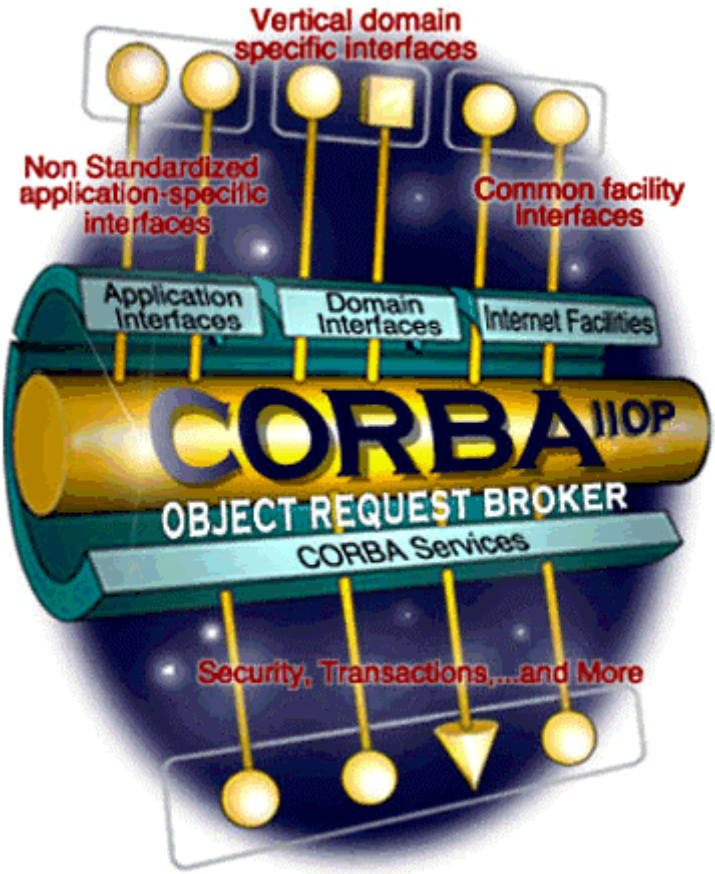
Steinbeis-Transferzentrum Softwaretechnik Esslingen

Corba 2.0

Eine Einführung in Architektur, Spezifikationen und Programmierung



Adam, Hofer, Kaller, Kaul, Landwehr
Version 1.1

Inhaltsverzeichnis

1 Motivation

Die ständig steigenden Anforderungen an Datenverarbeitungssysteme zur Lösung immer komplexer werdender Aufgaben erfordern die **Flexibilisierung** von Hard- und Software. Wo vor wenigen Jahren fast ausschließlich Mainframes die Datenverarbeitung in Unternehmen und wissenschaftlichen Einrichtungen beherrschten, sind heute aufgrund des technologischen Fortschritts der Mikroprozessoren überwiegend PC's und Workstation's vernetzt oder als Insellösung im Einsatz. Während die Insellösungen vor allem noch im Heimcomputer-Bereich anzutreffen sind, findet in Unternehmen die Kommunikation und die Datenübertragung über gut ausgebaute und **leistungsfähige Netzwerke** statt. Selbst ein Zugang in das weltumspannende Internet mit seinen riesigen Informationsresourcen gehört heute praktisch zum Standard am Arbeitsplatz.

Gleichzeitig mit der Flexibilisierung der Hardware haben sich auch die Anforderungen bei der Entwicklung von Software verändert. Die großen Softwaresysteme mit ihren riesigen **globalen Datenmengen** und nicht mehr überschaubaren Funktionalitäten lassen eine **Wartung** und Weiterentwicklung von Software kaum mehr zu. Einen möglichen Ausweg aus dieser Misere bietet die **Objektorientierung** an. Der Einsatz von Objekten als gekapselte Einheiten aus Daten und Funktionalitäten verbirgt Implementierungsdetails nach außen und vereinfacht dadurch die Entwicklung und Wartung von Software. Implementierungen dieser Objekte können jederzeit ohne Probleme verändert und erweitert werden, solange ihre **Schnittstellen** kompatibel bleiben.

