In diesem Kapitel:

CORBA

-

Hetereogene Verteilte Systeme

1.1. Konzepte: CORBA und Verteilte Systeme

Die Common Object Request Broker Architecture (CORBA) von der Object Management Group stellt eine platformunabhängige, sprachenunabhängige Architektur für das Schreiben von verteilten, objektorientierten Applikationen zur Verfügung.

CORBA Objekte können auf der selben oder entfernten Maschinen implementiert sein und laufen. Java ist eine exzellente Sprache zum Schreiben von CORBA Applikationen, obschon ursprünglich C++, Smalltalk und Ada im Vordergrund der Objektentwicklung standen.

Diese Sprachen werden mit Hilfe eines sogenannten Language Mappings auf die Interface Definition Language (IDL) abgebildet. Mit Hilfe von IDL werden Module, Komponenten und Objekt-Schnittstellen beschrieben. Lange stand keine offizielle Abbildung von Java auf IDL zur Verfügung. Das hat sich in der Zwischenzeit geändert: die Sprache Java (inklusive Garbage Collection) wird auf OMGIDL abgebildet.

Diese Einführung soll Ihnen einen Einblick geben in die Funktionsweise von CORBA. Zudem lernen Sie CORBA Programme schreiben. Speziell werden Sie auch die Unterschiede zwischen RMI und CORBA kennen lernen.

An zwei Beispielen : Sun's Java 2 ORB Implementation von CORBA und Inprise's VisiBroker für Java, lernen Sie freie bzw. kommerzielle Produkte kennen.

1.1.1. Lernziele

Nach dem Durcharbeiten der CORBA Unterlagen und Beispiele sollten Sie in der Lage sein:

- die grundlegenden Strukturen einer CORBA Applikation zu verstehen
- Java Applikationen für ein CORBA Umfeld zu entwicklen
- Schnittstellen für verteilte Objekte mit Hilfe der IDL zu spezifizieren
- abzuschätzen, welche Teile der CORBA Spezifikationen für Ihre Arbeiten (Diplomarbeit, Semesterarbeit) wesentlich oder relevant sein könnte, falls überhaupt

Konkret, auch in Hinblick auf Prüfungen, können Sie:

- einfache CORBA Interfaces in IDL beschreiben
- die vom IDL to Java generierten Java Klassen in Ihre Programme einbauen
- CORBA Client Applikationen in Java erstellen
- verteilte CORBA Objekte in Java implementieren
- verteilte Objekte zur Laufzeit finden

1.2. Verteilte Applikation aus CORBA Sicht

1.2.1. Verteilung der Applikationen

CORBA Produkte liefern ein Framework für die Entwicklung und die Ausführung *verteilter Applikationen*. Aber warum sollte man überhaupt verteilte Applikationen entwicklen?

Wie wir sehen werden haben verteilte Applikationen in jeder Programmiersprache viele zusätzliche Probleme zur Folge. Auf der andern Seite ist man oft gezwungen aus einem der folgenden Gründe verteilte Applikationen einzusetzen:

- die *Daten*, welche die Applikation benutzt, sind verteilt
- die *Rechnerleistung* ist nur dezentral und verteilt verfügbar
- die *Benutzer* einer Applikation sind verteilt

1.2.1.1. Daten sind verteilt

Einige Applikationen müssen auf mehreren Rechnern lauffähig sein, weil die Daten dieser Applikationen auf unterschiedliche Rechner verteilt sind, aus administrativen oder andern Gründen. Unter Umständen kann die Anwendung remote auf die Daten zugreifen, darf sie aber nicht lokal abspeichern; unter Umständen müssen die Daten zentral gespeichert werden, aus historischen Gründen.

1.2.1.2. Rechnerleistung ist verteilt

Einige Applikationen benutzen mehrere Rechner, um mehrere Prozessoren benutzen zu können. Andere Applikationen nutzen mehrere Rechner, weil die unterschiedlichen Rechner unterschiedliche Applikationen anbieten. Schliesslich kann eine Applikation auf Grund gestiegener Anforderungen oder eines erhöhten Datenvolumens auf grössere Rechnerleitung angewiesen sein und die Skalierbarkeit der Rechnerleistung unterschiedlicher Systeme nutzen.

1.2.1.3. Benutzer sind verteilt

Einige Applikationen benutzen mehrere Rechner, weil die Benutzer mit dieser Applikation kommunizieren. Jeder Benutzer nutzt einen Teil der verteilten Applikation auf seinem Rechner und der Objekte, die typischerweise auf einem oder mehreren Servern installiert sind.

Eine typische Architektur für ein solches System sieht etwa folgendermassen aus:

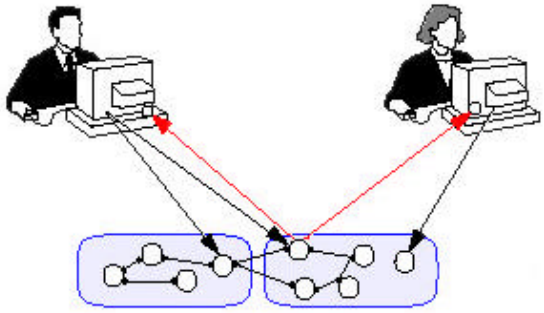


Abbildung 1 Verteilte Systeme : Objektsicht (verteilte Objekte)

Bevor man sich an das Design einer verteilten Applikation macht, ist es wesentlich sich mit einigen Fakten solcher Systeme vertraut zu machen,speziell mit den Gegebenheiten des verteilten Systems.

1.2.1.4. Grundlegende Tatsachen über Verteilte Systeme / Objekte

Entwickler verteilter Applikationen müssen verschiedene Aspekte berücksichtigen, welche bei der Entwicklung eines lokalen Programms, welches auf einem einzigen Rechner und auf nur einem Betriebsystem läuft, unwichtig sind. Die folgende Tabelle fasst einige dieser zusätzlichen Aspekte zusammen:

	lokal	**verteilt**
Kommunikation	schnell	langsam
Ausfälle	alle Objekte fallen gleichzeitig aus	Objekte fallem unabhängig voneinander aus
		das Netzwerk kann ausfallen
Gleichzeitigkeit	mit Hilfe mehrerer Threads	möglich
Sicherheit	ja (leichter machbar)	nein (aufwendiger machbar)

Die Kommunikation zwischen Objekten im selben Prozess ist wesentlich schneller als die Kommunikation zwischen Objekten auf unterschiedlichen Systemen. Dies impliziert, dass man es vermeiden sollte, verteilte Systeme zu bauen, in denen Objekte sehr eng gekoppelt werden und somit einen sehr hohen Kommunikationsbedarf haben. Eng gekoppelte Objekte sollten also wenn immer möglich auf dem selben Host untergebracht werden.

Falls zwei Objekte auf dem selben Rechner sind und der Prozess, in dem die Objekte instanziert wurden, stirbt, dann fallen beide Objekte aus.

Falls beide Objekte auf unterschiedlichen Systemen aktiv sind, muss sich der Designer darüber Gedanken machen, was passiert, wenn eines der Objekte im System ausfällt.
Bei Ausfall eines Objekts kann das andere (entfernte) Objekt andere Aktivitäten ausführen.

Lokale Systeme bestehen in der Regel aus einem Prozess, der einen oder mehrere Threads besitzt.

Verteilte Systeme bestehen immer aus mehreren Prozessen, die ihrerseits auch noch mehrere Threads besitzen können. Nebenläufigkeit und deren Probleme, Zugriffskonflikte, Aushungern und Verklemmungen sind daher in verteilten Systemen zentrale Fragen, da ein Server in der Regel mehr als einen Client besitzt.

Wenn zwei Objekte in ein und demselben Prozess aktiv sind, werden Sicherheitsfragen eher leicht zu lösen sein.

Bei verteilten Systemen muss die Sicherheit und Authentisierung der gegenseitigen Objektzugriffe klar geregelt sein.

1.2.1.5. Verteilte Objektsysteme

Verteilte Objektsysteme sind verteilte Systeme, in denen alle Grössen als Objekte modelliert sind. Verteilte Objektsysteme sind ein populäres Paradigma für Objekt-orientierte verteilte Applikationen. Da die Applikationen als kooperierende Objekte modelliert sind, lassen sie sich auf recht natürliche Art und Weise auf verteilte Systeme abbilden.

Obschon also verteilte Objektsysteme eher etwas Natürliches sind, müssen die obigen Punkte (Probleme in verteilten Systemen) stets berücksichtigt werden. Prozessgrenzen spielen immer eine Rolle und haben direkten Einfluss auf Systemdesigns.

1.3. Konzepte für Verteilte Applikationen

Wir haben bereits einige sehr unterschiedliche Architekturmuster kennen gelernt:
1. Layer : OSI Layer, GUI-BusinessLogik-Datenspeicherung,
2. Model-View-Controller : alle GUI Applikationen, die Swing benutzen
3. Broker : ORB, CORBA, Nameserver

In diesem Abschnitt geht es um eine Art "Roundup", ein Blick auf diese Architekturen aus etwas Entfernung.

1.3.1. Multi-Tiered Applikationen

(zur Information : tier = Reihe; Sitzreihe)
Traditionelle Firmenapplikationen sind zum grössten Teil immer noch monolithisch, also alles in einem riesigen Programm: Programmlogik, Datenbankzugriffe, Oberflächen...

Diese Applikationen lassen sich nur sehr schwer ablösen und auch sehr schwer warten, da bei Tests in der Regel riesige Systeme getestet werden müssen.

Im Unterschied dazu siht die Welt im CORBA, RMI, ... Umfeld wesentlich anders aus:
man spricht von einer *Multitired Architektur.*

Als Zwischenschritt setzen einige Firmen auf eine dreistufige Architektur. Bei dieser Three-Tiered Architektur trennt man die Bereiche GUI, Business Logic und Datenspeicher, analog zum MVC Entwurfs/Architekturmuster.

Modernere Applikationen gehen einen Schritt weiter und verwenden noch weitere Tiers:

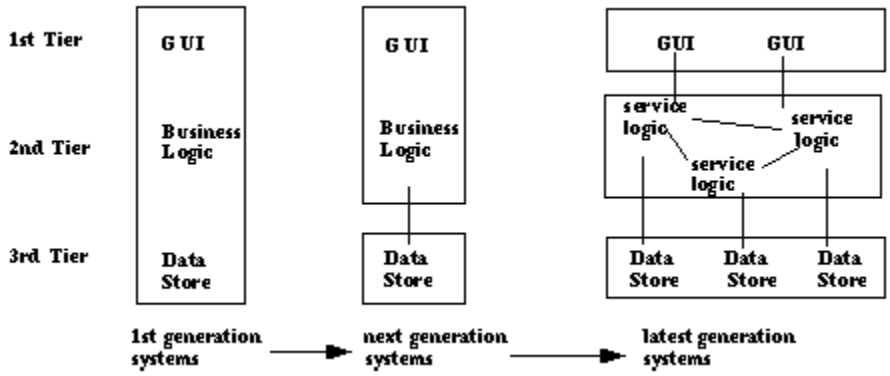


Abbildung 2 Evolution der Tier Architektur

1.3.2. User-Interface Tier

Der Benutzer kommuniziert mit dem User Interface Layer oder Tier. Dies ist die Ebene der grafischen Oberflächen. Dieser Tier kann oft ganz oder teilweise auf einen Client ausgelagert werden. Dadurch wird die Kommunikation drastisch reduziert. Ein Beispiel sind die Web Browser.

Der User Interface Tier verwendet in der Regel Methoden und Objekte des Business Logik Tiers und ist somit ein Client der Business Logik Implementation.

1.3.3. Service (Server) Tier

Der Service oder Business Logik Layer besteht aus serverseitigen Programmen, mit denen der Client kommunizieren kann.Der Business Logik Layer besteht aus Business Objekten - CORBA Objekten, die logische Business Funktionen ausüben, wie zum Beispiel Lagerverwaltung, Budget, Verkaufstatistik, Rcehnungswesen und viele andere. Alle diese Objekte benötigen ihrerseite den Datenspeicher Tier / Layer bzw. dessen Objekte.

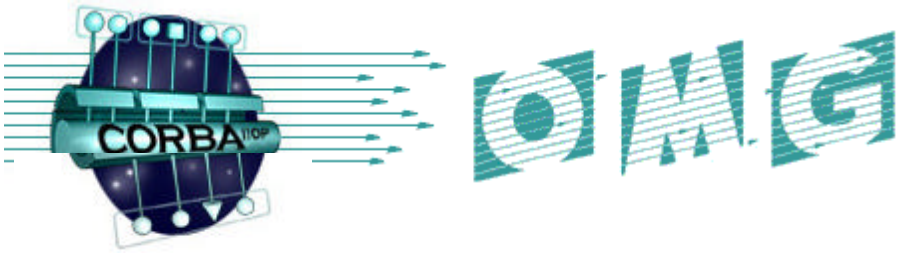
1.3.4. Data Store (Database) Tier

Der Datenspeicher Layer trägt die Verantwortung für die Kapselung der Datenbankzugrifffsroutinen. Dieser Layer, dieser Tier, arbeitet direkt mit den Datenbank Routinen zusammen, also den Zugriffsroutinen konkreter Datenbanksysteme, also zum Beispiel mit Hilfe von SQL.

Weiteres Vorgehen:

- zuerst behandeln wir den generellen Fall, also das Schema zur Entwicklung einer CORBA Applikation
- dann entwickeln wir ein konkretes Beispiel
- und betrachten dann typische Muster für CORBA (Persistente Datenspeicherung, Callback, ...)

1.4. Die OMG CORBA Spezifikation



Web Link : http://www.omg.org

1.4.1. Was ist CORBA?

CORBA, oder Common Object Request Broker Architecture, ist eine Standard Architektur für verteilte Objektsysteme. Mit Hilfe von CORBA können verteilte, heterogene Objekte interoperieren.

1.4.1.1. Die OMG

Die Object Management Group (OMG) ist verantwortlich für die Definition von CORBA. Die OMG umfasst über 700 Firmen und Organisationen, darunter alle bekannten Anbieter und Entwickler von verteilten Objekttechnologien, inklusive Platformen, Datenbanken, sowie Applikationsanbieter und Software Werkzeug Entwickler sowie Konzerninformatik-Vertreter.

1.4.1.2. CORBA Architektur

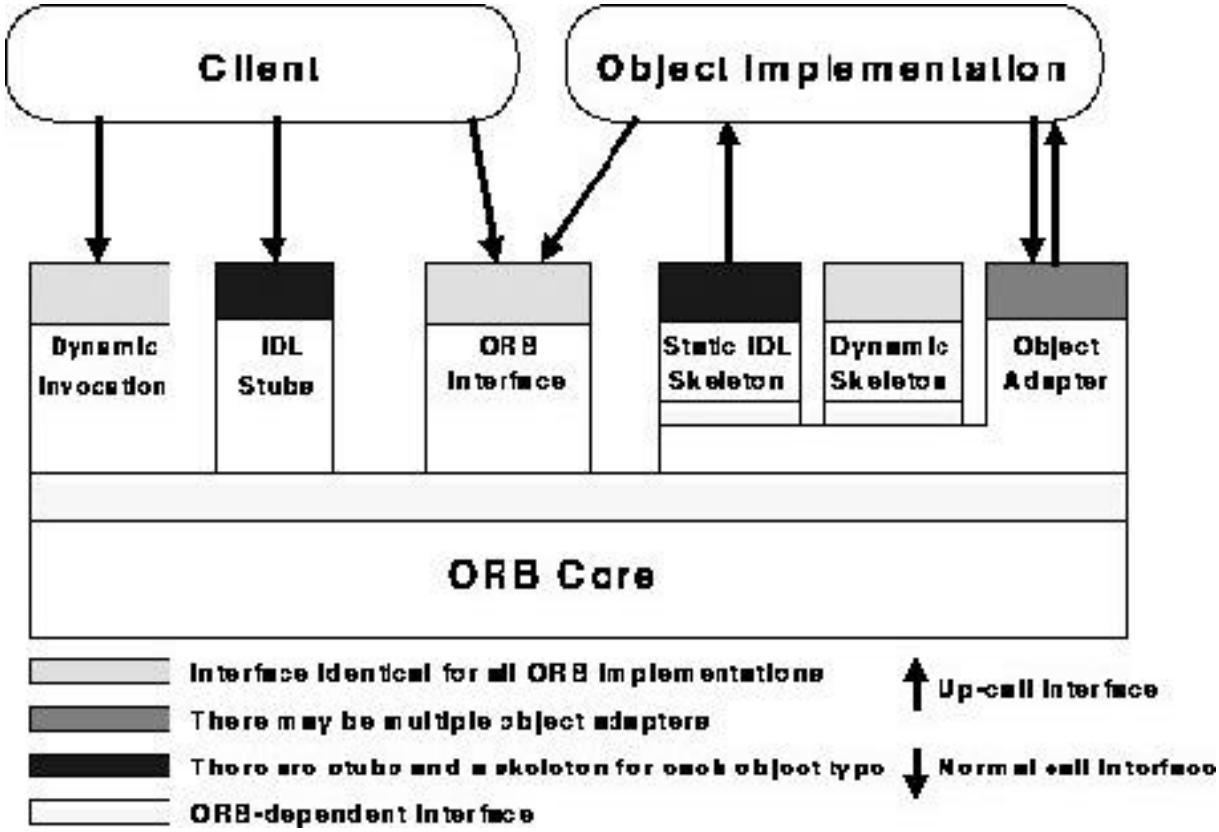


Abbildung 3 Die Common Object Request Broker Architecture

CORBA definiert eine Architektur für verteilte Objekte. Basis des CORBA Paradigmas ist die Serviceanfrage an ein verteiltes Objekt. Alles andere innerhalb von CORBA lässt sich auf dieses Paradigma zurück führen.

Alle Services werden mit Hilfe von *Interfaces* angeboten. Diese Schnittstellen werden mit Hilfe der OMG Interface Definition Language (IDL) definiert. Verteilte Objekte werden mit Hilfe einer Objekt Referenz identifiziert, der IDL Interfaces zugeordner sind.

Die folgende Abbildung illustriert den prinzipiellen Ablauf der Kommunikation zwischen dem Client und der Objekt Implementation:

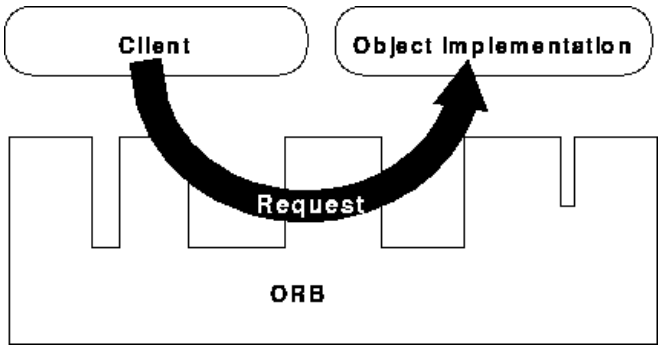


Abbildung 4 Client - ORB - Server Kommunikation

Der Client besitzt eine Referenz eines verteilten Objekts (besser: eines entfernten Objekts).

Der Objekt Request Broker (ORB) liefert die Anfragen an das "Server" Objekt und gibt allfällige Rückgaben an den Client zurück.

1.4.1.3. Der ORB

Der ORB ist ein verteilter Diensr, der die Aufrufe von Methoden entfernter Objekte an diese weiterleitet:

- zuerst wird das entfernte Objekt gesucht und lokalisiert
- dann wird die Anfrage an das Objekt weitergeleitet.
- Der ORB wartet auf allfällige Resultate
- und kommuniziert diese an den Kunden zurück

Der ORB implementiert die Lokalisation des entfernten Objekts transparent:
unabhängig davon, ob sich das Objekt lokal oder irgendwo im Netzwerk befindet wird immer der selbe Mechanismus eingesetzt. Der Client erkennt also keinen Unterschied.

Zudem implementiert der ORB diesen Mechanismus (Programmier-) Sprachen unabhängig. Es kann also durchaus sein, dass der Client und das remote Objekt in unterschiedlichen Programmiersprachen geschrieben wurden. Der ORB ist für die Umsetzung der einzelnen Sprachkonzepte zuständig. Dies geschieht mit Hilfe der sogenannten Language Bindings, also Abbildungen auf eine dem ORB bekannte "Sprache".

