoke (_out);
    } catch (org.omg.CORBA.portable.ApplicationException _ex) {
       _in = _ex.getInputStream ();
       String _id = _ex.getId ();
       throw new org.omg.CORBA.MARSHAL (_id);
    } catch (org.omg.CORBA.portable.RemarshalException _rm) {
       name (newName);
    } finally {
        _releaseReply (_in);
    }
  } // name

  public int ssn ()
  {
    org.omg.CORBA.portable.InputStream _in = null;
    try {
```

```
       org.omg.CORBA.portable.OutputStream _out = _request ("_get_ssn",
true);
       _in = _invoke (_out);
       int __result = _in.read_ulong ();
       return __result;
    } catch (org.omg.CORBA.portable.ApplicationException _ex) {
       _in = _ex.getInputStream ();
       String _id = _ex.getId ();
       throw new org.omg.CORBA.MARSHAL (_id);
    } catch (org.omg.CORBA.portable.RemarshalException _rm) {
       return ssn ();
    } finally {
        _releaseReply (_in);
    }
  } // ssn

  public float kontostand ()
  {
    org.omg.CORBA.portable.InputStream _in = null;
    try {
       org.omg.CORBA.portable.OutputStream _out = _request
("_get_kontostand", true);
       _in = _invoke (_out);
       float __result = _in.read_float ();
       return __result;
    } catch (org.omg.CORBA.portable.ApplicationException _ex) {
       _in = _ex.getInputStream ();
       String _id = _ex.getId ();
       throw new org.omg.CORBA.MARSHAL (_id);
    } catch (org.omg.CORBA.portable.RemarshalException _rm) {
       return kontostand ();
    } finally {
        _releaseReply (_in);
    }
  } // kontostand

  public void abheben (float betrag) throws Bank.UngenuegenderKontostand
  {
    org.omg.CORBA.portable.InputStream _in = null;
    try {
       org.omg.CORBA.portable.OutputStream _out = _request ("abheben",
true);
       _out.write_float (betrag);
       _in = _invoke (_out);
    } catch (org.omg.CORBA.portable.ApplicationException _ex) {
       _in = _ex.getInputStream ();
       String _id = _ex.getId ();
       if (_id.equals ("IDL:Bank/UngenuegenderKontostand:1.0"))
          throw Bank.UngenuegenderKontostandHelper.read (_in);
       else
            throw new org.omg.CORBA.MARSHAL (_id);
    } catch (org.omg.CORBA.portable.RemarshalException _rm) {
       abheben (betrag);
    } finally {
        _releaseReply (_in);
    }
  } // abheben

  public void einzahlen (float betrag)
  {
    org.omg.CORBA.portable.InputStream _in = null;
    try {
```

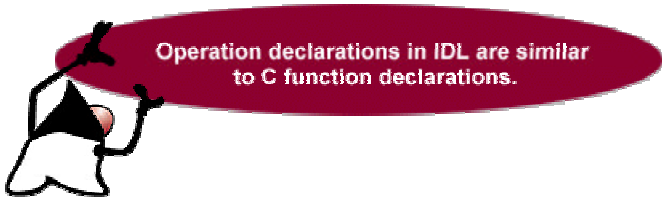
```
       org.omg.CORBA.portable.OutputStream _out = _request ("einzahlen",
true);
       _out.write_float (betrag);
       _in = _invoke (_out);
    } catch (org.omg.CORBA.portable.ApplicationException _ex) {
       _in = _ex.getInputStream ();
       String _id = _ex.getId ();
       throw new org.omg.CORBA.MARSHAL (_id);
    } catch (org.omg.CORBA.portable.RemarshalException _rm) {
       einzahlen (betrag);
    } finally {
        _releaseReply (_in);
    }
  } // einzahlen

  // Type-specific CORBA::Object operations
  private static String[] __ids = {
    "IDL:Bank/Konto:1.0"};

  public String[] _ids ()
  {
    return (String[])__ids.clone ();
  }

  private void readObject (java.io.ObjectInputStream s)
  {
     try
     {
       String str = s.readUTF ();
       org.omg.CORBA.Object obj = org.omg.CORBA.ORB.init
().string_to_object (str);
       org.omg.CORBA.portable.Delegate delegate =
((org.omg.CORBA.portable.ObjectImpl) obj)._get_delegate ();
       _set_delegate (delegate);
     } catch (java.io.IOException e) {}
  }

  private void writeObject (java.io.ObjectOutputStream s)
  {
     try
     {
       String str = org.omg.CORBA.ORB.init ().object_to_string (this);
       s.writeUTF (str);
     } catch (java.io.IOException e) {}
  }
} // class _KontoStub
```

Zudem werden für die Klassen `Konto` und die Ausnahmeklasse `UngenuegenderKontostand` je eine Helper und eine Holder Hilfsklasse generiert:

```
Konto.java
KontoHelper.java
KontoHolder.java
KontoOperations.java
UngenügenderKontostand.java
UngenuegenderKontostandHelper.java
UngenuegenderKontostandHolder.java
_KontoImplBase.java
_KontoStub.java
```

Bevor wir auf diese Dateien weiter eingehen und den Server bzw. Client bauen, schauen wir uns die Definition von IDL noch genauer an.

1.2.12. Operationen und Parameter Deklarationen



- Eine Operation besteht aus einem optionalen Attribut, dem `oneway` Attribut. Dieses Attribut gestattet Operationen beim Server aufzurufen, welche keine Antwort liefern - eine mögliche Methode, um das Blockieren von CORBA Aufrufen zu umgehen.

- Eine Operation besteht aus einer Angabe des Rückgabetyps der Operation oder `void`

- Eine Operation besteht aus einem Bezeichner, der die Operation in ihrem Gültigkeitsbereich benennt.

- Eine Operation besitzt keinen oder mehrere Parameter.

- Eine Operation besteht aus einer optionalen `raises` Klausel.

Hier ein Beispiel:
```
void abheben (in float betrag) raises (UngenuegenderKontostand);
```

Ein Operationsparameter kann ein IDL Basisdatentyp oder ein benutzerdefinierter Datentyp sein.

Der Parameter muss ein direktionales Attribut besitzen, welches den Kommunikationsservice Client- und Server-seitig informiert, in welche Richtung der Parameter verschoben wird. Die Richtungsangabe kann sein:

- `in` falls der Parameter vom Client zum Server verschoben wird.
- `out` falls der Parameter vom Server zum Client verschoben wird.
- `inout` falls der Parameter in beide Richtungen verschoben wird.

Da Java nur die wertmässige Parameterübergabe (*by-value*), keine Referenzübergabe kennt, können `out` und `inout` nicht direkt auf Basisdatentypen abgebildet werden. Daher werden die Basisdatentypen `int, float, boolean, ...`auf entsprechende Holder Klassen im CORBA Package abgebildet.

Zum Beispiel wird `int` auf `IntHolder` abgebildet, falls die Variable das `out` oder `inout` Attribut aufweist.

1.2.13. Attribut Deklarationen

Ein Interface kann neben Operationen auch noch Attribute besitzen. Eine Attribut Definition ist logisch äquivalent zu einer Deklaration einer Zugriffs- und einer Mutationsfunktion. Diese lesen und setzen Attributwerte.

Das optionale `readonly` Schlüsselwort zeigt an, dass es sich lediglich um eine Zugriffsfunktion handelt.

Beispiel:
betrachten wir einmal folgende Interfacebeschreibung in IDL

```
interface Konto {
    readonly attribute float kontostand;
    attribute long kontoNummer;

    void einzahlen(in float betrag);
    void transfer(inout float transferbetrag);
    float auszahlen(out float auszahlung);
};
```

Dieses liefert folgende Java Interface Beschreibung:

```
/**
* KontoOperations.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Konto.idl
* Mittwoch, 18. April 2001 18.56 Uhr GMT+02:00
*/

public interface KontoOperations
{
  float kontostand ();
  int kontoNummer ();
  void kontoNummer (int newKontoNummer);
  void einzahlen (float betrag);
  void transfer (org.omg.CORBA.FloatHolder transferbetrag);
  float auszahlen (org.omg.CORBA.FloatHolder auszahlung);
} // interface KontoOperations
```

Die Aufteilung der Kontonummer in eine Methode, welche einen Integer Wert zurück liefert und eine reine Aufrufmethode ist aus dem Beispiel klar ersichtlich.

Um die Abbildung der Basisdatentypen auch noch zu dokumentieren, wurden die zusätzlichen Methoden / (IDL) Operationen mit entsprechenden Parametern definiert.

1.2.14. Exceptions

Mit `raises` wird angegeben, dass eine Ausnahme geworfen werden kann, im Rahmen eines Aufrufes einer Operation.

Die Syntax dafür sieht folgendermassen aus:

```
raises (MeineException1 [, MeineException2 ...] )
```

Die Ausnahmen, welche durch eine Operation geworfen werden, können entweder operationsspezifisch sein, oder aber eine Standardausnahme. Diese Standardausnahmen brauchen nicht zwangsweise aufgelistet zu werden.

Eine Ausnahmedeklaration gestattet die Deklaration einer `struct` ähnlichen Datenstruktur, welche als Rückgabewert auftreten kann und das Auftreten einer aussergewöhnlichen Bedingung während der Ausführung des Aufrufes anzeigt.

Die Syntax einer solchen Deklaration sieht folgendermassen aus:

```
exception <identifier> { <Member>* }
```

Beispiel:

```
exception UngenuegenderKontostand {float aktuellerKontostand; };
```

Eine Exception besitzt einen Bezeichner (Identifier) und keine, einen oder mehrere Memberwerte. Falls eine Ausnahme bei einem Aufruf geworfen wird, dann ist ein Zugriff auf den Bezeichner möglich, um damit irgend eine Fehlerbehandlung einzuleiten.

Die Umsetzung in Java sehen Sie an folgendem generierten Interface mit obiger Ausnahme:

```
public interface KontoOperations
{
  String name ();
  void name (String newName);
  int ssn ();
  float kontostand ();
  void abheben (float betrag) throws Bank.UngenuegenderKontostand;
  void einzahlen (float betrag);
} // interface KontoOperations
```

Eine Java IDL Ausnahme wird wie oben ersichtlich auf eine Java Exception abgebildet. Diese selber ist wiederum eine Erweiterung von einer `org.omg.CORBA.UserException` Ausnahme:

```
public final class UngenuegenderKontostand extends
  org.omg.CORBA.UserException implements org.omg.CORBA.portable.IDLEntity {
  public float aktuellerKontostand = (float)0;
  public UngenuegenderKontostand ()  {
  } // ctor
  public UngenuegenderKontostand (float _aktuellerKontostand) {
    aktuellerKontostand = _aktuellerKontostand;
  } // ctor
} // class UngenuegenderKontostand
```

1.2.15. Bezeichnung der Datentypen

IDL enthält Konstrukte für die Benennung von Datentypen. Das `typedef` Schlüsselwort wird benutzt, um einen Datentyp mit einem Namen zu versehen.

Beispiele:

```
typedef     long        IDNummer
typedef     string      SSNummer
```

Die Java Programmiersprache kennt kein Sprachkonstrukt, welches der Typendefinition von IDL entspricht. Auch die Typendefinition für einfache Datentypen werden nicht direkt auf Java Konstrukte abgebildet.

Die folgende Zusammenstellung zeigt die Abbildung der IDL auf die Java Datentypen:

IDL	Java
`float`	`float`
`double`	`double`
`unsigned long`	`int`
`long long`	`long`
`long`	`int`
`short`	`short`
`unsigned long`	`int`
`unsigned short*`	`int`
`char`	`char`
`wchar`	`char`
`boolean`	`boolean`
`octet`	`byte`
`string`	`String (Klasse)`
`wstring`	`String`
`enum`	`int`
`fixed`	`java.mathBigDecimal`

IDL boolean Werte sind TRUE und FALSE; in Java werden daraus true und false.

1.2.16. IDL struct - Strukturen

Die IDL struct Anweisung wird eingesetzt, um Daten, die irgendwie zusammengehören, als eine Einheit verwendet zu können, beispielsweise als Parameter. Das `struct` Konstrukt wird auf eine Klasse abgebildet, welche Instanzen für die Felder und einen Konstruktor zur Verfügung stellt.

Am Besten sehen Sie wie der Mechanismus funktioniert, wenn Sie ein Beispiel anschauen:

```
module bank {
    struct KontoInfo {
        string name;
        float kontostand;
    };
};
```

und nun das Ganze in Java:

```
package bank;

public final class KontoInfo implements org.omg.CORBA.portable.IDLEntity
{
  public String name = null;
  public float kontostand = (float)0;

  public KontoInfo ()
  {
  } // ctor Konstruktor

  public KontoInfo (String _name, float _kontostand)
  {
    name = _name;
    kontostand = _kontostand;
  } // ctor Konstruktor mit Parametern

} // class KontoInfo

}
```

1.2.17. Sequenzen

Eine IDL Sequenz, `sequence`, ist ein eindimensionales Datenfeld mit einer maximalen Länge, die zur Übersetzungszeit festgelegt wird, sowie einer (aktuellen) Länge, welche zur Laufzeit festgelegt wird.

Eine Sequenz kann beschränkt oder unbeschränkt sein. Eine beschränkte Sequenz definiert ihre maximal erlaubte Länge.

Syntax:

```
sequence    <long>      UnbeschraenkteSeq;
sequence    <long, 10>  BeschraenkteSeq;
```

Eine so deklarierte Sequenz kann beispielsweise in Strukturen, `struct`, oder Vereinigungen, `union`, eingesetzt werden. Falls man eine Sequenz als Attribut oder als Parameter einer Operation verwendet werden soll, muss sie Teil einer Typendefinition, `typedef`, sein.

Hier ein Beispiel:

```
typedef     sequence    <long, 10>  longZehn;
            attribute   longZehn    vector;
```

Das folgende Beispiel zeigt einige weitere Aspekte der IDL Beschreibung:

```
module bank {
      struct KundenDetails {
            string Name;
            string Address;
      };
      typedef sequence<KundenDetails> UnbeschraenkteSeq;
      typedef sequence<KundenDetails, 5> BeschraenkteSeq;
};
```

Der IDL Compiler bildet diese Angaben auf eine Java Dateien ab. Schauen wir uns zuerst die Abbildung der Struktur in der Datei `KundenDetails.java` an:

```
package bank;

public final class KundenDetails implements
org.omg.CORBA.portable.IDLEntity
{
  public String Name = null;
  public String Address = null;

  public KundenDetails ()
  {
  } // ctor

  public KundenDetails (String _Name, String _Address)
  {
    Name = _Name;
    Address = _Address;
  } // ctor

} // class KundenDetails
```

Der IDL Compiler generiert auch eine Klasse pro Typendefinition, als 'Holder' Klasse.

```
package bank;

public final class BeschraenkteSeqHolder implements
org.omg.CORBA.portable.Streamable
{
  public bank.KundenDetails value[] = null;

  public BeschraenkteSeqHolder ()
  {  }

  public BeschraenkteSeqHolder (bank.KundenDetails[] initialValue)
  {
    value = initialValue;
  }

  public void _read (org.omg.CORBA.portable.InputStream i)
  {
    value = bank.BeschraenkteSeqHelper.read (i);
  }

  public void _write (org.omg.CORBA.portable.OutputStream o)
  {
    bank.BeschraenkteSeqHelper.write (o, value);
  }

  public org.omg.CORBA.TypeCode _type ()
  {
    return bank.BeschraenkteSeqHelper.type ();
  }
}
```

Sie sehen keinen Unterschied in der Holder Klasse! Aber in den Helper-Klassen treten Unterschiede auf, erwartungsgemäss.

1.2.18. IDL Arrays

IDL Arrays werden in Java genau so abgebildet wie beschränkte IDL Sequenzen. Die Grenze des Arrays, seine Dimension, wird beim Übermitteln, dem Marshalling, in einer IDL Operation überprüft. Falls man die IDL Dimension eines Arrays in der Java Umgebung benötigt, muss diese in der IDL Umgebung als Konstante definiert werden.

Beispiel:

```
module bank {
      struct KundenDetails {
            string Name;
            string Address;
      };
      typedef sequence<KundenDetails> UnbeschraenkteSeq;
      typedef sequence<KundenDetails, 5> BeschraenkteSeq;

      const long arrayGrenze = 10;
      struct namesListe {
            string namen[arrayGrenze];
      };

};
```

Und hier der generierte Java Programmcode:

```
package bank;

public final class namesListe implements org.omg.CORBA.portable.IDLEntity
{
  public String namen[] = null;

  public namesListe ()
  {
  } // ctor

  public namesListe (String[] _namen)
  {
    namen = _namen;
  } // ctor

} // class namesListe
```

zusätzlich wird eine Datei `arrayGrenze.java` mit den Angaben zu den Arraygrenzen:

```
package bank;

public interface arrayGrenze extends org.omg.CORBA.portable.IDLEntity
{
  public static final int value = (int)(10);
}
```

1.2.19. Die IDL `enum` und `const` Konstrukte

Die IDL enum (Aufzählung) Konstrukte werden in Java auf Klassen abgebildet mit je einer statischen finalen Variable pro Member des Auflistungstyps.

Beispiel:

```
...
      enum BankStandorte { Zuerich, Frankfurt, London, Paris, Tokyo,
Singapore, Boston, NewYork, Chicago };
...
```

Daraus wird folgende Java Datei:

```
package bank;

public class BankStandorte implements org.omg.CORBA.portable.IDLEntity {
  private        int __value;
  private static int __size = 9;
  private static bank.BankStandorte[] __array = new bank.BankStandorte
[__size];

  public static final int _Zuerich = 0;
  public static final bank.BankStandorte Zuerich = new
bank.BankStandorte(_Zuerich);
  public static final int _Frankfurt = 1;
  public static final bank.BankStandorte Frankfurt = new
bank.BankStandorte(_Frankfurt);
  public static final int _London = 2;
  public static final bank.BankStandorte London = new
bank.BankStandorte(_London);
  public static final int _Paris = 3;
  public static final bank.BankStandorte Paris = new
bank.BankStandorte(_Paris);
  public static final int _Tokyo = 4;
  public static final bank.BankStandorte Tokyo = new
bank.BankStandorte(_Tokyo);
  public static final int _Singapore = 5;
  public static final bank.BankStandorte Singapore = new
bank.BankStandorte(_Singapore);
  public static final int _Boston = 6;
  public static final bank.BankStandorte Boston = new
bank.BankStandorte(_Boston);
  public static final int _NewYork = 7;
  public static final bank.BankStandorte NewYork = new
bank.BankStandorte(_NewYork);
  public static final int _Chicago = 8;
  public static final bank.BankStandorte Chicago = new
bank.BankStandorte(_Chicago);

  public int value ()  {
    return __value;
  }

  public static bank.BankStandorte from_int (int value)  {
    if (value >= 0 && value < __size)
      return __array[value];
    else
      throw new org.omg.CORBA.BAD_PARAM ();
  }
```

```
  protected BankStandorte (int value)  {
    __value = value;
    __array[__value] = this;
  }
} // class BankStandorte
```

Bei Konstanten verhält es sich ähnlich: eine IDL Konstante, als `const` in IDL gekennzeichnet. Sie sind immer dann sinnvoll, wenn einfache Werte unveränderlich bleiben sollen. In Java werden die Konstanten auf eine Klasse abgebildet, welche `public final` ist.

In IDL

```
...
      const float pi=3.14159256;
...
```

und in Java (Datei `pi.java`):

```
package bank;

public interface pi extends org.omg.CORBA.portable.IDLEntity
{
  public static final float value = (float)(3.14159256);
}
```

Damit haben wir eine ganze Menge Dateien erzeugt, welche jeweils bestimmte Aspekte von IDL erläuterten. Dies ist jedoch nur ein kurzer Auszug aus der vollständigen IDLTOJAVA *IDL to Java Language Mapping Specifikation* der OMG.

Diese Spezifikation enthält auch viele weitere Beispiele, die teils sehr illustrativ sind. Falls Sie IDL Spezifikationen entwickeln wollen oder müssen, werden Sie auf diese Spezifikation zurück greifen müssen.

Wir haben auch jeweils nur einige der generierten Dateien angeschaut, speziell jene, die für das jeweilige Konzept typisch sind.

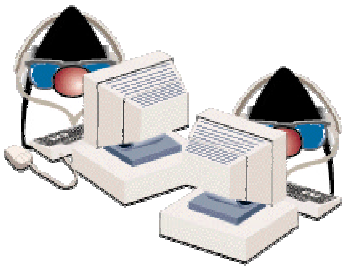
Zu beachten ist, dass in der Literatur oft noch von der (alten) IDLtoJava und dem idlgen Compiler die Rede ist. Diese wurden durch das offizielle Sprachmapping der OMG in Zusammenarbeit mit Sun abgelöst!

An Stelle von idlgen.exe wird nun idlj.exe zur Generierung der Java Dateien aus der IDL Beschreibung angewandt. Viele Umgebungsoptionen, die in idlgen wichtig waren und komplex beschrieben wurden, entfallen somit vollständig. idlgen existierte nie als Produkt, sondern lediglich als Beta / PreRelease.

1.2.20. Übung - Interface Definition Language Mapping für Java

In diesem Abschnitt schauen wir uns eine IDL Beschreibung für eine Fluggesellschaft an. Wir betrachten drei Interface Beschreibungen, je eine für die Kundeninformationen, die verfügbaren Flüge und Flugreservationsinformationen.

Die darin enthaltenen Informationen sind analog zur Beschreibung in der RMI Skizze für dieses bzw. ein analoges Informationssystem.



Die folgende Interface Beschreibung enthält drei Teile:

1. Kundeninformationen:
   KundenInfo definiert Zugriffsmethoden, um die Kundenliste abzufragen, individuelle Kundeninformationen zu verwalten und neue Kunden einzutragen
2. Fluginformationen:
   die Funktionalität des Flug Interfaces prüft, ob Flüge verfügbar sind, für bestimmte Abflug-Städte und Destinationen
3. Reservation:
   FlugReservation, die Reservationsschnittstelle beschreibt die Struktur einer Reservation und generiert eine Flugbestätigungsnummer