Die Vorteile, die durch den Einsatz von objektorientierten Technologien entstehen, sind:

- Kostenersparnis durch Teilen von Ressourcen
- erhöhte Wart- und Wiederverwendbarkeit von Softwarekomponenten
- Leistungssteigerung durch Parallelverarbeitung
- vereinfachte Zusammenarbeit

usw.

Bis vor kurzem steckte verteiltes Arbeiten mangels geeigneter Technologien und Werkzeuge noch im Anfangsstadium. Blicken wir zurück in das Jahr 1989, so war die Betriebssystemlandschaft zu diesem Zeitpunkt geprägt von MS-DOS, MacOS und diversen Unix-Derivaten, z.B. AIX oder Ultrix. **Verteilte Systeme** begannen langsam, in industriellen Softwaresystemen eine zunehmende Rolle zu spielen. Die Programmierung dieser Systeme erfolgte primär unter direkter Nutzung von **Kommunikationsprotokollen** (z.B. TCP/IP) oder - komfortabler - über Sockets oder **entfernte Prozeduraufrufe** (Remote Procedure Calls, RPC).

Die relativ komplexe und systemnahe Programmierung von **Interprozeß-kommunikation** blieb jedoch in der Hand einiger weniger Experten. In jedem Software-Projekt mußte praktisch das Rad neu erfunden werden, sprich: selbst rudimentäre Dienste, wie zum Beispiel Namensdienste, erforderten eine Eigenentwicklung. Zudem ergaben sich häufig architekturelle Probleme, da das verwendete **Programmierparadigma** - zum Beispiel Objektorientierung - sich im

allgemeinen nicht mit dem verwendeten **Kommunikationsparadigma** - zum Beispiel datenstrombasierte Übertragung unstrukturierter Rohdaten - deckte, ganz zu schweigen von der Unterstützung heterogener Landschaften mit unterschiedlichen Betriebssystemen und Programmiersprachen.

Mit genau diesen Problemen vor Augen wurde 1989 die Object Management Group (OMG) gegründet. Die Zielsetzung des heute weltweit größten Softwarekonsortiums bestand von Anfang an darin, eine geeignete **Middleware**[1] bereitzustellen, um die problemlose Interaktion von Softwarekomponenten, selbst in heterogenen[2] Umgebungen zu gewährleisten. Das Ergebnis dieser Standardisierung war der **CORBA-Standard** (CORBA = Common Object Request Broker Architecture).

CORBA war jedoch nicht die einzige Entwicklung in dieser Richtung: Microsofts **DCOM/OLE** (DCOM = Distributed Component Object Model, OLE = Object Linking & Embedding) oder das **verteilte Java-Konzept** von Sun-Microsystems (RMI = Remote Method Invocation) sind bekannte und häufig in der Praxis eingesetzte Alternativen.

Bevor nun auf die CORBA-Spezifikation näher eingegangen wird, soll zunächst ein kurzer Überblick über DCOM und RMI gegeben werden.

1.1 Überblick über DCOM und RMI

1.1.1 DCOM

Um die Interaktion von Softwarekomponenten auf einem Rechner zu ermöglichen, hat die Firma Microsoft das „Component Object Model“ (COM), die Grundlage für das „Object Linking and Embedding“ (OLE), entwickelt. Später erkannte Microsoft die Notwendigkeit, eine Plattform für verteilte Anwendungen anzubieten, und setzte vor das COM ein ‘D’ (für Distributed).

Bei **DCOM** können die einzelnen Softwarekomponenten in unterschiedlichen Programmiersprachen, beispielsweise C++, Java oder VisualBasic usw. implementiert sein. Im Gegensatz zu CORBA handelt es sich bei DCOM nicht um die Spezifikation einer verteilten Architektur, sondern um eine **Implementierung**. DCOM ist mittlerweile fester Bestandteil von Windows 95 und Windows NT4.0 und somit im Desktop-Bereich automatisch verfügbar. Allerdings gibt es keine Portierungen auf andere Systemplattformen.