Hier ein Auszug aus der "IDL to Java Language Mapping Specification" der OMG:

IDL Type	Java type	Exceptions
boolean	`boolean`	
char	`char`	CORBA::DATA_CONVERSION
wchar	`char`	CORBA::DATA_CONVERSION
octet	`byte`	
string	`java.lang.String`	CORBA::MARSHAL CORBA::DATA_CONVERSION
wstring	`java.lang.String`	CORBA::MARSHAL CORBA::DATA_CONVERSION
short	`short`	
unsigned short	`short`	
long	`int`	
unsigned long	`int`	
double	`double`	
fixed	`java.math.BigDecimal`	CORBA::DATA_CONVERSION

dabei wird die C++ Notation für Packes verwendet (CORBA::).

1.4.1.4. CORBA als Standard für Verteilte Objektsysteme

Eines der Ziele der CORBA Spezifikation ist, dass Clients und Objektimplementationen portabel sind. Die CORBA Spezifikation definiert ein Application Programmer's Interface (API) für Clients verteilter Objekte und ein API für die Implementierung eines CORBA Objekts.

Damit soll gewährleistet werden, dass Programme, die für ein CORBA Produkt, eine konkrete CORBA Implementation, geschrieben wurden, mit möglichst wenig oder idealerweise keinem Aufwand an die CORBA Implementation eines andern Herstellers angepasst werden können.

Die Realität sieht leider anders aus: in der Regel sind die CORBA Clients portabel; die Objekt Implementationen sind dies in der Regel nicht. Oft werden bei kommerziellen Produkten auch Erweiterungen angeboten (zum Beispiel in ORBIX), die sehr brauchbar sind, aber eben nur von einem Hersteller (hier ORBIX) angeboten und unterstützt werden.

CORBA 2.0 erweiterte die Interoperabilität, speziell ein Netzwerk Protokoll, genannt *IIOP*, das Internet Inter-ORB Protokoll. Damit lassen sich die verschiedenen CORBA Produkte der unterschiedlichen Hersteller miteinander verbinden. IIOP funktioniert über das Internet oder präziser ausgedrückt, mit Hilfe von TCP / IP.

Interoperabilität ist in einem verteilten System wichtiger als Portabilität. IIOP wird auch in andern Systemen, die nicht auf CORBA basieren, eingesetzt.

IIOP wird auch für eine Version des Transport Protokolls von Java RMI ("RMI over IIOP") eingesetzt. Da EJB (Enterprise Java Beans) auf der Basis von RMI definiert sind, können auch EJBs IIOP benutzen. RMI over IIOP ist in JDK 1.3 enthalten und wurde von IBM entwickelt. Die Unterlagen finden Sie bei http://java.sun.com/products/rmi-over-iiop .

Verschiedene Applikationsserver auf dem Markt verwenden ebenfalls IIOP, verwenden aber sonst keine CORBA APIs. Da alle IIOP benutzen, können diese Programme untereinander und mit CORBA API's kommunizieren.

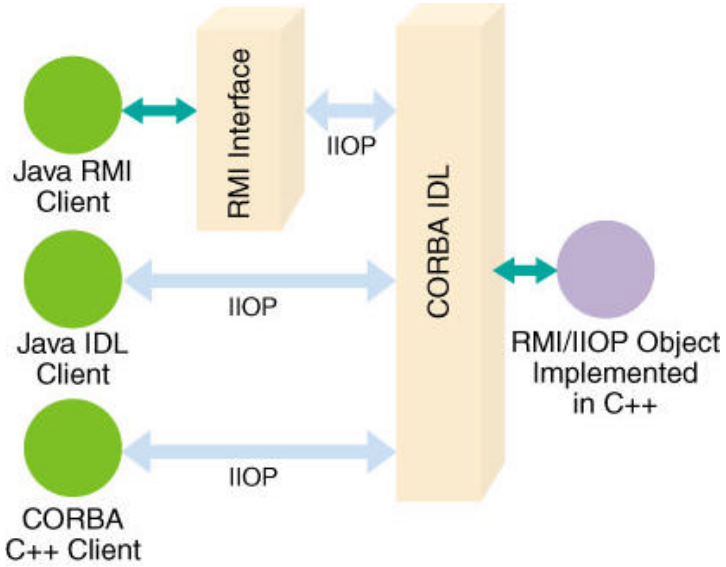


Abbildung 5 IIOP (Internet Inter ORB Protokoll) in Action

Die Einbindung von Enterprise Java Beans (EJB) in ein IIOP Umfeld wird schematisch in folgender Abbildung festgehalten.

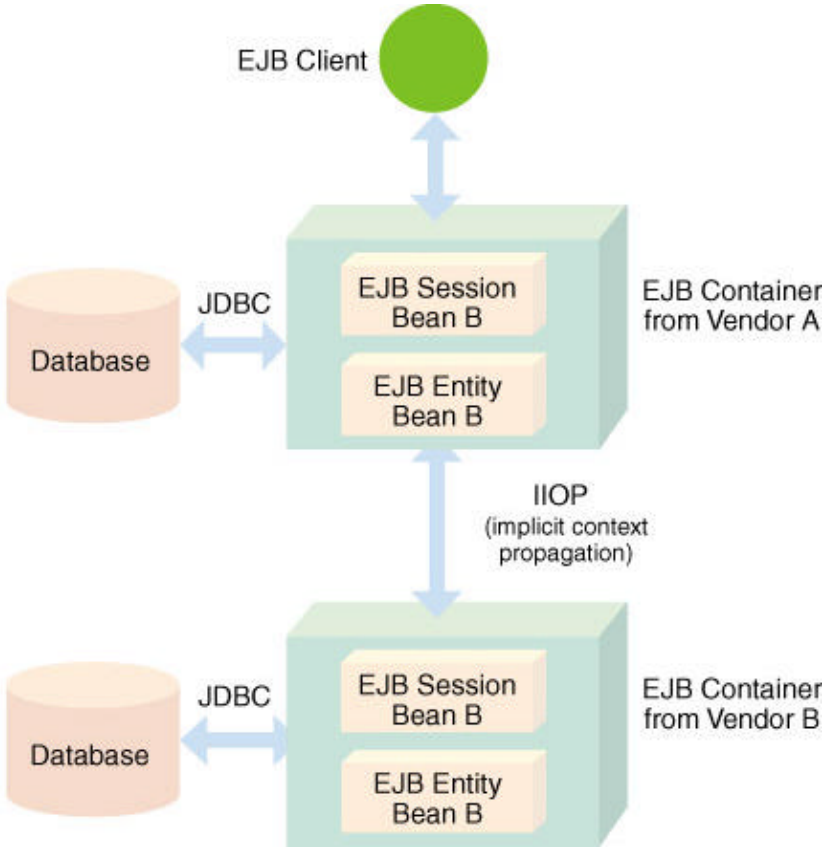


Abbildung 6 Enterprise Java Beans (EJB) kommunizieren über IIOP

Da das Kernsystem möglichst effizient und kompakt sein soll, wurden verschiedene, nur teilweise benötigte Dienste ausgelagert. In CORBA unterscheidet man Facilities und Services. Das Gesamtbild wird als OMA (Object Management Architecture) bezeichnet. Schematisch sieht die OMA wiefolgt aus:

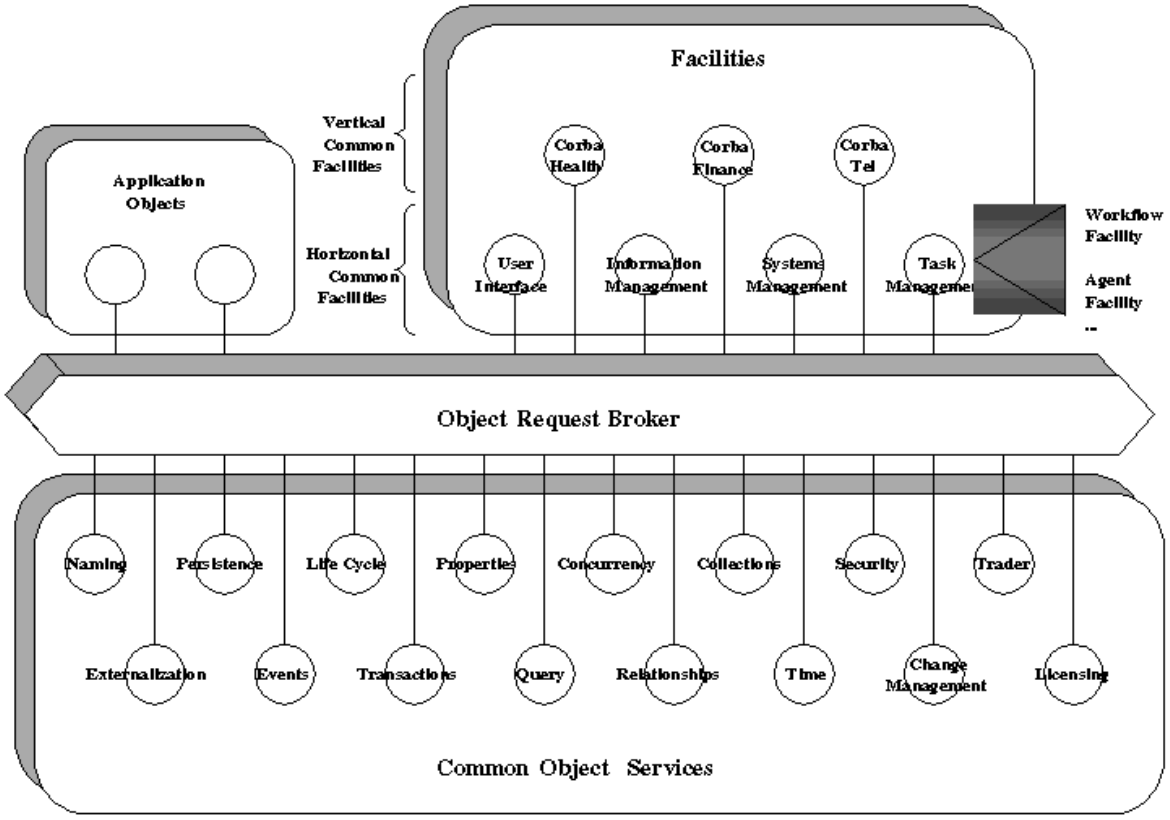


Abbildung 7 OMA Object Management Architecture der OMG

1.4.1.5. CORBA Services

Die CORBA Spezifikation besteht aus einem Kernsystem und einer Definition bestimmter Dienste, Services für verteilte Objektsysteme (*CORBA Services* oder COS).

Die CORBA-Services sind Sammlungen von Diensten auf der Systemebene, die selbst in IDL-definierten Schnittstellen zu finden sind. Sie sind als eine Erweiterung und Ergänzung der Funktionalität eines ORBs zu verstehen. Die OMG stellt zur Zeit 15 verschiedene CORBA-Services zur Verfügung.

Hier alle Services auf einen Blick:

- Life Cycle Service
  beschreibt, wie CORBA Objekte kreiert, entfernt, bewegt und kopiert wird.
- Persistence Service
- Naming Service
  beschreibt, wie Objekte in CORBA benannt werden können
- Event Service
  Entkoppelt die Kommunikation zwischen Objekten
- Concurrency Control Service
  beschreibt die Koordination der Zugriffe auf gemeinsame benutzte Objekte
- Transaction Service
  koordiniert den atomaren Zugriff auf CORBA Objekte
- Relationship Service
  beschreibt die n:m Relationen zwischen Objekten

- Externalization Service
  beschreibt die Transformation von CORBA Objekten zu und von externen Medien
- Query Service
  unterstützt die Abfrage von Objekten
- Licensing Service
- Properties Service
- Time Service
- Security Service
- Trader Service
- Collection Service

Weitere CORBA-Services sind zur Zeit in Arbeit. Der aktuelle Stand der Bemühungen kann jederzeit auf der Homepage von OMG abgefragt werden (http://www.omg.org).

Der Einsatz von CORBA-Services erlaubt CORBA-Objekte bestimmte Funktionalitäten zu verleihen, die sich mit der Systemebene befassen. CORBA-Objekte nutzen die CORBA-Services über Inheritance- und Containment-Relationships. So ist es beispielsweise möglich, eine Klasse "Auto" persistent und transaktionell zu machen, indem die entsprechenden CORBA-Services genutzt werden.

1.4.1.6. Die Facilities von CORBA

CORBA-Facilities sind das Modul für allgemein verwendbare, komplexe Diensteschnittstellen für den Enwickler von Endanwender-Applikationen. Sie definieren Regeln für das Zusammenspiel der Geschäftsobjekte, um effektiv zusammen arbeiten zu können. Zu den in der Entwicklung befindlichen Common Facilities gehören zum Beispiel Drucken, E-Mail, Datenaustausch, Frameworks für Business-Objekte und Internationalisierung.

1.4.1.7. CORBA Produkte

CORBA ist eine Spezifikation, eine Anleitung für die Implementation von Produkten, Brokern. Mehrere Anbieter brachten CORBA Produkte für unterschiedliche, und mit unterschiedlichen Programmiersprachen implementierte Produkte auf den Markt.

CORBA Produkte, welche Java direkt unterstützen, umfassen:

ORB	**Beschreibung**
Java 2 ORB	der Java 2 ORB wird mit dem Java 2 SDK ausgeliefert, allerdings ist dieser ORB unvollständig.
VisiBroker for Java	dieser ORB stammt von Inprise und ist Teil der JBuilder Enterprise Edition. VisiBroker ist auch in andere käufliche Systeme eingebettet, zum Beispiel dem Netscape Communicator (Browser)
OrbixWeb	ein populärer Java ORB von Iona Technologies, einem Pionier im CORBA Geschäft.
WebSphere	der Applikationsserver mit einem ORB, von IBM
Netscape Communicator	im Communicator ist ein VisiBroker eingebaut. Applets können daher CORBA Objekte integrieren, ohne dass ORB Klassen in den Browser geladen werden müssen.
mehrere frei erhältliche ORBS	es gibt mehrere frei erhältliche ORBs, zum Beispiel von Silvano Maffeis, von omiORB, Universitäten, JavaORB, ... Auch dazu finden Sie eine Liste auf dem OMG Site

1.4.1.7.1. JavaORB - ein frei erhältlicher Objekt Broker

JavaORB implementiert folgende CORBA Services :

- Interoperable NamingService (Teil vom JavaORB Core),
- EventService,
- TransactionsService,
- PropertyService,
- CollectionService.

1.5. Eine Beispielapplikation

Die Beschreibungen der CORBA Architektur, Services und Facilities umfasst in der Zwischenzeit mehrere tausend Seiten. Es liegt also ausserhalb der Möglichkeiten des Unterrichts, zu sehr in die Details zu gehen. Beispielsweise umfasst die Beschreibung der Security Services bereits ungefähr 500 Seiten. Darin wird zum Teil sehr technisch beschrieben, wie ein Sicherheitssystem für verteilte Objektsysteme im CORBA Umfeld aussehen könnte.

1.5.1. Die Börsen Applikation

Die Börsenapplikation ist eine verteilte Applikation, die den Einsatz von Java und CORBA in einem verteilten Objektsystem illustriert. Das Beispiel ist stark vereinfacht und stammt von zwei Java und CORBA Experten (Sun, Inprise). Wir werden im Übungsteil eine einfache Version davon implementieren

Sie können die Applikation beliebig erweitern!

Die Börsenapplikation erlaubt es mehreren Benutzern, Börsenaktivitäten zu beobachten. Dem Benutzer wird eine Liste mit Börsenfirmenkürzeln präsentiert. Er kann daraus mehrere auswählen und dann die "View" Taste drücken.

Im Screen Snapshot wurde SUNW gewählt.
Der Firmenname wird als Text eingeblendet und "View Aktie" aktiviert.

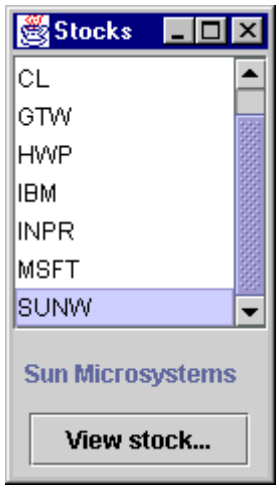


Abbildung 8 Auswahl der börsenkotierten Firma

Nachdem wir den "View" Button angewählt haben, erscheint ein Kurzbericht über den Börsenkurs, sowie das Börsensymbol, der Firmennamen, der aktuelle Preis, der Zeitpunkt, zu dem der Kurs zuletzt modifiziert wurde, sowie das Handelsvolumen und eine grafische Darstellung des Börsenverlaufs über eine bestimmte Zeitdauer.

Die Darstellung wird automatisch aktualisiert, wann immer neue Daten verfügbar werden.

Mit dem "Alarm..." Button kann ein Alarm eingeschaltet werden.

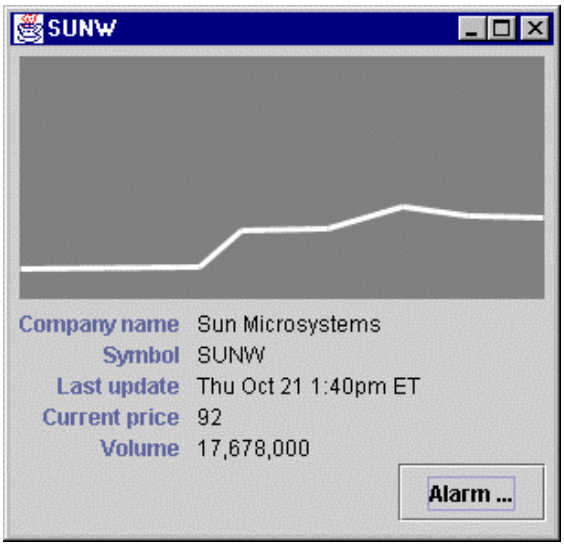


Abbildung 9 Börsenverlauf .

Dieser wird aktiviert, falls der Börsenkurs unter einen bestimmten Wert fällt oder eine bestimmte Schwelle überschreitet.

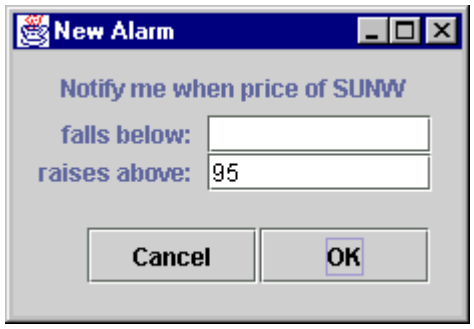


Abbildung 10 Schwellwerte

In diesem Falle wird der Benutzer unmittelbar benachrichtigt.

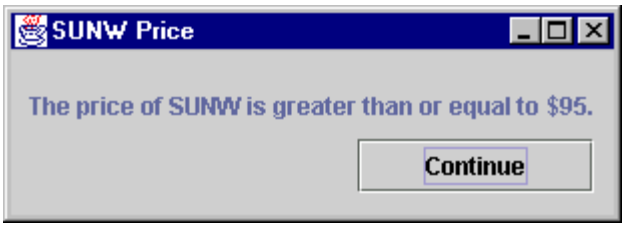


Abbildung 11 Information des Benutzers

Später werden wir die Anwendung ausbauen, so dass man auch Aktien kaufen und verkaufen kann.

1.5.1.1. Einige Objekte unserer Börsenapplikation

Auf Grund der obigen Beschreibung kann man folgende Objekte im verteilten System identifizieren.

Objekt	**Beschreibung**
Aktie [**Aktie**]	ein dezentrales Objekt, welches eine bestimmte Aktie repräsentiert
Aktiendarstellung [**AktiePresentation**]	ein dezentrales Objekt im GUI, welches die Börsendaten dem Benutzer anzeigt
Alarm [**Alarm**]	ein dezentrales Objekt, welches den vom Benutzer gesetzten Alarm festhält.
Alarmdarstellung [**Alarmpresentation**]	ein dezentrales Objekt im GUI, welches die Auslösung des Alarms dem Benutzer anzeigt.