```
module Fluggesellschaft {
      // 1. Kundeninformationen
      struct KundenInfo {
            string kundenID;
            string nachName;
            string vorName;
            string strassenAdr;
      };
      typedef sequence <KundenInfo> KundenInfoListe;
      interface Kunde {
            // Lesen eines  Kundendatensatzes
            KundenInfo liesKunde (in string kundenID);
            // Lesen einer Liste der Kunden zu einem gegebenen Namen
            KundenInfoListe liesKundenListe (in string vorName,
                                             in string nachName);
            // Schreiben eines neuen Kundendatensatzes
            void schreibNeueKundenInfo (in string kundenID,
                                        in string nachName,
                                        in string vorName,
                                        in string strassenAdr);
            // Mutation eines bestehenden Datensatzes
            void mutiereBestehendeKundenInfo (in string kundenID,
                                              in string nachName,
                                              in string vorName,
                                              in string strassenAdr);
```

```
        // Produktion einer neuen ID für diesen neuen Kunden
        string produziereKundenID (in string vorName,
                                   in string nachName);
        // Produktion einer Zeichenkette für den Kundeninfo Container
        string produziereKundenString (in KundenInfo info);
};

// 2. Fluginformationen
struct FlugInfo {
        string flugID;
        string abflugStadt;
        string destinationStadt;
        string abflugDatum;
        string abflugZeit;
        // FC=First Class; BC=Business Class; EC=Economy Class
        long anzahlVerfuegbarerFCSitze;
        long anzahlVerfuegbarerBCSitze;
        long anzahlVerfuegbarerECSitze;
        string flugzeugID;
};

typedef sequence <FlugInfo> FlugInfoListe;

// unlimitiertes Arrays mit Abflug- und Destinations-Städten
typedef sequence <string> AbflugStadtListe;
typedef sequence <string> DestinationStadtListe;

// 3. Flugreservation

// Kunden Reservation Datensatz
struct KundenReservationInfo {
        string kundenID;
        string flugNummer;
        string bestaetigungsNummer;
        string serviceKlasse;
};

interface Flug {

        // Ausgabe der Liste der Abflug-Sädte
        //
        AbflugStadtListe liesAlleAbflugStaedte ();

        // Ausgabe der Ziel-Flughäfen
        DestinationStadtListe liesAlleDestinationen ();

        // Ausgabe einer Liste mit Start und Ziel und Datum
        FlugInfoListe liesVerfuegbareFluege (in string abflugOrt,
                                             in string destOrt,
                                             in string datum);

        // Mutation der Flug Info
        boolean mutiereAnzahlVerfSitze (in KundenReservationInfo resv);

        // Kreieren einer Zeichenkette für den Flug
        string produziereFlugString (in FlugInfo flug);
};

// Reservations Agent
interface Reservation {
```

```
            // Reservation durchführen
            void eingabeReservation (in string kundenID,
                                     in string flugNummer,
                                     in string bestaetigungsNummer,
                                     in string serviceKlasse);
            // Lies die Bestätigungs-Nummer
            string produziereBestaetigungsNummer();
      };
};
```

Selbsttestaufgabe 1

Inwiefern unterscheidet sich die obige IDL Definition vom Beispiel, das wir schrittweise entwickelten?[1]

Selbsttestaufgabe 2

Sie übersetzen eine IDL Beschreibung mit folgendem Befehl

```
idlj -fall Bank.idl
```

Was bedeutet die Option `-fall`?[2]

Selbsttestaufgabe 3

Im Folgenden sehen Sie drei Klassen, welche schematisch die Interface Beschreibung des Flugreservationssystems implementieren. Die Java Beschreibung ist in mehrerer Hinsicht unvollständig und könnte auch noch einige Tippfehler enthalten. Wichtig ist ein anderer Aspekt.

Ihre erste Aufgabe ist es, diese Listings genauer anzuschauen und zwar in Hinblick auf Exceptions.
Anschliessend, nach den Listings, sollten Sie die sehr leichte Frage problemlos beantworten können.

Listing 1 KundeImpl.java

```
// Implementationsklasse für den Customer Agent
// Implementation der IDL Operations
//
public class KundeImpl implements Kunden {

      private Database db = null;

      public KundeImpl (Database db) {
            this.db = db;
      }


      // lies einen Kundendatensatz
      public KundenInfo liesKunde(String kundenID)
      throws sunw.corba.SystemException {
            KundenInfo kunde = null;
            try {
                  kunde = db.liesKunde(kundenID);
```

[1] wir haben keine Exceptions definiert
[2] damit werden Client und Server Dateien generiert (Stubs, Helper, Holder....)