[1] Unter Middleware versteht man die gesamte Kommunikation zwischen 2 Prozessen auf verschiedenen Rechnern, die oberhalb der Netzwerkprotokoll-Ebene (also auf der Ebene der Anwendungsschicht) anzusiedeln ist.

[2] Der Begriff „heterogen“ umfaßt neben unterschiedlicher Netztechnologien und Betriebssystemen auch die Programmiersprachenunabhängigkeit.

Architektur:

Das Objekt, mit dem kommuniziert werden soll, kann sich bei DCOM (gilt auch bei RMI und CORBA) grundsätzlich an drei verschiedenen Standorten befinden:

- im selben Prozeß
- in einem anderen Prozeß, aber auf demselben Rechner
- auf einem anderen Rechner

Die Objekte kennen den **Standort** ihrer jeweiligen Kommunikationspartner nicht; sie wissen nur, wie sie über DCOM miteinander Kontakt aufnehmen und kommunizieren können.

Der Zugriff auf die Eigenschaften einer entfernten Softwarekomponente erfolgt bei DCOM über eindeutig festgelegte Schnittstellen, die wie bei CORBA in einer **Definitionssprache** (Interface Definition Language, IDL) beschrieben werden. Für jede **Schnittstellenbeschreibung** werden ein Kopf- und ein Rumpfbereich definiert.

Der **Kopfbereich** beschreibt die allgemein benötigten **Informationen** der Schnittstelle, z.B. die Schnittstellen-ID. Die Schnittstellen-ID, die sogenannte **GUID** (Global Unique Identifier), liegt als 128-Bit Zahl vor. Diese GUID wird mittels eines Hilfsprogramms erzeugt und ist mit nahezu hunderprozentiger Wahrscheinlichkeit weltweit eindeutig. Im **Rumpfbereich** werden alle von der Schnittstelle zur Verfügung gestellten Eigenschaften definiert. Alle Schnittstellen sind von `IUnknown` abgeleitet.

Mit Hilfe des midl-Compilers wird aus den in der IDL-Datei beschriebenen Schnittstellen eine `TypeLibrary` (`tlb`-Datei) erzeugt. Über ein `TLB`-Tool werden dann die für das **Marshalling** und **Unmarshalling** benötigten Stubs und Skeletons generiert. Diese Klassen liegen im Gegensatz zu CORBA als binäre Dateien vor.

Eine Server-Klasse implementiert die aus der IDL-Beschreibung generierten Schnittstellen. Diese werden durch einen speziellen Compiler übersetzt und in einer `Registry`-Datei (z.B. der `NT-Registry`) registriert. Damit steht diese Klasse für ferne Zugriffe per DCOM zur Verfügung.

Das Binden des Client-Stubs an die Server-Implementierung erfolgt über einen auf dem **GUID** basierenden Mechanismus. Nach dem Binden kann über den Client-Stub auf die Server-Implementierung durch einen „**Remote Procedure Call**“ (RPC) zugegriffen werden. Neben selbstdefinierten Schnittstellen gibt es auch noch Standardschnittstellen, beispielsweise die Schnittstelle `IUnknown`. Anhand dieser Schnittstelle ist es dem Client möglich, über den Aufruf der Methode `QueryInterface` alle Schnittstellen des entfernten Objekts zu ermitteln. Dadurch wird neben dem statischen auch der **dynamische Zugriff** auf Methoden des Server-Objekts möglich.

1.1.2 RMI

Seit dem Release 1.1 des Java Developer's Kit (JDK) kann der Programmierer über RMI relativ einfach auf entfernte Objekte zugreifen. Durch seine **Beschränkung** auf **Java** bleibt dem Entwickler das Erlernen einer Definitionssprache, wie z.B. CORBA-IDL, erspart. Mit Ausnahme der Auswertung einer speziellen Exception unterscheidet sich der Aufruf einer lokalen Methode nicht von dem Aufruf einer Methode eines entfernten Objektes.