1.6. Implementation des Client

Nachdem wir uns eine Übersicht über das System verschafft haben, wollen wir das System auch implementieren.

1.6.1. Implementation eines CORBA Clients

Als erstes wollen wir kennen lernen, wie Objekt Clients CORBA Objekte **benutzen** können. Wir beschreiben unser Client Objekt mit Hilfe der OMG IDL Interface Definition Sprache lernen, wie man Objektreferenzen erhält und als Client dezentrale Objekte, im Rahmen von CORBA erstellen und nutzen kann.

Nach dem Durcharbeiten dieses Beispieles und der Bearbeitung der Übungen, sollten Sie in der Lage sein, einfache Clients in Java zu schreiben.

Als Beispiel verwenden wir auch in den Übungen die Börsenapplikation, um die Clientseite von CORBA kennen zu lernen.

1.6.1.1. CORBA Objekte werden als IDL Interfaces beschrieben

Die OMG Interface Beschreibungsprache IDL unterstützt die Spezifikation von Objektschittstellen. Eine Objektschnittstelle beschreibt die Operationen, welche vom Objekt unterstützt werden. Das *Wie*, die Implementierung der Operationen, wird **nicht** beschrieben. In IDL kann man also Zustände und Algorithmen nicht beschreiben. Die Implementation eines CORBA Objekts wird mit Hilfe einer Standard Programmiersprache durchgeführt, also zum Beispiel Java oder C++ oder ....

Eine Schnittstelle, ein Interface, spezifiziert einen Kontrakt zwischen dem Programmcode, welcher das Objekt beschreibt, und dem Programmcode, welcher die Kunden des Objekts beschreiben.

Clients müssen lediglich die Schnittstellen kennen, nicht die Implementierung.

IDL Interfaces sind unabhängig von der verwendeteten Programmiersprache. IDL definiert *Sprachbindungen* (language bindings) für viele verschiedene Programmiersprachen. Der Programmierer hat somit die Möglichkeit, die jeweils optimale Programmiersprache für die Implementation selber bestimmen zu können. Das Gleiche gilt für die Clientseite. Zur Zeit kennt die OMG folgende Sprachbindungen:

- C
- C++
- Java
- Ada
- COBOL
- Smalltalk
- Objective-C
- Lisp

Sie finden die Beschreibung dieser Sprachabbildungen in der entsprechenden Spezifikation der OMG, jeweils für die einzelnen Sprachen. Einige der Abbildungen sind denkbar einfach, wie Sie der Tabelle weiter oben, für die Basistypen, entnehmen können. In andern Fällen ist die Beziehung komplexer.

Mit Hilfe von OMG IDL kann man folgende Konzepte rein in IDL, also ohne konkreten Bezug zu einer bestimmten Programmiersprache, beschreiben:

- Modularisierte Objekt Interfaces
- Operationen und Attribute, die ein Objekt unterstützt
- Ausnahmen, die durch Operationen ausgelöst werden können
- Datentypen einer Operation : Rückgabewerte einer Operation und die Objekt Attribute.

Die IDL Datentypen sind:

- Basisdatentypen (**lon**g, **shor**t, **strin**g, **floa**t…)
- konstruierte Datentypen (**struct**, **union**, **enum**, **sequence**)
- den Datentyp **any**, der dynamisch zugeteilt werden kann
- Objektreferenzen

Auch hier beschreibt IDL nicht, wie diese Typen zu implementieren sind.

Listing 1 Beschreibung einer börsenkotierten Firma [Aktie]

```
module BoersenObjekte {

      struct Kurs {
            string symbol;
            long zeitpunkt;
            double preis;
            long volumen;
      };

      exception Unbekannt{}; // kein Eintrag in der Tabelle

      interface Aktie {
            // liefert den aktuellen Börsenwert
            Kurs liesKurs() raises(Unbekannt);

            // setzt den aktuellen Börsenwert
            void setzeKurs(in Kurs boersenkurs);

            // liefert die Beschreibung zur Firmenabkürzung
            // zum Beispiel den Firmennamen
            readonly attribute string beschreibung;
      };

      interface AktienFactory {
            Aktie kreiereAktie(
                        in string symbol,
                        in string beschreibung
            );
      };
};
```

Die obige Definition des IDL Modules **BoersenObjekte**, umfasst die Beschreibung von:

- der Datenstrukture **Kurs**
- der Ausnahme **Unbekannt**
- des Interfaces **Aktie**
- des Interfaces **AktienFactory**

Der Modul definiert den Gültigkeitsbereich für diese Namen.
Innerhalb des Modules wird die Datenstruktur **`Kurs`** und eine Ausnahme **`Unknown`** definiert. Diese werden anschliessend im **`Aktie`** Interface eingesetzt. Das **`Aktie`** Interface wird seinerseits in der Definition des **`AktienFactory`** Interfaces eingesetzt.

Zu beachten ist, dass bei Parametern angegeben wird, ob es sich um Eingaben oder Ausgaben handelt: **`in`**, **`out`**, oder **`inout`**. Das **`in`** Schlüsselwort besagt, dass die Daten von einem Client an ein Objekt übergeben werden; das **`out`** Schlüsselwort besagt, dass die Daten vom Objekt an den Client zurück gegeben werden und **`inout`** besagt, dass die Daten in beide Richtungen weiter gegeben werden können.

Die IDL Beschreibungen werden mit Hilfe des IDL Compilers für die entsprechende Zielsprache kompiliert, in unserem Falle also mit Hilfe des Idlj Compilers von Sun. Aber die IDL Beschreibung ist vollständig (Programmiersprachen) sprachunabhängig.

1.6.1.2. Objekt Referenzen und Requests

Ein Client sendet einen Request an ein CORBA Objekt mit Hilfe einer *Objektreferenz*. Eine Objektreferenz identifiziert das Objekt, welches eine Anfrage erhalten wird. Schauen wir uns dies in Java an für die Klasse, die eine Objektreferenz für **`Aktie`** enthält und damit den aktuellen Börsenkurs erhält.

Listing 2 Programmfragment : Objektreferenz

```
Aktie dieAktie = …
try {
      Kurs aktuellerKurs = dieAktie.liesKurs();
} catch (Throwable e) {}
```

Objektreferenzen kann man in verteilten Objektsystemen als Parameter oder Rückgabewerte verwenden.

Zum Beispiel:
das **`AktienFactory`** Interface definiert eine **`kreiere()`** Methode, welche eine Instanz von **`Aktie`** liefert. In Java sieht dies folgendermassen aus:

Listing 3 AktienFactory

```
AktienFactory factory = …
Aktie dieAktie = …
try {
      dieAktie = factory.kreiere(
                        "GII",
                        "Global Industries Inc.");
} catch (Throwable e) {       }
```

Sie sehen, dass wie in RMI eigentlich kein Unterschied besteht, ob man eine lokale oder eine entfernte Methode aufruft. Das CORBA Objekt kann einfach, wie das RMI Objekt, irgendwo im Netz sein, sofern es registriert und damit zugänglich ist.

Das CORBA System garantiert Lokalisationstransparenz, der Benutzer muss also nicht wissen, ob das Objekt lokal oder entfernt ist.

Das CORBA Objekt muss also auch nicht auf der selben Maschine laufen, wie der Client Prozess, aber auch nicht dort wo der ORB läuft!

Objektreferenzen können persistent sein: man kann die Objektreferenz als Zeichenkette abspeichern und daraus das Objekt neu konstruieren.

Das folgende Beispiel zeigt die Grundstruktur einer solchen "stringified" Persistenz.

Listing 4 Stringified Persistent von Objekten

```
String AktieString =
      orb.object_to_string(dieAktie);
```

Diese Zeichenkette kann gespeichert oder kommuniziert werden. Ein Client kann auf Grund der Information das Objekt rekonstruieren, wie der folgende Programmcode zeigt.

Listing 5 Rekonstruktion eines Stringified Objekts

```
// Rekonstruktion
org.omg.CORBA.Object obj =
                              orb.string_to_object(AktieString);
// casten des Object
Aktie dieAktie = AktieHelper.narrow(obj);
```

1.6.1.3. IDL Typen System

IDL Interfaces kann man auch mit Hilfe weiterer IDL Interfaces definieren. Das folgende Beispiel soll dies illustrieren (zur Illustration, nicht zum Lernen):

Listing 6 IDL Modul

```
module ReportingObjects {
            exception EventChannelFailure{};
      interface Reporting {

      // Pushevents
      CosEventComm::PushSupplier push_events(
                  in CosEventComm::PushConsumer consumer)
            raises(EventChannelFailure);
      // Pullevents
      CosEventComm::PullSupplier pull_events(
                  in CosEventComm::PullConsumer consumer)
            raises(EventChannelFailure);
      };
};
```

Das **`Reporting`** Interface unterstützt die Registration für Ereignisse, für die man sich interessiert. Das CORBA Eventsystem interessiert uns im Moment nicht.

Wenn wir nun das Aktie und das Reporting Interface kombinieren, können wir ein ReportingAktie Interface definiert werden.

```
      interface ReportingAktie: Reporting, Aktie {
      };
```

Ein **`ReportingAktie`** unterstützt alle Methoden und Attribute, welche in den beiden Interfaces **`Aktie`** als auch **`Reporting`** definiert wurden.

Grafisch sieht dies wiefolgt aus:

Abbildung 12 Kombination des Reporting und Aktie Interfaces

Alle CORBA Interfaces erben implizit das **`Object`** Interface. und unterstützen somit alle Methoden, die für **`Objec`**t definiert wurden. In einem Java Programm könnte man ein **`ReportingAktie`**. Objekt direkt referenzieren:

```
ReportingAktie dieReportierteAktie;
```

Wir können auch überall dort, wo ein `Reporting` Interface verlangt wird, zum Beispiel einem Methodenaufruf, eine Objektreferenz auf ein `ReportingAktie` Objekt einsetzen, da die Oberklasse von `ReportingAktie` Reporting ist.

1.6.1.3.1. IDL Type Operationen

IDL Interfaces können in einer Hierarchie angeordnet werden. IDL stellt für bestimmte limitierte Operationen Methoden zur Verfügung:

- die **`narrow()`** Operation castet eine Objekt Referenz zu einem spezifischeren Typ:
  ```
  org.omg.CORBA.Object obj = …
  Aktie dieAktie = AktieHelper.narrow(obj);
  ```
- die **`is_a()`** Methode bestimmt, ob eine Objektreferenz ein bestimmtes Interface unterstützt.
  ```
  if (obj._is_a(AktieHelper.id()) …
  ```
- `die` **`id()`** Methode, definiert in der Helper Klasse liefert eine Repository ID für das Interface. Die Repository ID ist eine Zeichenkette, welche das Interface repräsentiert. In unserem Börsenbeispiel sieht die Situation folgendermassen aus:
  ```
  IDL:AktieObjects/Aktie:1.0
  ```
- schliesslich ist es auch möglich, eine Objektreferenz zu erweitern, also zu einem weniger speziellen Interface zu casten:
  ```
  Aktie dieAktie = dieReportierteAktie;
  ```

Ob es sinnvoll ist, Methoden einzusetzen, die vom IDL to Java Prozess stammen, sei dahin gestellt.

1.6.1.3.2. Request Type Checking

Der IDL Compiler für Java generiert Client Stubs, welche das CORBA Objekt lokal und in Java darstellen. Im generierten Java Code werden auch alle Interfaces und Datentypen dargestellt.

Der Client Code stützt sich also auf den generierten (Stub) Programmcode:

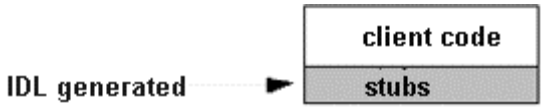


Abbildung 13 Abhängigkeit des Client Codes vom Stub

Falls wir zusammengesetze Typen, wie zum Beispiel `AktieReporting` haben, prüft der Java Compiler, ob die Parameterübergabe und die Verwendung der zusammengesetzten IDL Typen Java konform abläuft.

1.6.1.4. IDL zu Java Binding

Das Java Binding für IDL bildet verschiedene IDL Konstrukte auf entsprechende Java Konstrukte ab. Die folgende Tabelle zeigt, wie die IDL und Java Konzepte zusammenhängen (s. auch den dazu gehörenden Anhang).

IDL	Java	C++
module	package	namespace
interface	interface	abstract class
operation	method	member function
attribute	zwei Methoden	zwei Funktionen
exception	exception	exception

Die Nähe von Java zu C++ lässt sich auch in der obigen Tabelle erkennen.

Und hier die Beziehung zwischen den Datentypen in IDL und den Java Datentypen:

IDL Type	Java Type
boolean	boolean
char / wchar	char
octet	byte
short / unsigned short	short
long / unsigned long	int
long long / unsigned long long	long
float	float
double	double
string / wstring	String

Ein Begriff der immer wieder auftaucht in Zusammenhang mit den Typenzuordnungen ist *marshalin*g. Marshaling ist die Konversion von sprachspezifischen Datenstrukturen in das CORBA IIOP Datenstromformat (wie im RMI).

IIOP Daten können dann über das Netz an den Bestimmungsort übermittelt werden, wo sie wieder aus dem Datenstrom entnommen und (sprachabhängig) interpreteiert werden müssen.

1.6.1.5. Der IDL to Java Compiler

Zu CORBA IDL gehört ein Compiler, der IDL in die Java Programmiersprache übersetzt:

- Der IDL Compiler, der mit JDK ausgeliefert wird, heisst **`idlj`**.
- Der IDL Compiler, der mit VisiBroker für Java ausgeliefert wird, nennt sich **`idl2java`**.

Visibroker wird mit der JBuilder Enterprise Version (die auf dem Netz ist) ausgeliefert.

In unserem Börsenbeispiel generiert der Befehl

```
idlj Aktie.idl
```

(bis auf bestimmte Flags, die wir noch kennen lernen), generiert die Dateien,die unten aufgelistet sind.

Der VisiBroker ORB generiert die selben Dateien mit der Ausnahme, dass die Stubdateien andern genannt werden:

`_st_Aktie.java`, statt **`_AktieStub.java`**

- **`Aktie.java`**
  das IDL Interface als Java Interface
- **`AktieHelper.java`**
  implementiert die Typoperationen für das Interface (siehe oben)
- **`AktieHolder.java`**
  wird für die **`out`** und **`inout`** Parameter benötigt
- **`_AktieStub.java`**
  implementiert eine lokale Darstellung des remote CORBA Objektes. Diese Objekt ist für die Kommunikation mit dem remote Objekt zuständig. Der Client verwendet diese Klasse nicht direkt.

Der Entwickler übersetzt IDL mit dem IDL Compiler und übersetzt dann den generierten Java Code mit dem Java Compiler (`javac`). Dann muss nur noch der Classpath stimmen, um den Bytecode ausführen zu können.

1.6.1.6. Bestimmen einer Objekt Referenz

Es gibt drei Mechanismen für eine Methode zum Bestimmen einer Objektreferenz:

- die Objektreferenz kann als Parameter in einem Methodenaufruf übergeben werden
- die Objektreferenz kann das Ergebnis eines Methodenaufrufes sein, der Rückgabewert
- die Objektreferenz kann das Ergebnis einer Konversion einer Zeichenkette in eine Objektreferenz sein.

Alle drei Mechanismen werden durch ORBs unterstützt. Aufbauend auf diesen Mechanismen kann man höhere Dienste programmieren, mit deren Hilfe Objekte in verteilten Objektsystemen gefunden werden können.

1.6.1.7. Das Kreieren eines Objekts aus Clientsicht

Eine Möglichkeit, Objekte in einem verteilten Umfeld zu bauen, besteht darin, eine Objektfactory in der Objektwelt zu definieren. Die Factory ist dann für das Kreieren anderer Objekte in der Objektwelt zuständig.

Eine Factory ist ein **Entwurfsmuster** / Design Pattern, welche selbst ein Objekt ist, also auch ein IDL Interface besitzen kann, und sich damit irgendwo im Netzwerk befinden kann.

Eine Factory wird in einer Programmiersprache implementiert und CORBA Clients können Factorymethoden aufrufen.

.

Beim **`AktieObjects`** Beispiel ist das Factory Interface:

```
interface AktienFactory {
      Aktie kreiere_Aktie(    in string Aktie_symbol,
                              in string Aktie_description);
};
```

Um ein Börsenobjekt [`Aktie`] zu kreieren, muss der Client eine Anforderung an die Factory senden [`kreiere_Aktie`]. Wie die Factory konkret implementiert wird, ist wie üblich aus der IDL Beschreibung nicht ersichtlich.

1.6.1.8. Exceptions

In IDL existiert ein Exception Konzept, welches sich an Java Exceptions anlehnt. CORBA Exceptions werden auf Java Exceptions abgebildet. Wann immer eine CORBA Methode aufgerufen wird, muss das Java Sprachkonzept

`try` ... `catch`

eingesetzt werden.

CORBA kennt zwei Arten Exceptions, *System Exceptions* und *User Exception*s.

- System Exceptions werden dann geworfen, wenn irgend etwas mit dem System schiefläuft, zum Beispiel falls eine aufgerufene Methode auf dem Server nicht bekannt ist, oder falls ein Kommunikationsproblem existiert, oder falls der ORB nicht initialisiert wurde. Die Java Klasse **`SystemException`** erweitert **`RuntimeException`**. Damit meldet sich der Compiler auch nicht, falls sie den **`try... catch`** Block um die CORBA Aufrufe vergessen haben.
  System Exceptions in CORBA können sogenannte "minor codes" enthalten, welche zusätzliche Informationen über den Fehler liefern können.
  Leider sind diese herstellerspezifisch und müssen somit den Error Recovery Routinen des ORBs angepasst werden.
- User Exceptions werden dann geworfen, falls etwas beim Aufruf einer Methode selbst schief geht. Diese Exceptions werden innerhalb IDL definiert und der **`idlj`** Compiler generiert die entsprechenden Java Dateien.
  In unserer Börsenapplikation ist zum Beispiel **`Unknown`** eine Benutzerexception.
  UserExceptions sind eine Subklasse von **`java.lang.Exception`**, der Compiler beschwert sich also, falls diese Ausnahmen nicht abgefangen werden.

1.7. Implementation eines einfachen Verteilten Objekt Systems

Bisher haben wir die Implementation von Objekten in verteilten Systemen aus der Sicht der Clients besprochen. Das Ergebnis war, dass sich eine Implementation clientseitig kaum von einer lokalen Implementation unterscheidet. Jetzt möchten wir die Sichtweise ändern und die Objekte in einem verteilten System aus Sicht des Servers betrachten

1.7.1. Objekt Implementation

Als erstes muss man sich bewusst werden, dass die Implementation der Objekte abhängig ist vom konkreten ORB, obschon die Idee des ORBs genau das Gegenteil bezweckte.

Hier betrachten wir als erstes ein generisches, also allgemeines Vorgehen. Spezialitäten der einzelnen kommerziellen Systeme werden zum Teil später besprochen (VisiBroker und Java 2 ORB).

Es geht in diesem Abschnitt also um die serverseitige Sprachbindung für IDL, darum, wie wir Server Objekte implementieren können, CORBA Objekt Adapter. Sie sollten nach dem Durcharbeiten diese Abschnitts wissen, wie man CORBA Objekte implementiert.

Zur Illustration verwenden wir wieder die Börsenapplikation.

Die CORBA Implementation des Objekts ist dem Client völlig unbekannt. Er muss sich auch nicht dafür interessieren. Selbst die Programmiersprache darf keine Rolle spielen! Der Client darf nur von der IDL Schnittstelle abhängen.

Unter anderem sind folgende Implementierungen des **`Aktie`** Interfaces denkbar:

- Die Implementationsklasse wird selbst in Java geschrieben
- die Implementationsklassen sind in C++ geschrieben, inklusive Zugriff auf Datenbanken und das WWW.
- ein Smalltalk Fan implementiert die Serverseite in Smalltalk und baut zusätzliche Funktionalitäten ein, die nur Smalltalk bietet.