```
        } catch (Exception e) {
              System.out.println ("[liesKunde] Exception: " + e);
        }
        return kunde;
  }

  // lies ein Array mit Customer Records
  public KundenInfo[] liesKundenListe(String nachName, String vorName)
  throws sunw.corba.SystemException {
        KundenInfo [] kunden = null;

        try {
              kunden = db.liesKundenListe(vorName, nachName);
        } catch (Exception e) {
              System.out.println ("[liesKundenListe] Exception: " + e);
        }
        return kunden;
  }
  // schreib einen neuen Customer in die Datenbank
  public void schreibNeueKundenInfo(  String kundenID,
                                      String nachName,
                                      String vorName,
                                      String strassenAdr)
  throws sunw.corba.SystemException   {
        try {
              db.schreibNeueKundenInfo(     kundenID,
                                            nachName,
                                            vorName,
                                            strassenAdr);
        } catch (Exception e) {
        System.out.println ("[schreibNeueKundenInfo] Exception: " + e);
        }
  }
  // schreib einen bestehenden Customer in die Datenbank
  public void mutiereBestehendeKundenInfo(  String kundenID,
                                            String nachName,
                                            String vorName,
                                            String strassenAdr)
  throws sunw.corba.SystemException {
        try {
  db.mutiereBestehendeKundenInfo(     kundenID,
                                                  nachName,
                                                  vorName,
                                                  strassenAdr);
        } catch (Exception e) {
  System.out.println ("[mutiereBestehendeKundenInfo] Exception: " + e);
        }
  }
  // kreiere eine Zufalss Customer ID
  public String produziereKundenID(String nachName, String vorName)
  throws sunw.corba.SystemException {
        char FN = Character.toUpperCase(vorName.charAt(0));
        char LN = Character.toUpperCase(nachName.charAt(0));
        // was solls
        int ID = (int)(Math.random()*1000000);
        String kundenID = new String (new StringBuffer().
                                      append(FN).
                                      append(LN).
                                      append(Integer.toString(ID)));
        return kundenID;
  }
  // Kundeninfo als Zeichenkette
```

```
	public String produziereKundenString(KundenInfo info)
	throws sunw.corba.SystemException {
		String KundenString =	info.kundenID + "-" +
					info.nachName + "-" +
					info.vorName + "-" +
					info.strassenAdr;
		return KundenString;
	}
}
```

Nun die fast triviale Frage:
welche Exception wird bei solchen CORBA Anwendungen am meisten geworfen?[3]

a) ...corba.System.Exception
b) ...RemoteException
c) MsqlException

Und nun noch eine Skizze für den CORBA basierten Server. Der Programmcode funktioniert so nicht (Teile der Implementation fehlen)! Er dient lediglich der Illustration einzelner Konzepte!

Listing 2 FlugReservationsServer.java

```
// Skizze eines Flugreservations-Servers mit CORBA Anbindung

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
// weitere Imports
import ...;

class AirlineServer {
	static int servicePort = 4491;

	public static void main(String arg[]) {
		KundenRef		kunde = null;
		FlugRef			flug  = null;
		KundenReservationsRef	resv = null;

		// Usage prüfen
		if (arg.length <=0) {
			System.err.println("Usage:");
			System.err.println("java AirlineServer
					<hostname der Datenbank>");
			System.exit(1);
		}

		// DB
		Database db = null;
		try {
			// Datenbank Instanz
			db = new Database(arg[0]);
		} catch (Exception e) {
			System.err.println([Datenbank] Error : DB Instanz");
			System.exit(1);
		}

		//kreiere eine Implementation der Objekte und publiziere sie
		try {
```

[3] die corba Exception

```
                kunde =
                      KundenSkeleton.kreiereRef(new KundenImpl(db) );
                flug =
                      FlugSkeleton.kreiereRef(new FlugImpl(db) );
                resv =
                      KundenReservationSkeleton.kreiereRef(
                            new KundenReservationImpl(db) );
          } catch (Exception e) {
                System.err.println("[Implementation der Objekte]
                                        Exception");
          }

          // Publish :extern bekanntmachen
          try {
                // binde jedes Objekt an einen Service Namen
                // kreiere und initialisiere den ORB
                ORB orb = ORB.init(servicePort, null);
                // registriere die Referenzen beim ORB
                orb.connect(kunde);
                orb.connect(flug);
                orb.connect(resv);

                // bestimme die Wurzel des Naming Context
                org.omg.CORBA.Object objRef =
                       orb.resolve_initial_references("NameService");
                NamingContext ncRef = NamingContextHelper.narrow(objRef);