Architektur:

Mit Hilfe von RMI kann aus einer Java-Anwendung über alle Java VMs hinweg die Methode eines anderen Java-Objektes angesprochen werden, das sich in einem anderen **Adressraum** befindet.

Dazu sind die folgenden Schritte notwendig:

- Registrierung des Objekts auf dem Server mittels des **RMI-Namensdienstes**. Hierbei wird dem zu verteilenden Objekt eine URL („Uniform Resource Locator") zugewiesen.
- Binden der Objektreferenz auf dem Client.

Jeder Aufruf an den Client-Stub wird über die **Objektreferenz** (URL) an das entfernte Objekt weitergeleitet und das Ergebnis an das aufrufende Objekt zurückgegeben.

Stub und Skeleton-Klasse werden durch den RMI-Compiler **rmic** generiert.

Verteilte Objekte **erben** entweder von der Klasse `UniCastRemoteObject` des Pakets `java.rmi`, oder sie müssen dem RMI-System zur **Laufzeit** bekanntgegeben werden. In jedem Fall müssen sie aber das Interface `java.rmi.Remote` implementieren, damit sie als verteilte Objekte angesprochen werden können.

2 OMG und OMA

2.1 Die Object Management Group (OMG) und ihre Ziele

Die **Object Management Group** (OMG) ist ein internationaler, herstellerübergreifender, nicht profitorientierter Zusammenschluß von Hardwareherstellern, Softwareentwicklern, Netzwerkbetreibern und kommerziellen Anwendern von Computersystemen. Sie wurde im **Mai 1989** von 8 Unternehmen gegründet (3Com, American Airlines, Canon, Data General, Hewlett-Packard, Phillips Telecommunications, Sun Microsystems and Unisys), die allesamt an die Vorteile der Objektorientierung glaubten und auf diese Weise ein wenig **Ordnung** in das Chaos auf dem **objektorientierten Markt** bringen wollten. Mittlerweile gehören ihr weltweit über 800 (Stand: März 1998) Firmen, Universitäten und Institutionen an. Darunter sind alle bedeutenden System-und Softwarefirmen wie Microsoft, IBM, Sun, SAP oder Deutsche Telekom AG.

Die Firmen teilen sich in die Gruppen der beitragenden (sogenannte contributing und domain contributing), beeinflußenden (influencing) und prüfenden (auditing) Mitglieder. Darüber hinaus gibt es die Mitgliedergruppen Behörden (government) und Hochschulen (university).

Das Hauptquartier der OMG befindet sich in Framingham, USA. Um international präsent zu sein, besitzt die OMG aber auch noch Marketingpartner in Großbritannien, Japan, Australien, Indien und der Bundesrepublik Deutschland.

Aufgabe der OMG ist es, die Theorie und die praktische Umsetzung von **Objekttechnologien** für die Entwicklung von verteilten Systemen zu **fördern**. Sie hat sich das Ziel gesetzt, einen allgemeinen architektonischen Rahmen für objekt-orientierte Applikationen zu liefern. In diesem Zusammenhang erarbeitet sie allgemeine Richtlinien und industriell verwertbare **Spezifikationen** in Form von allgemeinen Architekturen, wie auch konkreten Komponenten für die Verteilung und Zusammenarbeit objektorientierter Softwarebausteine und stellt sie ihren Mitgliedern bereit. Das bedeutet, die OMG spezifiziert nur die **Schnittstellen** sowie **Funktionalitäten**. Die konkrete Umsetzung in Produkte ist die Aufgabe der Softwarehersteller.

Das Ergebnis dieser Arbeiten soll die Entwicklung von Anwendungen ermöglichen, die ohne Änderungen am Programmcode auf verschiedenen **Hardwarearchitekturen** und **Betriebssystemen** arbeitsfähig sind.