1.7.1.1. Erstellen einer Implementation

Ausgehend von der IDL Beschreibung generiert der IDL Compiler verschiedene Dateien für den CORBA Client, aber auch das *Skeleton*, also der Adapter auf der Server Seite.

Ein Skeleton ist der Entry Point für das verteilte Objektsystem. Ein Skeleton ist dafür verantwortlich, dass die Objekte aus dem IIOP CORBA Datenstrom gelesen und rekonstruiert werden (unmarshaling) und liefert anschliessen das Ergebnis an den Stub zurück (marshaled).

Der Entwickler braucht sich nicht um den inneren Aufbau des Skeletons zu kümmern. Seine Aufgabe ist die korrekte Implementation des IDL Interfaces.

Um CORBA Objekte zu implementieren, muss der Entwickler eine Java Klasse implementieren, welche die generierte Skeleton Klasse erweitert und alle in IDL definierten Methoden implementieren.

In unserem Börsenbeispiel generiert der IDL Compiler die Skeleton Klasse

_AktieImplBase für das Börsen [**Aktie]** Interface.

Eine mögliche Implementation des **Aktie** Interfaces ist:

```
public class AktieImpl extends AktieObjects._AktieImplBase {
      private Kurs _Kurs=null;
      private String _description=null;
      public AktieImpl(String name, String description) {
            super();
            _description = description;}
      public Kurs liesKurs() throws Unknown {
            if (_Kurs==null) throw new Unknown();
            return _Kurs;}
      public void set_Kurs(Kurs Kurs) {
            _Kurs = Kurs;}
      public String description() {
            return _description;}
}
```

1.7.1.2. Interface versus Implementation Hierarchien

Man muss beachten, dass es zwei Hierarchien gibt:
1. eine Interface Hierarchie und
2. eine Implementationshierarchie

Die Interface Hierarchie in unserem **ReportingAktie** Beispiel ist:

Abbildung 14 Interface Hierarchie

In IDL ausgedrückt:

```
interface ReportingAktie: Reporting, Aktie {};
```

Betrachten wir nun eine Implementation von **ReportingAktie**, genannt **ReportingAktieImpl**, welche das IDL generierte Skeleton **_ReportingAktieImplBase** erbt und einige Methoden an **AktieImp**l delegiert und schliesslich die **Reporting** Methoden implementiert:

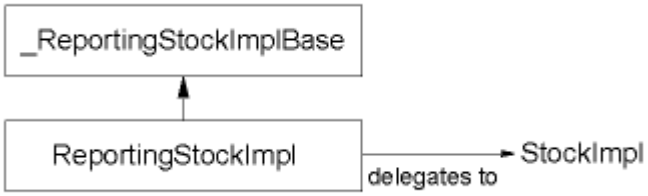


Abbildung 15 Implementationshierarchie

In Java sieht diese Klassenhierarchie folgendermassen aus:

```
class ReportingAktieImpl
      implements  ReportingAktie
      extends           _ReportingAktieImplBase {
…
}
```

Da Java keine Mehrfachvererbung implementiert, wird oft das **Delegation Entwurfsmuster** eingesetzt:

- die Implementation kreiert oft eine Instanz einer anderm Klasse und delegiert Aktivitäten an diese Klasse.

Im obigen Beispiel delegiert die **`ReportingAktieImpl`** an die **`AktieImpl`** Klasse für die Implementation einzelner Methoden.

Es ind auch andere Klassenhierarchien zur Implementierung der Interface Hierarchie denkbar, bei gleicher Interface Spezifikation. Änderungen in der Implementierung wirken sich, falls die IDL Beschreibung konstant bleibt, nicht auf den Client aus.

1.7.1.3. Implementation vom Type Checking

Type Checking wird auf der Client und Server Seite durchgeführt. Der IDL Compiler für die Java Programmiersprache generiert Skeletons und Java Code, der alle in IDL definierten Datentypen implementieren kann. Die Implementation hängt damit vom generierten Code des IDL Compilers ab.

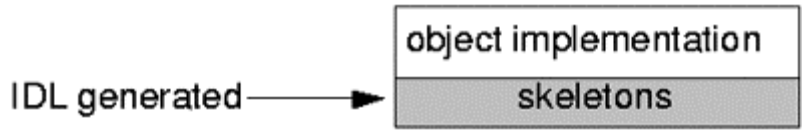


Abbildung 16 Die CORBA Objekte basieren auf Skeletons

Falls Typefehler im generierten Java auftreten, meldet dies der Java Compiler, nicht der IDL Compiler.

Betrachten wir ein einfaches Beispiel:

```
Kurs AktieImpl.liesKurs() {
      double price = …;
      return price;
}
```

Gemäss dieser Implementation gibt die Methode `liesKurs`() den Preis als double zurück. Falls in IDL ein anderer Datentyp angegeben wurde, kann der IDL Compiler diesen Unterschied nicht mehr anzeigen. Der Java Compiler wird den Fehler allerdings merken, da der Preis in den Anwendungen definiert und verwendet wird.

1.7.1.4. Implementation eines Server mit dem Java 2 ORB

Bisher haben wir allgemeine Themen besprochen. Jetzt möchten wir in Richtung konkreter Implementierung gehen. Hier geht es speziell darum, einen Server zu definieren und zu entwicklen, welcher Objekte den Clients zur Verfügung stellt. Der Server wird in der Implementation dieses Kapitels mit dem Java 2 ORB arbeiten.

Wie geht man konkret vor

- definieren einer main Methode
- initialisieren des ORBs
- Instanz mindestens einer Klasse (ein Objekt) bilden
- Objekt mit dem ORB verbinden
- auf Clients warten

Der Server muss mindestens ein Objekt instanzieren, da in CORBA nur mit Hilfe von Objekten Dienste angeboten werden können.

Schauen wir uns eine mögliche Implementation des Objektservers an, speziell für den Java 2 ORB:

```
public class theServer {
      public static void main(String[] args) {
            try {
                  // ORB iinitialisieren
                  org.omg.CORBA.ORB orb =
                              org.omg.CORBA.ORB.init(args,null);
                  // Aktie Objekt kreieren.
                  AktieImpl dieAktie =
                  new AktieImpl("GII","Global Industries Inc.");
                  // ORB informieren
                  orb.connect(dieAktie);
                  // Schreibe stringified Object Referenz in Datei
                  PrintWriter out =
                        new PrintWriter(new BufferedWriter(
                              new FileWriter(args[0])));
                  out.println( orb.object_to_string(dieAktie) );
                  out.close();
                  // warten auf Kunden
                  java.lang.Object sync = new java.lang.Object();
                  synchronized (sync) {
                        sync.wait();
                  }
            } catch (Exception e) {
                  System.err.println("Aktie server error: " + e);
                  e.printStackTrace(System.out);
            }
      }
}
```

Wichtig ist, dass diese Klasse mit **`new`** die **`AktieImpl`** Klasse zum **`Aktie`** Interface implementiert und dann den ORB mit **`connect()`** darüber informiert, dass das Objekt auf Anfragen wartet.

1.7.1.5. Implementieren eines Server mit VisiBroker

Der Ablauf beim Einsatz von VisiBroker statt Java 2 unterscheidet sich in einigen wenigen Punkten von der Java 2 Implementation.

Konkret sieht der Ablauf folgendermassen aus:

- definieren einer main Methode
- der ORB und der BOA (basic object adapter) wird instanziert
- mindestens ein Objekt instanzieren
- dem BOA mitteilen, dass das Objekt auf Kunden wartet.
- dem BOA mitteilen, dass der Server bereit ist

Der Server muss auch hier mindestens ein Objekt instanzieren, da in CORBA Dienste nur als Methoden von Objekten angeboten werden können.

Und so könnte der VisiBroker basierte Server für CORBA Objekte aussehen:

```
public class derServer {
      public static void main(String[] args) {
            try {
                  // initialisieren des ORB
                  org.omg.CORBA.ORB orb =
                              org.omg.CORBA.ORB.init(args,null);
                  // initialisieren des BOA.
                  org.omg.CORBA.BOA boa =
            ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();
                  // kreieren eines Börsenobjekts.
                  AktieImpl dieAktie =
                  new AktieImpl("JMJG","Joller Global Industries Inc.");
                  // stringified Objekt Referenz in Datei
                  PrintWriter out =
                        new PrintWriter(new BufferedWriter(
                              new FileWriter(args[0])));
                  out.println( orb.object_to_string(dieAktie) );
                  out.close();
                  // an BOA : das Objekt ist bereit
                  boa.obj_is_ready(dieAktie);
                  // an BOA :  der Server ist bereit
                  boa.impl_is_ready();

            } catch (Exception e) {
                  System.err.println("Aktie server error: " + e);
                  e.printStackTrace(System.out);
            }
      }
}
```

Auch hier instanziert der Server mit **`new`** ein **`AktieImpl`** Objekt, welches das **`Aktie`** Interface implementiert und liefert diese Instanz an den BOA (basic object adapter), damit dieser weiss, dass Anfragen von Clients entgegengenommen werden können.

1.7.1.6. Unterschiede der Server Implementationen

Als Zusammenfassung dient die folgende Tabelle, aus der die Unterschiede ersichtlich sind.

	Java 2 ORB	**VisiBroker 3.x for Java**
Initialisierung	ORB initialisieren	ORB und BOA implementieren
Objekt Export	`orb.connect(dieAktie)`	`boa.obj_is_ready(dieAktie)`
Server ist bereit	Suspend `main` Thread mit `wait()`	`boa.impl_is_ready()`

Dies sind die einzigen Unterschiede für transiente (=temporäre) Objektserver. Es gibt weitere Unterschiede, sobald Sie persistente Objekte und die automatische Aktivierung von Objekten betrachten..

1.7.1.7. Packaging Objekt Implementationen

Wie oben beschrieben, ist es sinnvoll, die Objektimplementation von der eigentlichen Serverapplikation zu trennen. Dies gestattet Ihnen eine bessere Verteilung der Objektimplementationen auf die einzelnen Server.

Die Objektimplementation hängt nicht vom Server ab; der Server hängt natürlich von den Objekten, die er enthält, ab.

Ein weiterer Vorteil der Trennung besteht in der Portierbarkeit, da der Server oft vom konkreten ORB abhängt, die Objektimplementation aber nicht.

Eine gute Strategie besteht darin, die Objektimplementation plus Stubs und Skeletons in eine Java Bean zu packen. Damit lässt sich die Implementation mit Hilfe der Java Beans Design Tools manipulieren.

1.8. Objekt Adapter

Die CORBA Spezifikation definiert das Konzept des Objekt *Adapters*. Aber das Konzept ist nicht unbestritten. Wir betrachten es im Folgende trotzdem.

1.8.1. Objekt Adapter

Ein Objekt Adapter ist ein Framework für die Implementierung von CORBA Objekten. Ein Adapter stellt somit eine Art API für die Objektimplementation für mehrere Services zur Verfügung.

Gemäss der OMG CORBA Spezifikation ist ein Objekt Adapter verantwortlich für:

- Generierung und Interpretation von Objektreferenzen
- Methodenaufrufe
- Sicherheit der Interaktionen
- Objekt und Implementation (De-)Aktivierung (Activation und Deactivation)
- Mapping der Objektreferenzen auf die dazugehörigen Objektimplementationen
- Registration der Implementationen

Die CORBA Architektur gestattet unterschiedliche Definitionen für viele unterschiedliche Objektadapter. Die Spezifikation umfasst insbesondere die Definition des *basic object adapter* (BOA).

Im vorigen Abschnitt sahen wir, das der VisiBroker BOA implementiert. Es gibt verschiedene CORBA Produkte, die den BOA implementieren. Die unterschiedlichen BOAs sindjedoch in der Regel nicht kompatibel, da die Spezifikation unvollständig war. Dies führt zur sofortigen Reduktion der Portierbarkeit der Applikationen von einem ORB zum andern.

Damit diese Probleme behoben werden können, wurde ein völlig neuer Adapter definiert, der *portable object adapter* (POA). Dieser wird jedoch von vielen ORB Produkten noch nicht unterstützt. Aber wir beschreiben den Adapter trotzdem..

1.8.1.1. Activation on Demand durch den Basic Object Adapter (BOA)

Eine der Grundfunktionen des BOA ist die Aktivierung von Objekten auf Abruf. Falls ein Client eine Anfrage plaziert, wird zuerst geprüft, ob das Objekt aktiv ist. Wenn nicht, wird das Objekt geladen, aktiviert und die Methode ausgeführt.

BOA kennt vier unterschiedliche Objektaktivierungsmechanismen:

- **Shared Server**
  mehrere aktive Objekte teilen sich einen Server. Der Server bedient mehrere Clients. Der Server bleibt dauernd aktiv.
- **Unshared Server**
  Auf dem Server ist nur ein Objekt aktiv. Der Server stopped mit dem Client.
- **Server-per-Method**
  jede Anfrage erzeugt einen Server. Nach der Ausführung stoppt der Server wieder.
- **Persistent Server**
  der Server wird unabhängig vom BOA gestartet. Mehrere Objekte sind aktiv.

1.8.1.2. Portable Object Adapter (POA)

Gemäss Spezifikation:
"The intent of the POA, as its name suggests, is to provide an object adapter that can be used with multiple ORB implementations with a minimum of rewriting needed to deal with different vendors' implementations."

Die meisten CORBA Produkte unterstützen den POA noch nicht. POA soll auch die Persistenz von Objekten unterstützen - zumindest Persistenz aus Clientsicht. Aus Clientsicht sollten diese Objekte also immer aktiv sein.

Der POA unterstützt viele zusätzliche Features:

- transparente Objektaktivierung
- mehrere aktive Objekte gleichzeitig
- transiente Objekte (temporäre)
- Namensräume für die Objekte (ObjID)
- Multithreading, Security und Objekt Management
- mehrere unterschiedliche POA's in einem Server mit unterschiedlichen Policies und Namensräumen

Der Programmierer muss sich also nicht um das Thread Management kümmern. Allerdings ist er dafür verantwortlich, dass das System thread-safe ist.

Die folgenden Abschnitte sind Appendices :
in ihnen werden verschiedene Aspekte, die in den vorhergehenden Kapitel diskutiert wurden, an Hand von Beispielen vertieft.

1.9. Resourcen

Die folgende Zusammenstellung ist unvollständig, wie jede Zusammenstellung. Aber sie liefert eine Übersicht über die im Web vorhandenen Informationen über CORBA im Web.

1.9.1. Resourcen

Die Zusammenstellung beschränkt sich auf die ORBs, die in den Beispielen verwendet werden. Weitere Links zu ORB's finden Sie unter anderem auf dem OMG Site.

1.9.1.1. Web Sites

OMG	http://www.omg.org

1.9.1.2. Dokumentation und Spezifikation

CORBA IIOP Spezifikation	http://www.omg.org/library/c2index.html
Objects by Value	ftp://ftp.omg.org/pub/docs/orbos/98-01-18.pdf
The VisiBroker For Java Reference *The VisiBroker For Java Programmers Guide*	http://www.inprise.com
JDK 1.2 CORBA	http://java.sun.com/products/jdk/1.2/docs/guide/idl/
Java 2 Dokumentation: **`org.omg.CORBA`**	http://java.sun.com/products/jdk/
Java Transactions	http://java.sun.com/products/jta/ http://java.sun.com/products/jts/ ftp://www.omg.org/pub/docs/formal/97-12-17.pdf
Java Naming Services	http://java.sun.com

1.9.1.3. Bücher

- Orfali & Harkey, *Client/Server Programming with Java and CORBA* (Wiley)
- Orfali, Harkey & Edwards, *Instant CORBA* (Wiley)
- Vogel, Andreas and Keith Duddy, *Java Programming with CORBA*(Wiley)
- Perdrick et al, *Programming with VisiBroker* (Wiley).

1.9.1.4. Verschiedene Quelle

`idlj` Compiler	http://java.sun.com
RMI over IIOP	http://java.sun.com/products/rmi-iiop
JacORB	gratis ORB mit POA, DII, DSI, CosNaming
OrbixWeb	http://www.iona.com
WebSphere Application Server	http://www.ibm.com
Übersichtsliste über gratis ORBs:	http://www.omg.org

1.10. Notizen zu Java 2 ORB

Der Java 2 ORB wird mit dem JDK ausgeliefert, ist also fixer Bestandteil des JDK und auch gratis.

1.10.1. Java 2 ORB

Der Java IDL ORB, der mit der Java 2 Platform ausgeliefert wird, erlaubt es, entweder Java Applikationen oder Applets über IIOP mit ORBs zu kommunizieren.

Der Java ORB ist eher generisch, allgemein gehalten. Das ist gut und schlecht:

- positiv, weil es wenig Überraschungen gibt
- negativ, weil viele Funktionen fehlen:
  - es gibt keinen Interface Repository (Java IDL Clients können ein Interface Repository eines andern ORBs nutzen)
  - Transaction Service
  - POA

Sie finden eine vollständigere Liste unter der JDK Dokumentation **`CORBA Package JavaDoc Comments.`**

Java IDL ist so aufgebaut, dass man analog zu "plug and play" unterschiedliche ORBs einsetzen kann, auch andere Java Virtual Machines.

Die kann man durch Setzen von Umebungsvariablen und Properties erreichen. Hinweise dazu finden Sie in der JavaDoc Dokumentation.

1.10.1.1. idlj und idltojava

Der **`idltojava`** Compiler kann gratis vom Java IDL Website herunter geladen werden. Standardmässig versucht der **`idltojava`** Compiler einen C Preprocessor zu starten.

Falls kein C Präprozessor installiert ist, wird **`idltojava`** eine tolle Fehlermeldung kreieren:

```
Bad command or file name
Couldn’t open temporary file
idlj: fatal error: cannot preprocess input;
No such file or directory
```

Besser ist es gleich den auf IIOP ausgerichteten idlj Compiler zu verwenden, welcher mit JDK mitgeliefert wird (im bin Verzeichnis).

In diesem Falle muss der **`idltojava`** Compiler mit er Option,dem Flag **`-fall`** gestartet werden:

```
idlj -fall foo.idl
```

1.10.1.2. System Properties

Die **`ORB.init()`** Methode kann Konfigurationsparameter aus unterschiedlichen Quellen lesen:

1. aus den Applikationsparametern, dem ersten Argument von **`ORB.init(`**),
2. aus einem applikationsspezifischen **`Properties`** Objekt, dem zweiten Argument von **`ORB.init(`**),
3. oder von den System Properties, die beim Start mit dem **`-D`** Flag angegeben werden

Gemäss IDL Guide werden zur Zeit folgende Properties für alle ORB Implementationen unterstützt:

- **`org.omg.CORBA.ORBClass`**
  Der Namen der Java Klasse, die das **`org.omg.CORBA.ORB`** Interface implementiert. Applets und Applikationen brauchen diesen Parameter nur dann zu spezifizieren, falls sie eine spezielle ORB Implementation ansprechen wollen.Für den Java IDL ORB ist dieser Wert gleich **`com.sun.CORBA.iiop.ORB`**.
- **`org.omg.CORBA.ORBSingletonClass`**
  Der Name einer Java Klasse, die das **`org.omg.CORBA.ORB`** Interface implementiert. Dieses Objekt wird durch Aufruf von **`orb.init()`** ohne Argumente, zurück geliefert. Der Wert für Java IDL ORB ist **`com.sun.CORBA.iiop.ORB`**.

Zusätzlich werden folgende Standardeigenschaften, Properties unterstützt, gemäss Java IDL:

- **`org.omg.CORBA.ORBInitialHost`**
  Der Host Name der Machine, auf der der Server oder Daemon läuft welcher die bootstrap Services zur Verfügung stellt. Der Defaultwert dieser Eigenschaft ist **`local`**`l`**`host`** für Applikationen. Für Applets ist es der Applet Host, also **`getCodeBase().getHost(`**).
- **`org.omg.CORBA.ORBInitialPort`**
  Der Port auf dem der Nameservice aktiv ist. Dieser ist per Default **`900`**.

1.11. VisiBroker

Der VisiBroker ORB ist Bestandteil der JBuilder Enterprise Edition. Im Folgende geht es darum einige wesentliche Punkte zu VisiBroker zusammen zu fassen.