                // binde die Objekt Referenz im Naming Context
                NameComponent nc = new NameComponent("FlugKunde", "");
                NameComponent nc = new NameComponent("Flug", "");
                NameComponent nc = new NameComponent("Reservation", "");

                NameComponent path[] = {nc};
                ncRef.rebind(path, kunde);
                ncRef.rebind(path, flug);
                ncRef.rebind(path, resv);

                // warte auf Kunden;
                // sync ist ein dummy Objekt zur Synchronisation
                java.lang.Object sync = new java.lang.Object();
                synchronized (sync) {
                      sync.wait();
                }
          } catch (Exception e) {
                System.err.println("[FlugReservationsServer]ERROR:" + e);
                e.printStackTrace(System.out);
          }
     }
}
```

Selbsttestaufgabe 4

Noch eine triviale Frage: wo, in welchem Block, wird die Datenbank initialisiert, an das Programm angebunden?[4]

[4] Block 2 (Database ...)

1.2.21. Abschluss Quiz

In diesem Quiz werden einfache Fragen zu IDL und der ORB Architektur gestellt und auch gleich beantwortet. Die Idee de Quiz ist es, Ihnen eine Möglichkeit zu geben, die Konzepte noch einmal aufzufrischen.

1. Welche der folgenden Aussagen beschreibt den ORB am besten?
   a) Der ORB stellt eine Kommunikations- Infrastruktur zur Verfügung, mit deren Hilfe Objekte transparent Methoden anderer Objekte in einer verteilten Umgebung aufrufen können.

      Diese Antwort ist partiell richtig.

   b) Der ORB stellt eine Basis dar, um verteilte objektorientierte Applikationen zu entwickeln und eine Interoperabilität zwischen Applikationen in heterogenen Umgebungen zu erreichen.

      Diese Antwort ist partiell richtig.

   c) Der ORB kann als Daemon Prozess, Bibliothek oder als Kombination dieser beiden Konstrukte implementiert werden.

      Diese Antwort ist partiell richtig.

   d) Die obigen Antworten sind alle korrekt

      Ja das stimmt!

2. Welcher Mechanismus für den Aufruf von Objektmethoden bietet die grösste Flexibilität in einer sich dauernd ändernden Umgebung?

1. statische Aufrufe

   Denken Sie nochmals nach: statische Aufrufe / static Invocations sind limitiert auf Interfaces, welche zum Zeitpunkt der Compilation von Client und Server genau bekannt sind

2. dynamische Aufrufe

   Genau: die dynamic invocation gestattet es dem Kunden die Details des Interfaces zur Laufzeit durch Abfrage des ORBs zu bestimmen

3. ein Interface Repository

   Denken Sie nochmals darüber nach: ein Interface Repository ist eine wichtige Komponenten, ein Facility, mit dem Dynamic Invocation erst möglich wird.

4. Welche der folgenden unterstützt Grundfunktionen, um verteilte Applikationen zu benutzen und zu implementieren, welche unabhängig vom Anwendungsgebiet sind?
   a) CORBAServices

      Korrekt!

   b) CORBAFacilities

      Sind Sie sicher?
      CORBAFacilities sind eine Sammlung von gemeinsam genutzten Services auf einem Architekturlevel, der oberhalb den CORBAServices liegt.

   c) Objekt Adapter

      Sind Sie sicher?
      Objekt Adapter gestatten es den Objekten mit inkompatiblen Interfaces miteinander zu kommunizieren. CORBA definiert den Objekt Adapter als eine ORB Komponente, welche Objektreferenzen, Objektaktivierung und zustandsbezogene Services für ein Implementationsobjekt zur Verfügung stellt.

5. Java IDL idlj generiert Stubs und Skeleton Programmskelette aus einer IDL Beschreibung mit Hilfe von idlgen (alt) oder idlj (neu ab Java 2).
   a) trifft zu

      korrekt
      Der IDL Compiler idlj (oder idlgen) generiert Stubs und Skeletons, mit deren Hilfe gemeinsame Funktionlitäten eines beliebigen Transportsystems implementiert werden können.

   b) trifft nicht zu

      warum nicht?

1.2.22. Zusammenfassung Java IDL

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- zu erklären, wie die OMG und CORBA zusammenhängen
- die Funktionsweise eines ORBs zu erläutern und seine Dienste zu kennen
- Stubs und Skeletons zu generieren, mit Hilfe des idlj Compilers.
- Client und Server Implementationen zu einer in (Java) IDL spezifizierten Schnittstelle zu entwickeln.
- die Abbildungsregeln von IDL in Java zu kennen.

Den Punkt "Client und Server Implementationen" werden wir noch vertiefen (müssen).