Es liegt allerdings auf der Hand, daß auch die Spezialisten der OMG nicht von vornherein alle Probleme und Möglichkeiten bedacht haben können.
So beinhalten manche Dokumente teilweise **unwichtige**, **überkomplizierte** oder schlichtweg **fehlerhafte** Spezifikationen. Um diese Spezifikationen zu erkennen und gegebenenfalls **zurückzunehmen** oder **anzupassen**, hat die OMG einen einfachen Automatismus entwickelt, der die Spreu vom Weizen trennen soll. Sollte **1 Jahr** nach der Bekanntgabe einer Spezifikation diese in kein Produkt umgesetzt worden sein, so wird diese Spezifikation zurückgenommen oder bei Bedarf überarbeitet.

2.2 Die Object Management Architecture (OMA)

Grundlage aller **Standardisierungsaktivitäten** der OMG bildet die **Object Management Architecture** (OMA). Sie wurde im November 1990 innerhalb eines Dokumentes mit dem Namen „Object Management Architecture Guide“ von der OMG veröffentlicht. Neben einer allgemeinen **Einführung** in die Thematik beschreibt die OMA eine **Referenzarchitektur**, auf die sich alle anderen Spezifikationen der OMG beziehen. Die OMA definiert quasi eine Ablaufumgebung, die sich über mehrere heterogene Systeme ausdehnen kann und in der alle grundlegenden Bausteine Objekte sind.

Eine in der OMA-Umgebung ablaufende Anwendung besteht aus einer Vielzahl miteinander kooperierender Objekte. Alle Objekte, die gemeinsam eine bestimmte Aufgabe erfüllen (z.B. das Auffinden von Objekten im Netz, die Umformung eines Objektzustandes in einen Datenstrom usw.) werden in der OMA-Terminologie als **Komponente** bezeichnet. Die von ihnen wahrgenommene Aufgabe heißt **Dienst** (Service). Jede Komponente wird anhand der von ihr bereitgestellten Dienste eine der folgenden 5 Gruppen (siehe Abbildung) zugeordnet: Object Request Broker (ORB), Object Services, Common Facilities, Domain Services und Application Objects.