1.11.1. VisiBroker Übersicht

Es geht also um die Beschreibung einzelner Werkzeuge, nicht um eine detaillierte Beschreibung von VisiBroker. Die Dokumentation finden Sie auf der CD oder im Web: http://www.inprise.com/visibroker

1.11.1.1. VisiBroker Tools

VisiBroker für Java enthält eine ganze Menge Tools. Einige dienen als Wrapper oder Ersatz für JDK Werkzeuge. Andere sind spezifische VisiBroker Werkzeuge:

- **`vbj`** ist ein Wrapper für **`java`**, welcher auch zusätzlich Properties setzt und den Classpath ergänzt.
- **`vbjc`** ist ein Wrapper für **`javac`**, welcher einige Properties setzt und den Classpath ergänzt.
- **`idl2java`** ist ein IDL Compiler, der proprietary oder portable Stubs und Skeletons generiert.
- **`osagent`** startet einen Smart Agenten

1.11.1.2. Einsatz von VisiBroker mit Java 2

Damit der VisiBroker mit Java 2 zusammenarbeiten kann, müssen einige Änderungen gemacht werden. Dies liegt daran, dass die JDK CORBA Klassen im **`org.omg.CORBA.*`** Package sich von den CORBA Klassen des VisiBroker unterscheiden.

Die VisiBroker Klassen enthalten verschiedene nicht Standarderweiterungen von CORBA; einige dieser Erweiterungen sind notwendige Voraussetzungen für ein funktionierendes System. Zum Teil müssen die Programme dem Broker angepasst werden:

```
org.omg.CORBA.ORB orb =
                  org.omg.CORBA.ORB.init(args, null);
org.omg.CORBA.BOA boa =
      ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();
```

Sie finden den vollständigen Server weiter vorne im Skript.

Damit das Programm läuft, muss der Aufruf von **`ORB.init()`** den VisiBroker ORB und nicht der Standard JavaIDL ORB von Sun zurück liefern. Damit dies passiert, müssen die Systemproperties des Javasystems gesetzt werden. Dies geschieht automatisch, falls man **`vbj`** anstatt **`java`** verwendet, um das Programm zu starten.

Falls Sie doch **`java`** einsetzen wollen, müssen Sie die Properties entsprechend setzen:

- **`org.omg.CORBA.ORBClass -com.visigenic.vbroker.orb.ORB`**
- **`org.omg.CORBA.ORBSingletonClass -com.visigenic.vbroker.orb.ORB`**

Auch die VisBroker Dokumentation enthält eine Beschreibung dieser Properties.

1.11.1.3. Portable Stubs und Skeletons

Standardmässig generiert *VisiBroker* für Java Stub und Skeleton Code, der *nicht* interoperabel und portabel ist. Aus Sicht von VisiBroker macht dies natürlich Sinn, auch in Bezug auf die Performance. Man hat jedoch die Möglichkeit diese Portabilität zu erzwingen:

```
idl2java -portable -no_bind Foo.idl
```

damit Stubs und Skeletons portabel werden. Dies ist oft wichtig, weil zum Beispiel Netscape eine Version des Java ORB einsetzt. Damit Applets auch auf den VisiBroker zugreifen können, muss der Code entsprechend portabel sein.

1.11.1.3.1. Was ist der Unterschied zwischen portabler und proprietärer Version?

Ein portabler Stub benutzt das DII (Dynamic Invocation Interface) um das Objekt zu marshallen; ein portables Skeleton benutzt DSI (Dynamic Skeleton Interface).

Die Visibroker Version verwendet direktere Aufrufe und vermeidet so den Overhead von DII und DSI.

Diese Änderungen werden vom IDL Compiler **`idl2java`** automatisch generiert, bei entsprechenden Flags.

Die einzige Differenz besteht darin, dass im Falle des portablen Codes:

`_FooImplBase` extends **`org.omg.CORBA.DynamicImplementation`**

statt

`_FooImplBase` extends **`com.inprise.vbroker.CORBA.portable.Skeleton`**

und dass die Stub Klasse **`_portable_stub_Foo.java`** statt **`_st_Foo.java`** heisst.

Man kann den Stub sogar dynamisch ändern:
`FooHelper.setProxyClass(_portable_stub_Foo.class)`

obschon dies etwas wild und nicht sehr zuverlässig ist

1.11.1.4. Einsatz des BOA mit VisiBroker

Das VisiBroker BOA benutzt eine nicht standardkonforme Version der BOA Initialisierung.

Ein typisches Beispiel sieht folgendermassen aus:

```
// kreieren und initialisieren des ORB
ORB orb = ORB.init(args, null);

// Initialisieren des BOA.
// Casting für VBJ ORB um Java 2 Kompatibilität herzustellen
org.omg.CORBA.BOA boa =
      ((com.inprise.vbroker.CORBA.ORB)orb).BOA_init();
```

VisiBroker Objekte sind in der Regel **persistent**, dauerhaft. In VisiBroker Terminologie heisst das, dass die Objekte benannt werden:

```
// kreieren des Objekts und registerieren beim ORB
Aktie dieAktie = new AktieImpl(name);
```

Der VisiBroker BOA überspringt die **`boa.kreiere()`** Phase und springt direkt zu :

`obj_is_ready(`).

```
// Export des neu kreierten Objekts.
boa.obj_is_ready(dieAktie);
```

`//` die **`impl_is_ready()`** Methode wartet einfach

```
// Warten auf einteffende Anfragen
boa.impl_is_ready();
```

1.11.1.5. Einsatz des VisiBroker Smart Agent

VisiBroker for Java wird mit einem eigenen Location Service geliefert, dem Smart Agent. Der Smart Agent ist ein verteilter Lokalisierungsdienst. Der Agent arbeitet mit andern Smart Agents, die im Netzwerk aktiv sind, zusammen, um die gesuchten Objekte zu finden.

Falls mehrere Implementierungen existieren, wählt der Smart Agent eine Implementation aus. Damit erreicht man eine Fehlertoleranz und ein Loaf Balancing. Falls ein Server abstürzt, sucht der Smart Agent automatisch eine andere Implementation. Der Client merkt nichts davon.

Falls man ein persistentes Objekt kreiert, indem man das Objekt benennt (Zeichenkette als Parameter des Konstruktors), dann informiert der BOA automatisch deb Smarten Agent.

Auf der Client Seite steht eine nicht CORBA konforme Methode **`bind()`** zur Verfügung:

```
// "bind" (in etwa eintragen) der Objektreferenz
Aktie dieAktie = AktieHelper.bind(orb, "GII");
```

1.12. Hello World als CORBA Applikation

Da die Börsenapplikation bereits recht komplex ist, finden Sie hier im Anhang ein einfacheres Beispiel, aus der Java Beschreibung : HelloWorld als CORBA Anwendung.

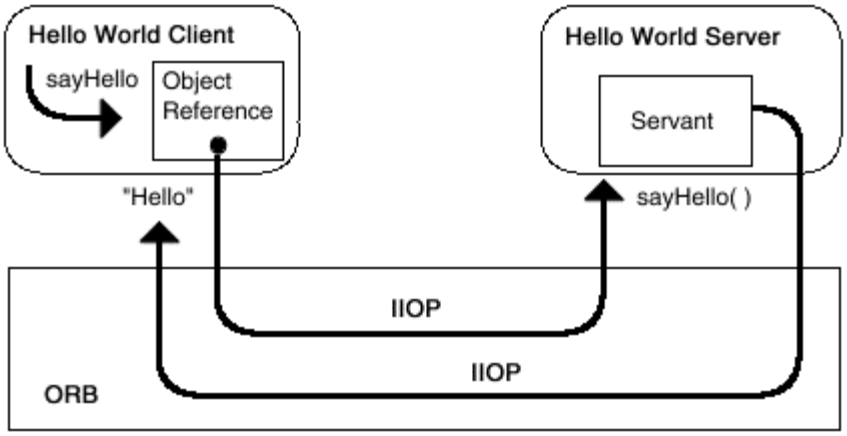


Abbildung 17 Hello World CORBA Applikation

1.12.1. IDL Beschreibung der HelloWorld Objektwelt

```
module HelloApp
{
  interface Hello
  {
    string sayHello();
  };
};
```

Compilieren dieser IDL Beschreibung mit Hilfe des idlj Compilers (der IDL-to-Java Compiler ist auf dem Website http://java.sun.com erhältlich bzw. über die Enterprise Edition von JBuilder).

```
idlj -fall Hello.idl
```

(das Flag -fall besagt, dass kein C++ Preprocessor aufgerufen werden soll; dieses Flag müsste per Default eigentlich auf no sein, ist es aber nicht) liefert folgende Dateien:

- `_HelloImplBase.java`
  Diese abstrakte Klasse (Listing weiter unten) ist das Server **Skeleton**, welches die grundlegenden CORBA Funktionalitäten zur Verfügung stellt. Sie implementiert das `Hello.java`a Interface. Die Serverklasse HelloServer wird diese `_HelloImplBase` Klasse erweitern.
- `_HelloStub.java`
  Diese Klasse ist der Client **Stub**, welcher die CORBA Funktionalitäten für den Client zur Verfügung stellt. Die Klasse implementiert die `Hello.java` Schnittstelle.
- `Hello.java`
  Diese Schnittstelle enthält die **Java Version der IDL** Beschreibung der Schnittstelle. Die Schnittstelle enthält genau eine Methode, `sayHello()`. Das Hello.java Interface

erweitert org.omg.CORBA.Object, welches Standard CORBA Objektfunktionalität zur Verfügung stellt.

- `HelloHelper.java`
  Diese finale Klasse stellt **zusätzliche Funktionalität** zur Verfügung, zum Beispiel eine `narrow`() Methode, mit deren Hilfe CORBA gecastet werden können (in den korrekten Datentyp umgewandelt werden).
- `HelloHolder.java`
  diese finale Klasse instanziert die Klasse `Hello` und liefert Operationen, mit deren Hilfe **`out` und `inout` Argumente** der IDL Beschreibung implementiert werden, in CORBA also vorhanden sind, die sich in Java aber nicht leicht darstellen lassen.

Jetzt müssen wir nur noch die Client und die Server Applikation implementieren. Die Details der obigen Dateien und Klassen braucht uns nicht weiter zu beschäftigen. In der Regel ist es das Beste, diese Dateien unverändert zu lassen.

1.12.1.1. _HelloImplBase.Java

```
/*
 * File: ./HELLOAPP/_HELLOIMPLBASE.JAVA
 * From: HELLO.IDL
 * Date: Thu May 25 12:02:37 2000
 *   By: idltojava Java IDL 1.2 Aug 18 1998 16:25:34
 */

package DemoAppl.HelloApp;
public abstract class _HelloImplBase extends
org.omg.CORBA.DynamicImplementation implements DemoAppl.HelloApp.Hello {
    // Constructor
    public _HelloImplBase() {
         super();
    }
    // Type strings for this class and its superclases
    private static final String _type_ids[] = {
        "IDL:HelloApp/Hello:1.0"
    };

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

    private static java.util.Dictionary _methods = new
java.util.Hashtable();
    static {
      _methods.put("sayHello", new java.lang.Integer(0));
     }
    // DSI Dispatch call
    public void invoke(org.omg.CORBA.ServerRequest r) {
       switch (((java.lang.Integer) _methods.get(r.op_name())).intValue())
{
           case 0: // HelloApp.Hello.sayHello
              {
              org.omg.CORBA.NVList _list = _orb().create_list(0);
              r.params(_list);
              String ___result;
                           ___result = this.sayHello();
              org.omg.CORBA.Any __result = _orb().create_any();
              __result.insert_string(___result);
              r.result(__result);
              }
              break;
            default:
```

```
            throw new org.omg.CORBA.BAD_OPERATION(0,
org.omg.CORBA.CompletionStatus.COMPLETED_MAYBE);
       }
 }
}
```

1.12.1.2. _HelloStub.java

```
/*
 * File: ./HELLOAPP/_HELLOSTUB.JAVA
 * From: HELLO.IDL
 * Date: Thu May 25 12:02:37 2000
 *   By: idltojava Java IDL 1.2 Aug 18 1998 16:25:34
 */

package DemoAppl.HelloApp;
public class _HelloStub
      extends org.omg.CORBA.portable.ObjectImpl
      implements DemoAppl.HelloApp.Hello {

    public _HelloStub(org.omg.CORBA.portable.Delegate d) {
          super();
          _set_delegate(d);
    }

    private static final String _type_ids[] = {
        "IDL:HelloApp/Hello:1.0"
    };

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

    //      IDL operations
    //          Implementation of ::HelloApp::Hello::sayHello
    public String sayHello()
 {
          org.omg.CORBA.Request r = _request("sayHello");

r.set_return_type(org.omg.CORBA.ORB.init().get_primitive_tc(org.omg.CORBA.T
CKind.tk_string));
          r.invoke();
          String __result;
          __result = r.return_value().extract_string();
          return __result;
   }

};
```

1.12.1.3. Hello.java

```
/*
 * File: ./HELLOAPP/HELLO.JAVA
 * From: HELLO.IDL
 * Date: Thu May 25 12:02:37 2000
 *   By: idltojava Java IDL 1.2 Aug 18 1998 16:25:34
 */

package DemoAppl.HelloApp;
public interface Hello
    extends org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity {
    String sayHello()
;
}
```

1.12.1.4. HelloHelper.java

```
/*
 * File: ./HELLOAPP/HELLOHELPER.JAVA
 * From: HELLO.IDL
 * Date: Thu May 25 12:02:37 2000
 *   By: idltojava Java IDL 1.2 Aug 18 1998 16:25:34
 */

package DemoAppl.HelloApp;
public class HelloHelper {
     // It is useless to have instances of this class
     private HelloHelper() { }

    public static void write(org.omg.CORBA.portable.OutputStream out,
DemoAppl.HelloApp.Hello that) {
        out.write_Object(that);
    }
    public static DemoAppl.HelloApp.Hello
read(org.omg.CORBA.portable.InputStream in) {
        return DemoAppl.HelloApp.HelloHelper.narrow(in.read_Object());
    }
   public static DemoAppl.HelloApp.Hello extract(org.omg.CORBA.Any a) {
     org.omg.CORBA.portable.InputStream in = a.create_input_stream();
     return read(in);
   }
   public static void insert(org.omg.CORBA.Any a, DemoAppl.HelloApp.Hello
that) {
     org.omg.CORBA.portable.OutputStream out = a.create_output_stream();
     write(out, that);
     a.read_value(out.create_input_stream(), type());
   }
   private static org.omg.CORBA.TypeCode _tc;
   synchronized public static org.omg.CORBA.TypeCode type() {
          if (_tc == null)
             _tc = org.omg.CORBA.ORB.init().create_interface_tc(id(),
"Hello");
      return _tc;
   }
   public static String id() {
       return "IDL:HelloApp/Hello:1.0";
   }
   public static DemoAppl.HelloApp.Hello narrow(org.omg.CORBA.Object that)
          throws org.omg.CORBA.BAD_PARAM {
       if (that == null)
           return null;
       if (that instanceof DemoAppl.HelloApp.Hello)
           return (DemoAppl.HelloApp.Hello) that;
     if (!that._is_a(id())) {
         throw new org.omg.CORBA.BAD_PARAM();
       }
         org.omg.CORBA.portable.Delegate dup =
((org.omg.CORBA.portable.ObjectImpl)that)._get_delegate();
        DemoAppl.HelloApp.Hello result = new
DemoAppl.HelloApp._HelloStub(dup);
        return result;
   }
}
```

1.12.1.5. HelloHolder.java

```
/*
 * File: ./HELLOAPP/HELLOHOLDER.JAVA
 * From: HELLO.IDL
 * Date: Thu May 25 12:02:37 2000
 *   By: idltojava Java IDL 1.2 Aug 18 1998 16:25:34
 */

package DemoAppl.HelloApp;
public final class HelloHolder
     implements org.omg.CORBA.portable.Streamable{
    //      instance variable
    public DemoAppl.HelloApp.Hello value;
    //      constructors
    public HelloHolder() {
      this(null);
    }
    public HelloHolder(DemoAppl.HelloApp.Hello __arg) {
      value = __arg;
    }

    public void _write(org.omg.CORBA.portable.OutputStream out) {
        DemoAppl.HelloApp.HelloHelper.write(out, value);
    }

    public void _read(org.omg.CORBA.portable.InputStream in) {
        value = DemoAppl.HelloApp.HelloHelper.read(in);
    }

    public org.omg.CORBA.TypeCode _type() {
        return DemoAppl.HelloApp.HelloHelper.type();
    }
}
```

1.12.2. Implementation des Servers

In unserem Beispiel besteht der Server lediglich aus zwei Klassen, Servant und Server:

- Servant Klasse `HelloServant`
  diese implementiert das `Hello` IDL Interface. Die Klasse ist eine Unterklasse von `_HelloImplBase`, welche vom `Idlj` Compiler generiert wurde. Pro IDL Operation muss im Servant eine Methode implementiert werden. In unserem Falle ist dies lediglich die `sayHello`() Methode. Servant Methoden sind normale Java Methoden.

  Der zusätzliche Programmcode, um mit dem ORB kommunizieren zu können, also zum Mashalling der Argumente und der Rückgabewerte muss vom Server und den Stubs zur Verfügung gestellt werden.

- Server Klasse, welche die `main` Methode des Servers enthält. Funktionen der Server Klasse:
  1. kreieren einer ORB Instanz
  2. kreieren einer Servant Instanz (Implementation eines CORBA Hello Objekts) und Mitteilung an den ORB
  3. bestimmen einer CORBA Objekt Referenz für den Namenskontext, indem das neue CORBA Objekt registriert wird.
  4. registrieren des neuen Objekts im Namensraum, unter dem Namen "Hello"
  5. warten auf den Aufruf der Methode / des Objekts durch einen Client

1.12.2.1. Die Servant Klasse : HelloServant.Java

```
package DemoAppl;

import DemoAppl.HelloApp._HelloImplBase;

class HelloServant extends _HelloImplBase
{

    HelloServant()
    {
    }

    public String sayHello()
    {
        return "\nHello world!!\n";
    }
}
```

1.12.2.2. Die Server Klasse : HelloServer.Java

```
package DemoAppl;

import DemoAppl.HelloApp.*;   // Stubs und Skeletons.
import org.omg.CosNaming.*;   // Nameserver

// Exceptions einbinden
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*; // CORBA Basisklassen

public class HelloServer
{
  public static void main(String args[])
  {
    try{

      // kreieren und initialisieren des ORB
      ORB orb = ORB.init(args, null);

      // Servant kreieren und ORB initialisieren
      HelloServant helloRef = new HelloServant();
      orb.connect(helloRef);

      // Root des Naming Kontextes
      org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(objRef);

      // binden des Objektes in den namensraum
      NameComponent nc = new NameComponent("Hello", " ");
      NameComponent path[] = {nc};
      ncRef.rebind(path, helloRef);

      // Clientanfragen abwarten
      java.lang.Object sync = new java.lang.Object();
      synchronized(sync){
        sync.wait();
      }

    } catch(Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
      }
  }
}



class HelloServant extends _HelloImplBase
{
  public String sayHello()
  {
    return "\nHello world!!\n";

  }
}
```

1.12.3. Implementation des Clients

Unser einfaches Beispiel führt folgende Aktivitäten durch:

- kreieren eines ORB
- bestimmen einer Referenz zum Namenskontext
- nachschlagen von "Hello" im Namenskontext / Namensraum und lesen einer Referenz auf ein CORBA Objekt
- aufrufen der Methode sayHello() des Objekts und Ausgabe des Ergebnisses.

1.12.3.1. Die Clientklasse : HelloClient.Java

```
package DemoAppl;
import DemoAppl.HelloApp.*;    // Stubs / Ausgabe aus dem IDL Compiler
import org.omg.CosNaming.*;   // Naming Service
import org.omg.CORBA.*;       // CORBA Basisklassen


public class HelloClient
{
  public static void main(String args[])
  {
    try{

      // ORB kreieren und initialisieren
      ORB orb = ORB.init(args, null);

      // Root des Namenskontextes
      org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(objRef);

      // Objektreferenz auflösen
      NameComponent nc = new NameComponent("Hello", " ");
      NameComponent path[] = {nc};
      Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));

      // Server aufrufen und ausgeben
      String Hello = helloRef.sayHello();
      System.out.println(Hello);

    } catch(Exception e) {
        System.out.println("ERROR : " + e);
        e.printStackTrace(System.out);
      }
  }
}
```

1.12.4. Der Client als Applet

Das Applet agiert analog zum normalen Client:

- kreieren eines ORB
- bestimmen einer Referenz zum Namenskontext
- nachschlagen von "Hello" im Namenskontext / Namensraum und lesen einer Referenz auf ein CORBA Objekt
- aufrufen der Methode sayHello() des Objekts und Ausgabe des Ergebnisses.