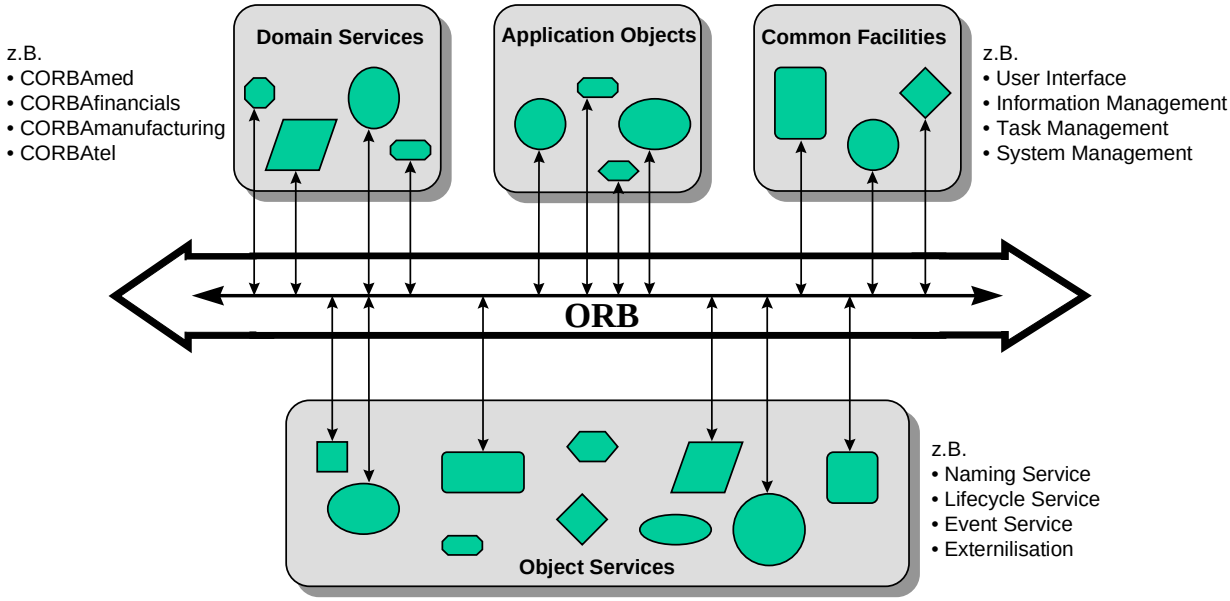


Abbildung 2-1: Aufbau von Diensten der OMA

Die **Kommunikation** zwischen Objekten wird immer mit Hilfe der Dienste des Object Request Brokers (ORB) abgewickelt. Dieser ermöglicht einen weitgehend **transparenten** Informationsaustausch zwischen Objekten einer Anwendung, die sich auf verschiedenen Rechnern befinden können, unabhängig vom verwendeten Betriebssystem und der Programmiersprache, in der das Objekt implementiert ist.

Kernbestandteil der OMA ist die Beschreibung von Schnittstellen mit Hilfe der festgelegten **Schnittstellenbeschreibungssprache** Interface Definition Language (IDL). Mit ihrer Hilfe werden alle äußerlich sichtbaren Eigenschaften von Objekten

(Attribute, Methoden, mögliche Ausnahmen (Exceptions), für die Kommunikation benötigte Datentypen) festgelegt. Für die **Integration** in die OMA muß eine Anwendung nur die OMG-konformen Schnittstellen unterstützen.

Es folgt nun ein kurzer Überblick über die Bestandteile der OMA:

- Object Request Broker (ORB)
  Der zentrale Bestandteil der OMA ist der ORB, der den **Kommunikationsmechanismus** für alle Objekte im System bereitstellt. Den ORB kann man sich in Analogie zur Rechnerarchitektur als „**Software-Bus**“ vorstellen, in den einzelne Steckkarten (Client- und Server-Objekte) eingesteckt werden. Da der ORB die **Grundlage** für alle anderen Architekturbestandteile darstellt, wurde er folgerichtig zuerst von der OMG ausgearbeitet und 1991 im Dokument „The Common Object Request Broker: Architecture and Specification“ veröffentlicht. Dieses Dokument legt alle **Schnittstellen** und **Spezifikationen** eines ORBs fest. Diese Spezifikation wurde mehrfach überarbeitet und um wesentliche Bestandteile erweitert. Die aktuelle Version wurde im Februar 1998 veröffentlicht und hat die Versionsnummer 2.2.
  Der ORB gestaltet die Kommunikation der Objekte in der OMA **verteilungstransparent**. Die Clients brauchen keine Rücksicht darauf nehmen, wie das aufgerufene Objekt implementiert ist und wie es aufgerufen, erzeugt und gelöscht werden kann. Der ORB bildet damit die Grundlage zum Bau von Anwendungen, die aus **verteilten Objekten** zusammengesetzt sind.
  Damit eine Instanz der OMA über mehrere Systeme ausgedehnt werden kann, müssen Aufrufe transparent zwischen verschiedenen ORBs weitergereicht werden. Dies wird durch standardisierte Schnittstellen des **Internet Inter-ORB Protocols** (IIOP) ermöglicht, daß die Kommunikation zwischen den ORBs verschiedener Systeme unterstützt.