1.12.4.1. Die Appletklasse : HelloApplet.Java

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.awt.Graphics;

public class HelloApplet extends java.applet.Applet
{
    public void init() {
        try {
            // kreieren und initialisieren des ORB
            ORB orb = ORB.init(this, null);

            // Root des Namingkontextes bestimmen
            org.omg.CORBA.Object objRef =
                        orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);

            // Objektreferenz im Namenskontext auflösen
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));

            // Aufruf der Server Objektmethode und Ausgabe des Ergebnisses
            message = helloRef.sayHello();

        } catch (Exception e) {
            System.out.println("HelloApplet exception: " + e.getMessage());
            e.printStackTrace(System.out);
        }
    }
    public void paint(Graphics g)    {
        g.drawString(message, 25, 50);
    }
    String message = "";
}
```

1.12.5. Übersetzen und Starten der Applikation : Schritt für Schritt

Der Source Code für das Hello World Beispiel befindet sich im entsprechenden Verzeichnis zu diesem Kapitel. Dabei wird das Verzeichnis HelloCORBAWorld.

Wir werden Port 1050 als Standardport für den IDL Namenserver benutzen. Falls dieser besetzt ist (Abfrage mit `netstat -a`), sollten Sie einen Port angeben.

1.12.5.1. IDL Beschreibung in Java übersetzen

Sie müssen den idlj Compiler auf Ihrem PC installieren. Dieser ist auf dem Server und generiert nach anklicken ein Verzeichnis, in dem ein ReadMe und Release Informationen, sowie das Exe File befinden. Sie können diese in das \bin Verzeichnis von JDK kopieren, verschieben. Dann sollte es sich im Ausführungspfad befinden.

Die Webseite bei Sun befindet sich an folgender Adresse:
http://developer.java.sun.com/developer/earlyAccess/jdk12/idlj.html

Im Idltojava Compiler ist fix ein Preprocessor für C++ einprogrammiert (siehe oben). Auf Windows wird dabei angenommen, dass man MS Visual C++ installiert hat. Sie müssen also explizit, wie oben angegeben, den C++ Compiler ausschalten, mit dem Flag `-fno-cpp`

Der Aufruf in `idlj` sieht also folgendermassen aus:

```
idlj  -fall Hello.idl
```

und *nicht* wie bei Sun in diversen Unterlagen angegeben `idlj Hello.idl`: dabei würde die Serverseite nicht generiert!

Der IDL Compiler generiert ein Verzeichnis mit den oben besprochenen Dateien. Das Verzeichnis heisst `HelloApp`.

1.12.5.2. Compilieren der Java Dateien

Sie müssen alle *.java Dateien, inklusive Stubs und Skeletons (im Verzeichnis HelloApp) übersetzen.

```
javac *.java HelloApp/*.java
```

1.12.5.3. Starten des Namensservers

Sie müssen jetzt, falls alles fehlerfrei üebrsetzt wurde, den Namenserver starten:

```
tnameserv -ORBInitialPort 1050
```

wobei Sie den Port unbedingt angeben sollten. Standardport ist 900.
Sie müssen darauf achten, dass der Port nicht ohne das Flag -ORBInitialPort akzeptiert wird!

1.12.5.4. Starten der Hello Servers

Beim Starten der Servers wird der relevante Nameserver Port als Parameter übergeben:

```
java HelloServer -ORBInitialPort 1050
```

Sie müssen allfällige Packagenamen und Classpfade zusätzlich angeben.

1.12.5.5. Starten des Hello Clients

Auch beim Client wird der Port als Parameter übergeben:

```
java HelloClient -ORBInitialPort 1050
```

1.12.5.6. Stoppen des Servers und des Nameservices

Das Stoppen geschieht nur mit Hilfe von CTRL-C (kill auf Solaris).

1.13. Naming Service

In diesem Abschnitt wollen wir folgende Themen an Hand einfacher Beispiele genauer anschauen:

- Java IDL Name Server
- Starten des Java IDL Name Servers
- Stoppen des Java IDL Name Servers
- Beispiel Client: Objekte in den Namensraum eintragen
- Beispiel Client: Abfragen des Namespace

1.13.1. Java IDL Name Server

Der CORBA COS (Common Object Services) Naming Service stellt ein baumartiges Verzeichnis für Objektreferenzen zur Verfügung, analog einem Dateiverzeichnis. Der Naming Service von Java IDL ist eine einfache Implementation der COS Naming Service Spezifikation.

Objekt Referenzen werden im Namespace als Namen abgespeichert und jedes Objektreferenz - Namenspaar <Name, Objekt-Rerefenz> bezeichnet man als "binding", als Namensbindung.

Die Namensbindung kann weiter unterteilt werden in Namenskontexte. Diese sind selber Bindings und entsprechen den Unterverzeichnissen in einem Dateisystem. Diese sind also auch unter dem ursprünglichen Namenskontextes gespeichert.

Bei einem Neustart des Nameservices gehen die Namenskontexte verloren, nur das Root wird neu aufgebaut.

Damit ein Applet oder eine Applikation COS Naming einsetzen kann, muss der Namen und der Port des Namenserver bekannt sein. In Java kann man auch COS konforme Namensdienste einsetzen, nicht nur den im Java System mitgelieferten Tnamesrvr.

1.13.2. Starten des Java IDL Name Server

Der Java IDL Namenserver muss gestartet werden, bevor Applikationen (Clients, Server) oder Applets daraus zugreifen möchten.

Die Ausführung von

```
tnameserv.exe
```

startet den Java IDL Nameserver. Dieser läuft dann im Hintergrund, als Dienst, in einem DOS Fenster.

Standardmässig wird der Nameserver den Port 900 zum Bootstrappen verwenden. Der Nameserver hört an Port 900 und führt die ORB resolve_initial_references() und list_initial_references() Methoden aus. Ein anderer Port kann durch Angabe des Ports und des -ORBInitialPort Flags angegeben werden.

```
tnameserv -ORBInitialPort 1050
```

Clients des Nameservers müssen den Port kennen. Dies kann man mit Hilfe des Properties

```
org.omg.CORBA.ORBInitialPort
```

beim Kreieren eines ORB Objekts erreichen.

1.13.3. Stoppen des Java IDL Name Server

Um den Java IDL Nameserver zu stoppen, muss das relevante Betriebsystemkommando ausgeführt werden (in Solaris : kill; in Windows : CTRL-C).

1.13.4. Beispiel Client: Objekte einem Namespace hinzufügen

Die folgende Anwendung kommt ohne Server aus. Sie kann also zum Testen der Installation verwendet werden. Das Programm fügt Namen in ein Namespace ein und kreiert Namenskontexte. Schematisch sieht dies wiefolgt aus:

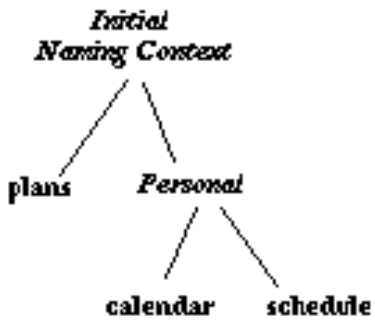


Abbildung 18 Namespace, Naming Context (Personal) und Namebinding

Plans ist eine Objektreferenz und Personal ist ein Naming Context, der zwei Objektreferenzen calendar und schedule.

Listing 7 Grundstruktur des Testprogramms Schritt für Schritt

```
import java.util.Properties;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

public class NameClient
{
   public static void main(String args[])
   {
      try {
```

Da der Java IDL Nameserver auf Port 1050 gestartet wurde, müssen wir nun dem Client dies mitteilen. Dies geschieht mit Hilfe der folgenden Programmzeilen, welche diese Property setzen.

Listing 8 Programmfragment : Setzen des Nameserver Port

```
 Properties props = new Properties();
props.put("org.omg.CORBA.ORBInitialPort", "1050");
ORB orb = ORB.init(args, props);
```

Nun ordnen wir den initialen Namenscontext (das Verzeichnisroot) der Objektreferenz `ctx` zu. objref werden wir benutzen, um die weiteren Blätter im Baum einzutragen.

Listing 9 Definition des Namenservices

```
 NamingContext ctx  =
 NamingContextHelper.narrow(orb.resolve_initial_references("NameServic
 e"));
NamingContext objref = ctx;
```

Jetzt tragen wir den Namen "plans" von Typus "text" in den Namensraum ein. Dazu verwenden wir die Methode rebind(), weil diese garantiert, dass kein Fehler beim mehrfachen Eintragen des Namens auftritt. Wir wollen die vorige Zeichnung reproduzieren, deswegen "plans"

Listing 10 erstes Binden eines Namens

```
NameComponent nc1 = new NameComponent("plans", "text");
NameComponent[] name1 = {nc1};
ctx.rebind(name1, objref);
System.out.println("plans rebind erfolgreich!");
```

Jetzt tragen wir eine neue Verzweigung im Verzeichnisbaum ein. Das Subverzeichnis heisst "Personal" und ist vom Typus "directory". Dadurch, dass wir diese Namenskomponente nc2 an die erste Wurzel binden, mit Hilfe von ctx2, gelangen wir zu einem Baum.

Listing 11 Definition eines Subverzeichnisses "Personal"

```
 NameComponent nc2 = new NameComponent("Personal",
     "directory");
NameComponent[] name2 = {nc2};
NamingContext ctx2 = ctx.bind_new_context(name2);
System.out.println("new naming context added..");
```

Jetzt vervollständigen wir das Programm, indem wir einige weitere Namensbindungen vornehmen.

```
  NameComponent nc3 = new NameComponent("schedule", "text");
  NameComponent[] name3 = {nc3};
  ctx2.rebind(name3, objref);
  System.out.println("schedule rebind sucessful!");

  NameComponent nc4 = new NameComponent("calender", "text");
  NameComponent[] name4 = {nc4};
  ctx2.rebind(name4, objref);
  System.out.println("calender rebind war erfolgreich!");


 } catch (Exception e) {
        e.printStackTrace(System.err);
}
  }
}
```

1.13.5. Beispiel Client: Abfrage des Namespace

Das folgende Beispiel illustriert die Abfrage eines Namensraumes. auch hier zeigen wir schrittweise, wie sich das Programm zusammensetzt.

Listing 12 Abfrage des Namespace : Grundlage

```
import java.util.Properties;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

public class NameClientList
{
   public static void main(String args[])
   {
      try {
```

Wir gehen wieder davon aus, dass der Java IDL Nameserver an Port 1050 auf Anfragen von Clients wartet. Wir setzen im Client diesen Port als den Standardport.

Listing 13 Setzen des Nameserver Ports im Client

```
    Properties props = new Properties();
    props.put("org.omg.CORBA.ORBInitialPort", "1050");
    ORB orb = ORB.init(args, props);
```

Nun bestimmen wir den Namenskontext, das Root / die Wurzel des Namensverzeichnisses.

Listing 14 Wurzel des Nameservers

```
NamingContext nc =
NamingContextHelper.narrow(orb.resolve_initial_references("NameService"));
```

Falls dies gelungen ist, können wir nun alle Einträge im Namespace abfragen. Dabei muss man eine Spezialität beachten:

- bei bis zu 1000 Bindungen kann man dazu den `BindingListHolder` verwenden
- bei mehr als 1000 Bindungen benötigt man dazu den BindingIteratorHolder

Listing 15 Abfrage des Namensraumes

```
   BindingListHolder bl = new BindingListHolder();
   BindingIteratorHolder blIt= new BindingIteratorHolder();
   nc.list(1000, bl, blIt);
```

Damit erhalten wir ein Array mit Bindungen. Falls keine Bindungen existieren, bricht das Programm ab.

Listing 16 Bindings bestimmen

```
        Binding bindings[] = bl.value;
        if (bindings.length == 0) return;
```

Der Rest des Programms listet alle Bindungen auf und druckt sie aus:

Listing 17 Binding

```
      for (int i=0; i < bindings.length; i++) {
            // Bestimmen der Objektreferenz für jede Bindung
            org.omg.CORBA.Object obj =
                  nc.resolve(bindings[i].binding_name);
                  String objStr = orb.object_to_string(obj);
                  int lastIx = bindings[i].binding_name.length-1;
                  // ist dies ein Namenskontext?
                  if (bindings[i].binding_type
                        == BindingType.ncontext) {
                    System.out.println( "Context: " +
                              bindings[i].binding_name[lastIx].id);
                  } else {
                  System.out.println("Object: " +
                              bindings[i].binding_name[lastIx].id);
            }
       }
       } catch (Exception e) {
        e.printStackTrace(System.err);
       }
   }
}
```

1.14. CORBA Exceptions

CORBA kennt zwei Arten von Ausnahmen:
1. Standard Systemausnahmen, welche von der OMG spezifiziert wurden;
2. und Benutzerausnahmen, die vom Benutzer selbst definiert werden können.

CORBA Ausnahmen unterscheiden sich von den in Java üblicherweise definierten Ausnahmen. Aber diese Unterschiede werden durch die Sprachbindung, das Language Mapping, weitestgehend abgefangen.

Zum Testen und Betreiben benötigen wir Kenntnisse über die spezielle Struktur dieser Exceptions. Daher werden wir uns in diesem Abschnitt mit diesem Thema befassen :
- Unterschiede zwischen CORBA und Java Exceptions
- System Exceptions
  - System Exception Struktur
  - Minor Codes
  - Completion Status
- User Exceptions
- Bedeutung der Fehlercodes Minor Codes

1.14.1. Unterschiede zwischen CORBA und Java Exceptions

In IDL spezifiziert man Exceptions mit Hilfe des *raises* Schlüsselwortes. Dies ist analog zum Schlüsselwort *throws* in Java. Immer wenn man dieses Schlüsselwort in IDL verwendet, wird eine benutzerdefinierte Ausnahme kreiert. Die Standard Systemausnahmen werden aber nicht spezifiziert, müssen und können auch nicht spezifiziert werden.

1.14.2. System Exceptions

CORBA definiert ein Set von Standard Systemexceptions, die im allgemeinen vom ORB ausgelöst werden können, um Systemfehler anzuzeigen. Solche Fehler können sein:
- serverseitige Systemexceptions, wie zum Beispiel fehlende Systemresourcen oder Aktivierungsfehler
- Fehler im Kommunikationssystem, zum Beispiel der Verlust des Kontaktes zum Objekt, ein herunter gefahreren oder abgestürzter Host oder Probleme mit dem ORB Daemon (orbd)
- clientseitige Systemexceptions, wie zum Beispiel ungültige Operanden und (fast) alle Fehler, die auftreten, bevor eine Anfrage an den ORB gesandt wurde, oder nachdem die Bearbeitung mit dem ORB abgeschlossen ist.

Alle IDL Operationen können Systemausnahmen werfen, sobald sie ausgeführt werden. Der Interface Designer muss diese Ausnahmen nicht explizit spezifizieren; dieses werden automatisch geworfen.

Dies macht sehr viel Sinn, da wie trivial die Operation auch sein kann, immer die Möglichkeit besteht, dass irgend ein Fehler auftritt.

Der Entwickler hat also keine andere Wahl, als Exceptions abzufangen und Präventionen für die Fehlerfälle sich auszudenken., speziell `java.lang.RuntimeException`.

1.14.2.1. System Exception Struktur

Alle CORBA Systemexceptions besitzen die selbe Struktur:

```
exception <SystemExceptionName> { // Beschreibung des Fehlers
    unsigned long minor;          // Detaillierung des Fehlers
    CompletionStatus completed;   // yes, no, maybe
}
```

Systemexceptions sind Subtypes von `java.lang.RuntimeException` in der Form von `org.omg.CORBA.SystemException`:

```
java.lang.Exception
 |
 +--java.lang.RuntimeException
     |
     +--org.omg.CORBA.SystemException
         |
         +--BAD_PARAM
         |
         +--//etc.
```

1.14.2.2. Minor Codes

Alle CORBA Systemausnahmen besitzen ein Minor Code Feld, eine Nummer, welche zusätzliche Informationen über die Natur des Ausfalls und den Grund des Ausfalls liefern kann. Die OMG hat die Minor Codes nicht volständig spezifiziert. Jeder ORB Anbieter kann also passende Minor Codes spezifizieren (siehe weiter unten).

1.14.2.3. Completion Codes

Alle CORBA Systemexceptions besitzen ein Abschlusscodefeld, welches den Status der Operation angibt, welche diese Ausnahme ausgelöst hat.

Folgende Abschlusscodes sind definiert:

- COMPLETED_YES
  Der Objektaufruf wurde erfolgreich abgeschlossen, bevor der Fehler auftrat.
- COMPLETED_NO
  Die Objektimplementation wurde nicht involviert bevor die Ausnahme auftrat.
- COMPLETED_MAYBE
  Der Status des Aufrufes ist unbekannt

1.14.3. User Exceptions

Benutzerdefinierte Ausnahmen sind Subtypen von `java.lang.Exceptions` und `org.omg.CORBA.UserException`:

```
java.lang.Exception
 |
 +--org.omg.CORBA.UserException
     |
     +-- Akties.BadSymbol
     |
     +--//etc.
```

Jede benutzerdefinierte Ausnahme in IDL wird in eine Java Exception umgesetzt.
Diese Ausnahmen müssen vom Entwickler spezifiziert werden.

1.14.4. Minor Code Bedeutung

Die folgende Tabelle listet die Fehler auf, die vom CORBA Anbieter weiter verfeinert werden können.

.

ORB Minor Codes und deren Bedeutung

Code	Bedeutung
BAD_PARAM Exception Minor Codes	
1	Einer Java IDL Methode wurde ein null Parameter übergeben.
COMM_FAILURE Exception Minor Codes	
1	Die Verbindung zum Host und Port, die in der Objektreferenz spezifiziert wurden, kann nicht aufgebaut werden.
2	Beim Schreiben in einen Socket, ist ein Fehler aufgetreten. Der Socket wurde geschlossen oder steht nicht mehr zur Verfügung.
3	Fehler beim Schreiben in einen Socket. Die Verbindung wurde abgebrochen.
6	Selbst nach mehreren Versuchen ist der Verbindungsaufbau zum Server nicht geglückt.
DATA_CONVERSION Exception Minor Codes	
1	Bei der Umwandlung `string_to_object`, einer ORB Methode, trat ein Fehler auf.
2	Die Länge des IOR für `string_to_object()` ist inkorrekt.
3	Der String, der als Parameter an to `string_to_object()` übergeben wurde, startet nicht mit `IOR`.
4	Die ORB Methode `resolve_initial_references` zeigt einen Fehler an: entweder ist der Host oder der Port inkorrekt oder nicht spezifiziert oder der remote Host unterstützt das Java IDL Bootstrap Protokoll nicht.
INTERNAL Exception Minor Codes	
3	IIOP Fehler
6	Unmarshalling Fehler.
7	Das Java APIs `InetAddress.getLocalHost().getHostName()` schlug fehl
8	Fehler beim Kreieren eines Listeners Evtl. ist der Port bereits besetzt.
9	Locate Status Fehler in IIOP.
10	Umwandlung eines Objekts in eine Zeichenkette svhlug fehl.
11	IIOP Message mit schlechtem GIOP v1.0 Message Type wurde gefunden
14	Unmarshaling einer User Exception schlug fehl
18	Interner Initialisierungsfehler.
INV_OBJREF Exception Minor Codes	
1	IOR Fehler.
MARSHAL Exception Minor Codes	
4	Unmarshaling einer Objekt Referenz schlug fehl.
5	Marshalling/unmarshaling Unterstützung für IDL Typ fehlt.
6	Fehlerhaftes Zeichen bein Zeichenumwandlung in String.

NO_IMPLEMENT Exception Minor Codes	
1	Dynamic Skeleton Interface ist nicht implementiert.
OBJ_ADAPTER Exception Minor Codes	
1	es wurde kein Objektadapter gefunden, der zum Objektkey passen würde.
2	Objektadapter Fehler.
4	Fehler beim Kommunikationsaufbau Servant zum ORB.
OBJ_NOT_EXIST Exception Minor Codes	
1	Locate Request Fehler : das Objekt ist unbekannt.
2	Server id des Server und Server id im Object Key der Objektreferenz stimmen nicht überein..
4	Skeleton Fehler
UNKNOWN Exception Minor Codes	
1	Unbekannte User Exception beim Unmarshalling.
3	Unbekannte Runtime Exception serverseitig.

Name Server Minor Codes und deren Bedeutung

Code	Meaning
INITIALIZE Exception Minor Codes	
150	Transient Name Service : `SystemException` bei der Initialisierung.
151	Transient Name Service : Java Exception bei der Initialisierung.
INTERNAL Exception Minor Codes	
100	eine `AlreadyBound` Exception trat in einer `rebind` Operation auf.
101	eine `AlreadyBound` Exception trat in einer `rebind_context` Operation auf.
102	Binding Type Fehler: `BindingType.nobject` or `BindingType.ncontext`.
103	Objekt Referenz war an einen Context gebunden aber nicht `CosNaming.NamingContext`.
200	`bind` Operation wurde zum wiederholten Male ausgeführt.
201	Implementation der `list` Operation führte zu einer Java Exception.
202	Implementation der `new_context` Operation führte zu einer Java Exception.
203	Implementaton der `destroy` Operation führte zu einer Java Exception.

1.15. Initialisierung

Bevor ein Java Programm ein CORBA Objekt benutzen kann, muss dieses initialisiert werden:

1. es muss ein ORB Objekt kreiert werden
2. es muss eine oder mehrere Objektreferenzen, typischerseise mindestens ein Naming Context, erhalten werden.

Zum besseren Verständnis betrachten wir diesen Prozess im Folgenden im Detail:

- Kreieren eines ORB Objekts
  - kreieren eines ORBs für eine Applikation
  - kreieren eines ORB für ein Applet
  - Argumente für die ORB.init() Methode
  - Systemeigenschaften
- bestimmen einer initialen Objektreferenz
  - Umwandlung von Objektreferenzen in Zeichenketten
  - Referenzen mit Hilfe des ORB bestimmen

1.15.1. Kreieren eines ORB Objekts

Bevor ein CORBA Objekt kreiert und benutzt werden kann, muss ein Applet oder eine Applikation zuerst ein ORB Objekt kreieren. Damit wird das Applet / die Applikation dem ORB bekannt gemacht und erhält Zugriff zu wichtigen Operationen, die im ORB Objekt definiert sind.

Applets und Applikationen kreieren ORB Instanzen leicht unterschiedlich, weil die Parameter, welche an `ORB.init()` übergeben werden, unterschiedlich sind.

1.15.1.1. Kreieren eines ORBs für eine Applikation

Das folgende Programmfragment zeigt,wie eine Applikation einen ORB kreieren könnte:

Listing 18 Initialisieren des ORBs für eine Applikation

```
import org.omg.CORBA.ORB;
public static void main(String args[]){
  try{
    ORB orb = ORB.init(args, null);
// ...
```

1.15.1.2. Kreieren eines ORB für ein Applet

Im Falle des Applets sieht die Initialisierung wiefolgt aus:

Listing 19 Programmfragment - Initialisieren des ORBs für ein Applet

```
  import org.omg.CORBA.ORB;
  public void init() {
        try {
              ORB orb = ORB.init(this, null);
// ...
```

Einige Web Browser haben bereits einen eingebauten ORB. Dies kann zu Problemen führen, falls der ORB nicht voll kompatibel sind. In diesem Falle muss man mit einigen Tricks dafür sorgen, dass der Java IDL ORB initialisiert wird.

Dies ist insbesondere der Fall bei Netscape 4.01, da diese Version einen ORB implementiert, der nicht voll kompatibel zur ORB Spezifikation ist.

Falls ein Applet in Netscape Browsern problemlos funktionieren soll, muss der Programmcode ähnlich aussehen wie das folgende Programmfragment:

Listing 20 init() Fragment

```
import java.util.Properties;
import org.omg.CORBA.*;

public class MeinApplet extends java.applet.Applet {
      public void init(){
            Properties props = new Properties();
 props.put("org.omg.CORBA.ORBClass", "com.sun.CORBA.iiop.ORB");
      ORB orb = ORB.init(this, props);
            ...
     }
}
```

1.15.1.3. Argumente für ORB.init()

Für Applikationen und Applets besitzt die Initialisierungsmethode folgende Argumente:

- **args** oder **this**
  damit kann der ORB auf Applikationsargumente oder Applet Parameter zugreifen.
- **null**
  ein `java.util.Properties` Objekt.

Die `init()` Operation benutzt diese Parameter, neben Systemeigenschaften, um Informationen, die für die Konfiguration des ORBs benötigt werden, zu erhalten.

Die Suchreihenfolge sieht wiefolgt aus:
1. die Applikation oder das Applet (erstes Argument)
2. ein `java.util.Properties` Objekt (zweites Argument), falls eines angegeben wurde.
3. das `java.util.Properties` Objekt, welches von `System.getProperties()` zurück geliefert wird.

Der erste gefundene Wert wird in der `init()` Operation eingesetzt.
Falls kein Wert gefunden wird, muss das System Standardwerte annehmen.

Um maximale Portabilität zu erreichen, ist es ratsam, die Parameter explizit anzugeben.

1.15.1.4. System Eigenschaften

System Eigenschaften, die als `Properties` Objekte definiert werden, können bei der Java VM auch mit Hilfe des -D Kommandozeilenarguments spezifiziert werden.

Die `Properties` Klasse ist im Package `java.util` definiert:

```
Class Properties
java.lang.Object
  |
  +--java.util.Dictionary
        |
        +--java.util.Hashtable
              |
              +--java.util.Properties
```

Zur Zeit können folgende Konfigurationsparameter für alle ORB Implementationen definiert werden:

- `org.omg.CORBA.ORBClass`
  Dies ist der Namen der Java Klasse, die die Schnittstelle `org.omg.CORBA.ORB` implementiert. Applets und Applikationen spezifizieren diesen Parameter nur, falls eine spezielle ORB Implementation verwendet werden soll. Der Wert für den Java IDL ORB ist `com.sun.CORBA.iiop.ORB`.

- `org.omg.CORBA.ORBSingletonClass`
  Dies ist der Name der Java Klasse, die die Schnittstelle `org.omg.CORBA.ORB` implementiert. Man erhält dieses Objekt als Rückgabewert bei Aufruf von `orb.init()` ohne Argumente. Man verwendet diese vorallem im Umfeld von gesicherten Umgebungen.

Zusätzlich zu den Standardeigenschaften oben, unterstützt Java IDL folgende Properties:

- `org.omg.CORBA.ORBInitialHost`
  Der Hostname einer Maschine, auf der ein Server oder Dämon läuft, stellt Bootstrap Services zur verfügung, wie zum Beispiel Name Services. Der Defaultwert für diese Eigenschaft ist `localhost` für Applikationen. Für Applets ist es der Applet Host, also `getCodeBase().getHost()`.

- `org.omg.CORBA.ORBInitialPort`
  Dies ist der Port des Namenservers, also 900 als Default.

Bemerkung: falls eine Eigenschaft auf der Kommandozeile spezifiziert wird, wird der Package Teil des Properties weggelassen: the `org.omg.CORBA`. Zum Beispiel wird der InitialPort auf der Kommandozeile folgendermassen spezifiziert:

```
-ORBInitialPort 800
```

Applet Parameters sollten den vollständigen Namen verwenden.

1.15.2. Bestimmen einer initialen Objektreferenz

Damit ein Applet oder eine Applikation ein CORBA Objekt benutzen kann, benötigt es / sie eine Referenz auf das Objekt. Es gibt drei Wege eine Referenz auf ein CORBA Objekt zu erhalten

1. von einer Zeichenkette, die speziell von einer Objektreferenz kreiert wurde
2. von einem andern Objekt, zum Beispiel einem Namingcontext
3. mit Hilfe der `ORB` Operation `resolve_initial_references()`

1.15.2.1. Stringified Objektreferenzen

Die erste Methode, die Konversion einer Zeichenkette in ein aktuelles Objekt, ist ORB Implementation unabhängig, sagen wir mal Java ORB unabhängig. Es liegt jedoch am Entwickler, dafür zu sorgen, dass

- das referenzierte Objekt auch tatsächlich für das Applet oder die Applikation zugänglich ist.
- die stringified Referenz auch tatsächlich erhalten wird, als Parameter oder von einer Datei

Das folgende Programmfragment konvertiert ein CORBA Objekt in einen String (stringifying):

```
org.omg.CORBA.ORB orb =  ...        // ORB Objekt bestimmen
org.omg.CORBA.Object obj =          // Objekt Referenz
String str = orb.object_to_string(obj);
// ...
```

Das folgende Programmfragment zeigt, wie ein Client ein stringified Objekt zurück verwandelt:

```
org.omg.CORBA.ORB orb =    // ORB Objekt
String stringifiedref =    // String lesen
org.omg.CORBA.Object obj = orb.string_to_object(stringifiedref);
// ...
```

1.15.2.2. Referenzen von einem ORB erhalten

Falls man keine stringified Referenz (Referenz in Zeichenformdarstellung) verwendet, muss man den ORB selbst verwenden, um die initiale Referenz zu erhalten. Dies hat den Nachteil, dass die Applikation oder das Applet ORB abhängig wird.

Das ORB Interface definiert eine Operation, `resolve_initial_references(),` die dem Bootstrapping von Objektreferenzen in neu gestarteten Applikationen und Applets dient. Die Methode übernimmt als Parameter eine Zeichenkette, die eines von mehreren Objekten bezeichnet. Die Methode liefert ein CORBA Objekt, welches der Applikation oder dem Applet entsprechend umgewandelt werden muss.

Zwei String Werte sind gegenwärtig definiert:

- `NameService`
  dieser Wert liefert eine Referenz zu einem Root Naming Context, welche verwendet werden kann, um Referenzen für Objekte, deren Namen den Applets oder Applikationen bekannt sind..
- `InterfaceRepository`
  Dieser Wert liefert eine Referenz zu einem Interface Repository, einem CORBA Objekt, welches die Interfacedefinition enthält. Die jetzige Implementation von Java IDL besitzt kein Interface Repositoty. Bei andern ORBs kann man den "InterfaceRepository" als Argument der Methode `resolve_initial_references() verwenden`.

Die Java IDL Implementation von `resolve_initial_references()` verlangt einen bereits laufenden Naming Service, dessen Host und Port als `ORBInitialHost` und `ORBInitialPort` Eigenschaften definiert wurden.

1.16. Übersicht : IDL zu Java Mapping

Die OMG hat zusammenmit Sun, HP und weiteren Firmen die Spezifikation des Java Language Mappings definiert. Die Definition ist bei der OMG gratis als PDF erhältlich. Im Folgenden betrachten wir nur einige der wichtigsten und verständlicheren Language Mappings.

1.16.1. IDL zu Java Übersicht

CORBA Objekte werden in der OMG IDL (Object Management Group Interface Definition Language) definiert. Damit sie von Java Entwicklern eingesetzt werden können, müssen die Interface Beschreibungen auf Java Klassen, Interfaces und andere Konstrukte abgebildet werden. Java IDL umfasst ein spezielles Tool, idlj, welches die entsprechenden Java Dateien generiert.

Alle IDL Konstrukte beschränken sich auf die Beschreibung der Objekte und der Signaturen der Methoden, also die Parameter und Parametertypen. Die Implementation der Methoden ist dem Programmierer überlassen.

Die folgende Tabelle fasst die wichtigsten Konzepte in IDL und Java zusammen.

IDL Konstrukt	**Java Konstrukt**
module	package
interface	interface, helper class, holder class
constant	public static final
boolean	boolean
char, wchar	char
octet	byte
string, wstring	java.lang.String
short, unsigned short	short
long, unsigned long	int
long long, unsigned long long	long
float	float
double	double
enum, struct, union	class
sequence, array	array
exception	class
readonly attribute	Zugriffsmethode
readwrite attribute	Zugriffsmethode mit veränderndem Zugriff
operation	Methode

1.17. Speichern des Zustandes - Hello World mit persistentem Zustand

Java IDL unterstützt transiente Objekte, also vergängliche Objekte. Falls der Server Prozess anhält oder neu gestartet wird, werden alle alten Objektreferenzen ungültig. In diesem Beispiel geht es um die Speicherung der Zustandsinformationen der Objekte, damit das System auch Restart Eigenschaften bekommt.

Das Beispiel unterscheidet sich kaum vom normalen HelloWorld. Im Beispiel wird eine Zeichenkette ("Hello World") im Server Objekt abgespeichert. Das Server Objekt erinnert sich an diese Zeichenkette und kann nach dem Neustart des Servers dem Client diese Zustandsvariable wieder zur Verfügung stellen.

Alle Änderungen im Programmcode, die neu hinzu kommen, werden im folgenden Text andersfarbig dargestellt.

1.17.1. Interface Definition (`Hello.idl`)

```
module HelloApp{
    interface Hello {
        exception cantWriteFile{};
        exception cantReadFile{};
        string sayHello(in string message) raises (cantWriteFile);
        string lastMessage()raises (cantReadFile);
    };
};
```

Da unser neues Hello Objekt den Zustand speichern muss, benötigen wir Dateizugriffe, mit Hilfe von read und write, und damit gemäss Java auch Exceptions, da beim Lesen oder Schreiben ein Fehler auftreten könnte. In CORBA benötigen wir diese Exceptions bereits in der IDL Beschreibung, da ja Stub und Skeleton vom IDL Compiler generiert werden.

Die Modifikation des Hello World ist die, dass nun die `sayHello` Methode eine Zeichenkette als Argument hat und diese in einer Datei abgespeichert wird. Mit Hilfe der `lastMessage`() Methode kann diese Zeichenkette wieder gelesen und an den Client übermittelt werden.

Der IDL Compiler generiert ein Verzeichnis PersistentHelloWorld/HelloApp (HelloApp ist der Namen des Moduls in IDL), wie bereits im ersten HelloWorld Programm. Darin befinden sich die üblichen Dateien, wie Stub, Skeleton, Helper und Holder Klasse.

Neu hinzu kommt ein weiteres Unterverzeichnis HelloPackage, in dem sich alle Exceptions befinden `HelloApp/ HelloPackage`:

```
cantReadFile.java
cantReadFileHelper.java
cantReadFileHolder.java
cantWriteFile.java
cantWriteFileHelper.java
cantWriteFileHolder.java
```

Die meisten dieser Klassen sind final. Wie diese Klassen konkret eingesetzt werden, sehen wir weiter unten.

1.17.2. Implementation des Servers (`HelloServer.java`)

Wichtig: hier wird die Datei für die Speicherung explizit angegeben!

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

class HelloServant extends _HelloImplBase{
    public String sayHello(String msg)
        throws HelloApp.HelloPackage.cantWriteFile{
        try {
                synchronized(this) {
                   File helloFile = new
                                File("helloStorage.txt");
                   FileOutputStream fos = new
                                FileOutputStream(helloFile);
                   byte[] buf = new byte[msg.length()];
                   msg.getBytes(0, msg.length(), buf, 0);
                   fos.write(buf);
                   fos.close();
                }
        } catch(Exception e) {
                throw new
                        HelloApp.HelloPackage.cantWriteFile();
        }
        return "\nHello world !!\n";
    }
    public String lastMessage()
        throws HelloApp.HelloPackage.cantReadFile{
        try {
                synchronized(this) {
                   File helloFile = new
                                File("helloStorage.txt");
                   FileInputStream fis = new
                                FileInputStream(helloFile);
                   byte[] buf = new byte[1000];
                   int n = fis.read(buf);
                   String lastmsg = new String(buf);
                   fis.close();
                   return lastmsg;
                }
        } catch(Exception e) {
                throw new HelloApp.HelloPackage.cantReadFile();
        }
    }
}

public class HelloServer {

    public static void main(String args[])   {
        try{
            // ORB
            ORB orb = ORB.init(args, null);
            // registrieren
            HelloServant helloRef = new HelloServant();
            orb.connect(helloRef);
            // root naming context
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef =
                        NamingContextHelper.narrow(objRef);
```

```
            // binden
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            ncRef.rebind(path, helloRef);
            // Client Anfragen abwarten
           java.lang.Object sync = new java.lang.Object();
           synchronized (sync) {
                sync.wait();
            }
        } catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
}
```

Die `sayHello` und `lastMessage` Methoden Implementationen werfen Ausnahmen, die wir in IDL definiert haben. Da gelesen und geschrieben wird, müssen wir die entsprechenden IO Exceptions in Java implementieren.

Da der Server von mehreren Clients Anfragen erhalten kann, werden die Methodenzugriffe im Server synchronisiert. Der Client darf nicht synchronisieren, da sonst verklemmungen auftreten könnten, falls der Client stirbt.

Das Programm für den Hello Server bleibt unverändert.

Eine Verbesserung könnte man erreichen, indem man den Dateinamen als Eingabeparameter definieren würde,

1.17.3. Implementation des Clients (`HelloClient.java`)

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class HelloClient
{
    public static void main(String args[])
    {
        try{
            // ORB initialisieren
            ORB orb = ORB.init(args, null);

            // root des naming context
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef =
                        NamingContextHelper.narrow(objRef);

            // Object Referenz auflösen
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            Hello helloRef =
                       HelloHelper.narrow(ncRef.resolve(path));

            // Aufruf des Servers und Ausgabe

             String oldhello = helloRef.lastMessage();
            System.out.println(oldhello);
            String hello = helloRef.sayHello(args[0]);
            System.out.println(hello);

        } catch (Exception e) {
            System.out.println("ERROR : " + e) ;
            e.printStackTrace(System.out);
        }
    }
}
```

Der Client kann die zuletzt verwendete Zeichenkette mit `lastMessage` abfragen und mit `sayHello` eine neue Zeichenkette abspeichern.

1.17.4. Konkretes Vorgehen : Bau, Übersetzen und Start der Applikation

Im Folgenden wird angenommen, dass der Nameserver an Port 1050 gestartet wurde.

Das Vorgehen im Einzelnen:

1. kreieren der Quelldateien
2. `idlj` ausführen; damit werden Stub und Skeleton generiert:
   ```
   idlj -fall PersistentHello.IDL
   ```
3. übersetzen der .`java` Dateien
   ```
   javac .java HelloApp\.java
   ```
4. starten des Nameservers
   ```
   tnameserv -ORBInitialPort 1050
   ```
5. starten des Persistent Hello Servers
   ```
   java PersistentHelloServer -ORBInitialPort 1050
   ```
6. starten der PersistentClients
   ```
   java HelloClient "Meldung" -ORBInitialPort 1050
   ```

Das vollständige Beispiel mit aktuellen Versionen des Programmcodes und der Batch Dateien finden Sie auf dem Server.

1.18. Hello World mit Callback Objekt

Wichtig: beachten Sie die Unterschiede des Servant Objekts auf der Server und auf der Client-Seite. Die beiden Versionen sind unterschiedlich und erweitern unterschiedliche Klassen!

Nachdem wir nun einen einfachen Server mit persistenter Speicherung haben (letzter Abschnitt) möchten wir natürlich mehr. Das folgende Beispiel ist fast genau so einfach, illustriert aber einige generell interessante Komnzepte und Entwurfsmuster :

- in der Regel reagieren Client Programme auf Änderungen im Server, zum Beispiel beimEintreffen neuer Börsenkurse, neuer Wetterdaten, neuer Sensordaten in einer Gebäudeüberwachung, ...
- die Lösung kann so aussehen, dass:
  - der Client periodisch den Server abfragt
  - der Client informiert wird, wenn sich etwas auf dem Server ereignet hat (**Entwurfsmuster : Publish / Subscribe**; der Server publiziert, der Client hat ein Abo):
    "CallBack" Lösung

Die zweite Option ist intellektuell herausfordernder als die erste. Also nehmen wir uns dieses Muster vor und illustrieren den konkreten Einsatz des Publish / Subscribe Entwurfsmuster an einem einfachen Beispiel:

auch hier markieren wir den Programmcode in rot, falls es sich von der HelloWorld Lösung unterscheidet.

1.18.1. Interface Definition (`Hello.idl`)

```
module HelloApp
{
    interface HelloCallback
    {
        void callback(in string message);
    };


    interface Hello
    {
        string sayHello(in HelloCallback objRef,
                                in string message);
    };
};
```

Eine `HelloCallback` Schnittstelle wurde definiert. Diese wird vom Client implementiert werden.

Die `sayHello` Methode wurde so modifiziert, dass eine *Objektreferenz als Argument* übergeben werden kann. Diese *Objektrerefenz ist das Callback Objekt* und kann vom Server eingesetzt werden. Das zweite Argument ist die Zeichenkette, die der Server an den Client zurück senden soll.

1.18.2. Implementation des Servers (`HelloServer.java`)

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

class HelloServant extends _HelloImplBase {
    public String sayHello(HelloCallback callobj, String msg){
        callobj.callback(msg);
        return "\nHello world !!\n";
    }
}

public class HelloServer {
    public static void main(String args[]){
        try{
            // kreiere und initialisiere den ORB
            ORB orb = ORB.init(args, null);

            // kreiere den Servant und registriere beim ORB
            HelloServant helloRef = new HelloServant();
            orb.connect(helloRef);

            // root naming context
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef =
                NamingContextHelper.narrow(objRef);

            // binden der Object Reference in Naming
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            ncRef.rebind(path, helloRef);

            // warten auf Clients
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) {
                sync.wait();
            }

        } catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
}
```

Die `sayHello` Methode unterscheidet sich nur durch die Callback Option.

1.18.3. Implementation des Clients (`HelloClient.java`)

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

class HelloCallbackServant extends _HelloCallbackImplBase
{
    public void callback(String notification)
    {
        System.out.println(notification);
    }
}

public class HelloClient {
    public static void main(String args[]){
        try{
            // kreieren und initialisieren des ORB
            ORB orb = ORB.init(args, null);

            // root naming context
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef =
                NamingContextHelper.narrow(objRef);

            // auflösen der Object Reference im Naming
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            Hello helloRef =
                HelloHelper.narrow(ncRef.resolve(path));

            HelloCallbackServant helloCallbackRef = new
                        HelloCallbackServant();
            orb.connect(helloCallbackRef);

            // Aufruf des Server Objekts und Ausgabe
            String hello =
                helloRef.sayHello(helloCallbackRef,"\ntest..\n");
            System.out.println(hello);

        } catch (Exception e) {
            System.out.println("ERROR : " + e) ;
            e.printStackTrace(System.out);
        }
    }
}
```

Der Client implementiert das `HelloCallbackServant` Objekt.

Die `HelloClient.main` Methode instanziert das Callback Objekt und übergibt es an den Server. Der Client muss das Callback Objekt auch beim ORB registrieren

1.18.4. Konkretes Vorgehen für Hello World Callback

Wir nehmen wieder an, dass der Nameserver an Port 1050 gestartet wurde.

Das Vorgehen im Einzelnen:

1. kreieren der Quelldateien
2. `idlj` ausführen; damit werden Stub und Skeleton generiert:
   ```
   idlj -fall CallbackHello.IDL
   ```
3. übersetzen der `.java` Dateien
   ```
   javac .java HelloApp\.java
   ```
4. starten des Nameservers
   ```
   tnameserv -ORBInitialPort 1050
   ```
5. starten des Callback Hello Servers
   ```
   java CallbackHelloServer -ORBInitialPort 1050
   ```
6. starten der CallbackClients
   ```
   java HelloClient "Meldung" -ORBInitialPort 1050
   ```

Auch das sollte ohne grosse Schwierigkeiten funktionieren. Aber von der Architektur her, ist diese Lösung für viele Anwendungen sehr viel besser als unnötiges, periodisches Abfragen. Der Kommunikationsaufwand wird reduziert; der Serveraufwand wird erhöht (Sun lässt danken).

Auch hier befinden sich aktuelle Versionen der Programme plus Batch Dateien auf dem Server.

1.19. Berücksichtigung von Vererbung

In unseren Beispielen hatten wir bisher sehr viel Glück: die Servant Klasse erweiterte die `ImplBase` Klasse, die vom `idlj` Compiler generiert wurde.

Das führt in Java zu einem Problem: weil Java keine Mehrfacvererbung kennt, kann der Servant keine andere Oberklasse als `ImplBase` besitzen. Das ist aber sehr unschön, weil eine CORBA systembedingte Klasse dadurch die Möglichkeiten der Java Programmierung einschränkt.

Im folgenden Beispiel zeigen wir, wie man dieses Problem beseitigen oder lösen kann:

- `HelloServant` erbt die Implementation von einer Klasse `HelloBasic`.
- zur Laufzeit werden alle Anfragen an `HelloServant` an eine von `idlj` generierte Klasse delegiert (**Entwurfsmuster : Delegation**)

Auch in diesem Beispiel basieren wir das gesamte Programm auf der ersten Version des HelloCORBAWorld Programms. Alle Ergänzungen und Änderungen sind in rot geschrieben.

1.19.1. Interface Definition (`Hello.idl`)

Die Moduldefinition sieht genau gleich aus:

```
module HelloApp
{
    interface Hello
    {
        string sayHello();
    };
};
```

Aber der Idlj Compiler wird mit einem speziellen Flag aufgerufen:

```
idlj -fallTie Hello.idl
```

Dadurch werden zwei zusätzliche Dateien im `HelloApp` Subverzeichnis generiert:

`_HelloOperations.java`
: Die Servant Klasse wird dieses Interface implementieren.

`_HelloTie.java`
: Diese Klasse agiert als Skeleton, empfängt Aufrufe vom ORB und delegiert sie an den Servant, der die Arbeit erledigen muss..

1.19.1.1. _HelloOperations.java

```
/*
 * File: ./HELLOAPP/_HELLOOPERATIONS.JAVA
 * From: HELLO.IDL
 * Date: Thu May 25 12:02:37 2000
 *   By: idlj Java IDL 1.2 Aug 18 1998 16:25:34
 */

package HelloApp;
/**

 */public interface _HelloOperations {
     String sayHello()
;
}
```

1.19.1.2. _HelloTie.java

```
/*
 * File: ./HELLOAPP/_HELLOTIE.JAVA
 * From: HELLO.IDL
 * Date: Thu May 25 12:02:37 2000
 *   By: idlj Java IDL 1.2 Aug 18 1998 16:25:34
 */

package HelloApp;
public class _HelloTie extends HelloApp._HelloImplBase {
    public HelloApp._HelloOperations servant;
    public _HelloTie(HelloApp._HelloOperations servant) {
           this.servant = servant;
    }
    public String sayHello()
    {
        return servant.sayHello();
    }
}
```

1.19.2. Implementation des Servers (`HelloServer.java`)

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;


class HelloBasic {
    public String sayHello(){
        return "\nHello world !!\n";
    }
}

class HelloServant extends HelloBasic implements _HelloOperations{ }



public class HelloServer {

    public static void main(String args[])
    {
      try{
          // kreieren und initialisieren des ORB
          ORB orb = ORB.init(args, null);

          // Servant kreieren und beim ORB registrieren
          HelloServant servant = new HelloServant();
          Hello helloRef = new _HelloTie(servant);
          orb.connect(helloRef);

          // root naming context
          org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
          NamingContext ncRef =
                  NamingContextHelper.narrow(objRef);

          // binden der Object Reference in Naming
          NameComponent nc = new NameComponent("Hello", "");
          NameComponent path[] = {nc};
          ncRef.rebind(path, helloRef);

          // warten auf Kunden
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) {
                sync.wait();
            }

      } catch (Exception e) {
          System.err.println("ERROR: " + e);
          e.printStackTrace(System.out);
      }
    }
}
```

1.19.3. Implementation des Clients (`HelloClient.java`)

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class HelloClient {
    public static void main(String args[]){
      try{
          // kreieren und initialisieren des ORB
            ORB orb = ORB.init(args, null);
            // root naming context
          org.omg.CORBA.Object objRef =
                  orb.resolve_initial_references("NameService");
          NamingContext ncRef =
                  NamingContextHelper.narrow(objRef);
            // Objekt Referenz in Naming auflösen
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            Hello helloRef =
                  HelloHelper.narrow(ncRef.resolve(path));
          // aufruf des Server Objekts und Ausgabe
          String hello = helloRef.sayHello();
          System.out.println(hello);
      } catch (Exception e) {
          System.out.println("ERROR : " + e) ;
          e.printStackTrace(System.out);
      }
    }
}
```

1.19.4. Konkretes Vorgehen zum Bau von Callback Hello World

Wir nehmen wieder an, dass der Nameserver an Port 1050 gestartet wurde.

Das Vorgehen im Einzelnen:

7. kreieren der Quelldateien
8. `idlj` ausführen; damit werden Stub und Skeleton generiert:
   ```
   idlj -fall -ftie CallbackHello.IDL
   ```
9. übersetzen der `.java` Dateien
   ```
   javac .java HelloApp\.java
   ```
10. starten des Nameservers
    ```
    tnameserv -ORBInitialPort 1050
    ```
11. starten des Callback Hello Servers
    ```
    java CallbackHelloServer -ORBInitialPort 1050
    ```
12. starten der PersistentClients
    ```
    java HelloClient "Meldung" -ORBInitialPort 1050
    ```

Der indrekte Aufruf von Objektmethoden, die Delegation an andere Objekte, ermöglicht oft ein saubereres Design und eine bessere Architektur der Systeme als traditionellere Bauweisen. Das Entwurfsmuster zählt sicher zu den zehn bekanntesten.

1.20. Das Dynamic Skeleton Interface DSI

Das DynamicSkeleton Interface DSI erlaubt es Servern ein Servantobjekt (ein CORBA Objekt) einzusetzen, ohne zur Übersetzungszeit das Interface des Objekts zu kennen. Der Server verwendet dynamisch generierten Code an Stelle des generierten Skeleton Codes.

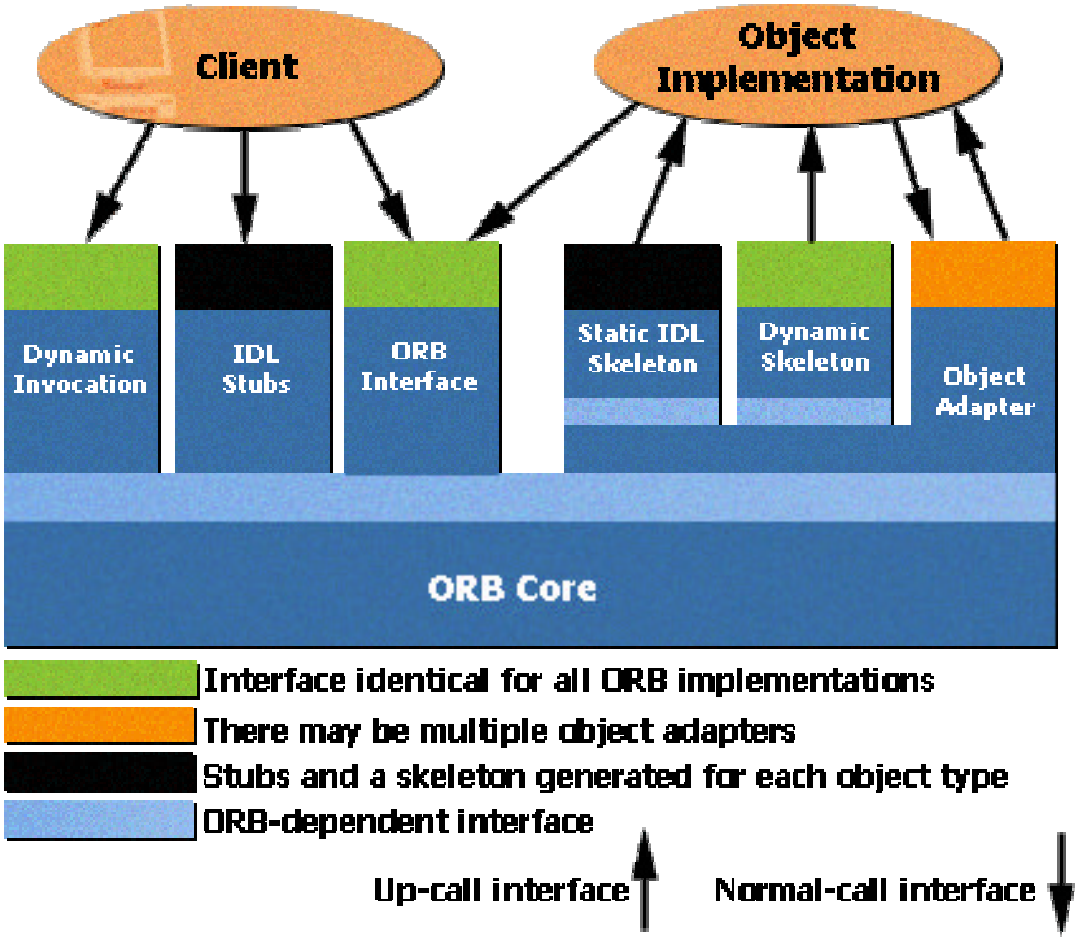


Abbildung 19 CORBA Übersicht

Client- und Serverseitig kennt man zwei Interfacetypen: statische und dynamische. Meistens werden die statischen eingesetzt. Die Programme werden dadurch effizienter, aber auch starrer.

Das Pendant zum DSI, Dynamic Skeleton Interface, auf der Client Seite ist das DII, das Dynamic Invocation Interface.

1.20.1. Warum soll man DSI überhaupt einsetzen?

DSI erlaubt es einem CORBA System mit nicht-CORBA Umgebungen zu kommunizieren. Solche Brücken / Bridges erlauben es CORBA Applikationen Methoden von nicht-CORBA Systemen zu nutzen, zum Beispiel Microsoft COM "Objekte".

Ein dynamisches Serverobjekt wird in diesem Falle so implementiert, dass die Methodenaufrufe vom COM Server verstanden werden. Es liegt aber bei Ihnen den entsprechenden Code zu schreiben.

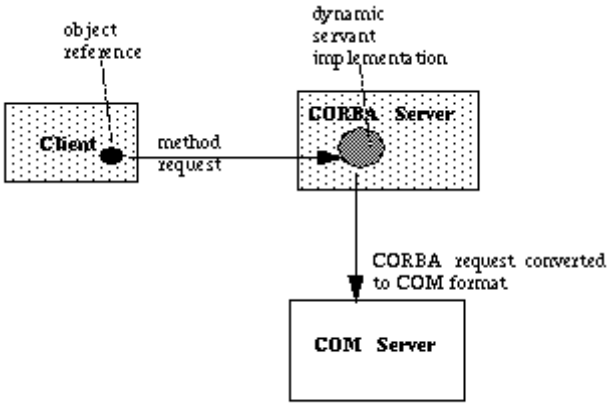


Abbildung 20 Das DSI Skeleton Interface

1.20.2. DSI - Ein Beispiel

Wichtig: dieses Beispiel ist nicht voll ausprogrammiert, da DII einen vollen ORB benötigt, beispielsweise VisiBroker.

Als erstes müssen wir das Interface beschreiben

```
module HelloApp
{
    interface Hello
    {
        string printHelloArgs(in string arg1, in short arg2);
    };
};
```

Das folgende Beispiel ist eine einfache Implementierung eines dynamischen Serverobjektes. Allerdings sind nur die wesentlichen Elemente enthalten, um das Beispiel nicht zu kompliziert zu machen.Die Bridge ist zum Beispiel nicht implementiert.

```
import HelloApp.*;

import java.util.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import java.io.*;
import org.omg.CORBA.*;

// Servant muss DynamicImplementation erweitern
class HelloServant extends DynamicImplementation
{

   // speichern der Repository ID für das implementierte Interface
   static String[] myIDs = {"IDL:JavaIDL/DSIExample:1.0"};
   ORB orb;

   // Reference für den ORB kreieren
   HelloServant(ORB orb) {
      this.orb = orb;
   }
   // invoke() handelt Requests
   public void invoke(ServerRequest request) {
      try {
         System.out.println("DSI: invoke called, op = "+request.op_name());

                  // die NVList speichert die Parameter
         NVList nvlist = orb.kreiere_list(0);

                  // pro Methodenname ein if
         if (request.op_name().equals("printHelloArgs") == true) {

                           // Any für jedes Argument
               Any any1 = orb.kreiere_any();
               any1.insert_string("");
               nvlist.add_value("arg1", any1, ARG_IN.value);

               Any any2 = orb.kreiere_any();
               any2.insert_string("");
               nvlist.add_value("arg2", any2, ARG_IN.value);
```

```
                    // NVListe an den Request geben, um den Wert zu erhalten
                request.params(nvlist);

                System.err.println("Argument 1: In value: " + nvlist.item(0).value().extract_string());

                System.err.println("Argument 2: In value: " + nvlist.item(1).value().extract_short());

                TypeCode result_tc = orb.get_primitive_tc(TCKind.tk_void);
                Any result_any = orb.kreiere_any();
                result_any.type(result_tc);

                request.result(result_any);
            }
        }
        catch (Exception ex) {
            ex.printStackTrace();
            System.out.println("DSIExample: Exception thrown: " + ex);
        }
    }

// implementieren  einer _ids Methode um die Repository ID für Interfaces zu erhalten
public String[] _ids() {
        return myIDs;
}

// HelloServer implementiert wie üblich
public class HelloServer {

    public static void main(String args[])
    {
        try{
            // ORB
            ORB orb = ORB.init(args, null);

            // Serverobjekt kreieren und anmelden ORB
            HelloServant helloRef = new HelloServant(orb);
            orb.connect(helloRef);

            OutputStream f = new
FileOutputStream(System.getProperty("user.home")+System.getProperty("file.separator")+"DSI.ior");
            DataOutputStream out = new DataOutputStream(f);
            String ior = orb.object_to_string(helloRef);
            out.writeBytes(ior);
            out.close();

            System.out.println("IOR is "+ ior);

            // warten auf Anfragen
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) {
                sync.wait();
            }

        } catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
}
```

1.21. Schlussbemerkungen

Einige grosse Fragen stellen sich nach dem Durcharbeiten des Skripts sicher:

- wann muss welche Klasse erweitert werden?
  Antwort: sicher eine ImplBase Klasse.
  Merken Sie sich das allgemeine Schema aus dem Bankenbeispiel, da die Callback, Persistent ... Beispiele Spezialanwendungen sind.
- was muss man installieren, damit auch alle Beispiele funktionieren?
  Sie benötigen einen vollen ORB, beispielsweise VisiBroker oder einen gratis ORB, wie MICO, omniORB oder JacORB.
  welche Probleme habe ich beim Einsatz von CORBA?
  Das Hauptproblem ist, dass Sie synchron zwischen Objekten kommunizieren:
  ein Objekt will etwas von einem anderen, ruft dessen Methode auf, wartet bis die Antwort da ist und fährt dann weiter.
  Alternativ dazu könnte man asynchron kommunizieren: beispielsweise mit einem Messaging System. Dabei werden Meldungen in eine Warteschlange gestellt und der Benutzer holt sie ab, wann immer er Zeit hat, auch in Echtzeit!
  Die Verwaltung der vielen IDLs ist ein weiteres Problem: sobald Sie mit ändern anfangen ist die Hölle los!
- was ist IOR?
  IOR steht für interoperable object representation; es geht also um eine Darstellung der Objekte, welche unabhängig von der Plattform und dem bestimmten ORB ist.

Falls Sie sich für CORBA ineressieren oder auch einfach tolle Bilder zu diesem Thema ansehen möchten, kann ich Ihnen die Referenz weiter vorne (Orfali) wärmstens empfehlen. Leider ist der Lerneffekt oft gering, aber die Zeichnungen sind gut!