- Object Services (CORBAservices)
  CORBAservices, wie man sie auch nennt, sind allgemeine, **elementare** und **betriebssystemähnliche Dienste**, die für CORBA-basierte Anwendungen essentiell sind. Aus diesem Grund schreibt die OMG zwingend vor, daß die CORBAservices auf jedem OMA-konformen System implementiert sein müssen. Die Schnittstellen und Funktionen der CORBAservices sind in der Spezifikation „Common Object Services Specification“ festgelegt. Wenn eine CORBA-basierte Anwendung nur diese Dienste benutzt, so ergibt sich für die Anwendung ein gewisses Maß an **Unabhängigkeit** vom konkret benutzten System.
  Typischerweise umfaßt die Spezifikation eines CORBAservice eine Menge von in der IDL formulierten **Schnittstellen**, die Objekte bereitstellen oder benutzen müssen, um diesen Dienst verwenden zu können. Die CORBAservices werden oft implementiert, indem Funktionalität vom Betriebssystem und Standard-Software wie beispielsweise Datenbanken in **Objekte** eingebettet wird. Wie grundlegend die CORBAservices sind, kann man an den Beispielen für bereits festgelegte Dienste erkennen:
    - Naming Service: Auffinden von Objekten mit Hilfe logischer Namen.
    - Event Service: Behandlung von Ereignissen.
    - Persistence Object Service: Dauerhafte Speicherung von Objekten.
    - Object Transaction Service: Durchführung von Transaktionen mit Objekten.

- Common Facilities (CORBAfacilities)
  Die Spezifikationen der CORBAfacilities legen die Schnittstellen und Funktionen von relativ komplexen, **endbenutzerorientierten** Diensten fest. Sie sind in vielen Anwendungen nützlich, aber nicht essentiell. CORBAfacilities werden auch als **horizontale**, d.h. breit einsetzbare **Dienste** bezeichnet.
  Typischerweise umfaßt die Spezifikation eines CORBAfacility eine Menge von in IDL formulierten Schnittstellen, die Objekte bereitstellen oder benutzen müssen, um den Dienst verwenden zu können. Das wohl bekannteste Beispiel für ein CORBAfacility ist die Architektur für Verbunddokumente OpenDOC.

- Domain Interfaces (CORBAdomains)
  Die Spezifikationen der CORBAdomains legen die Schnittstellen und Funktionen von Diensten fest, die für bestimmte **Anwendungsbereiche** besonders interessant sind. Die CORBAdomains werden auch als **vertikale**, d.h. relativ stark spezialisierte, aber trotzdem verbindlich festgelegte **Dienste** bezeichnet. Beispiele wären Dienste für Finanzen oder Telekommunikation.

- Application Objects (Application Interfaces)
  Application Objects sind hoch spezifische Objekte, die unter Benutzung der Objekte anderen Objektgruppen eine Anwendung realisieren. Sie sind das, was ein Benutzer als die konkrete **Anwendung** bezeichnen würde.

3 CORBA

CORBA (Common Object Request Broker Architecture) ist eine von der OMG spezifizierte **Architektur** zur Kommunikation zwischen im Netz verteilten Objekten. Die CORBA-Spezifikation definiert die Aufgaben und Schnittstellen des eigentlichen **Kern** der OMA, den ORB. Auf Basis dieses Architekturvorschlages werden die Produkte, wie der VisiBroker von Visigenic oder der ORBIX von Iona, implementiert. Dabei wird in CORBA die transparente **Verteilung** von Objektaufrufen definiert. Diese Objekte können sowohl auf dem selben Rechner, auf verschiedenen Rechnern innerhalb eines lokalen Netzes oder auf verschiedenen Rechnern im Internet liegen.

Damit bieten die auf der Basis CORBA vorhandenen Produkte Möglichkeiten, größere objektorientierte Applikationen/Systeme zu verteilen, d.h. verschiedene Teile auf verschiedene Rechner auszulagern.

Hierbei agieren diese einzelnen Teile der Applikation nicht unabhängig voneinander, sondern tragen gemeinsam zur Ergebniserbringung bei. Sie müssen deshalb miteinander **kommunizieren**, d.h. Daten oder Ereignisse austauschen.

Dies ist unter UNIX z.B. über Sockets schon lange möglich, aber nicht unbedingt komfortabel und mit folgenden Nachteilen behaftet:

- nicht transparent, d.h. lokale Kommunikation unterscheidet sich von Kommunikation über Netzwerkgrenzen hinweg.
- plattformabhängig
- sprachabhängig
- zeitaufwendig

Der grundsätzliche Aufbau dieser Kommunikation stellt nachfolgendes Diagramm dar:

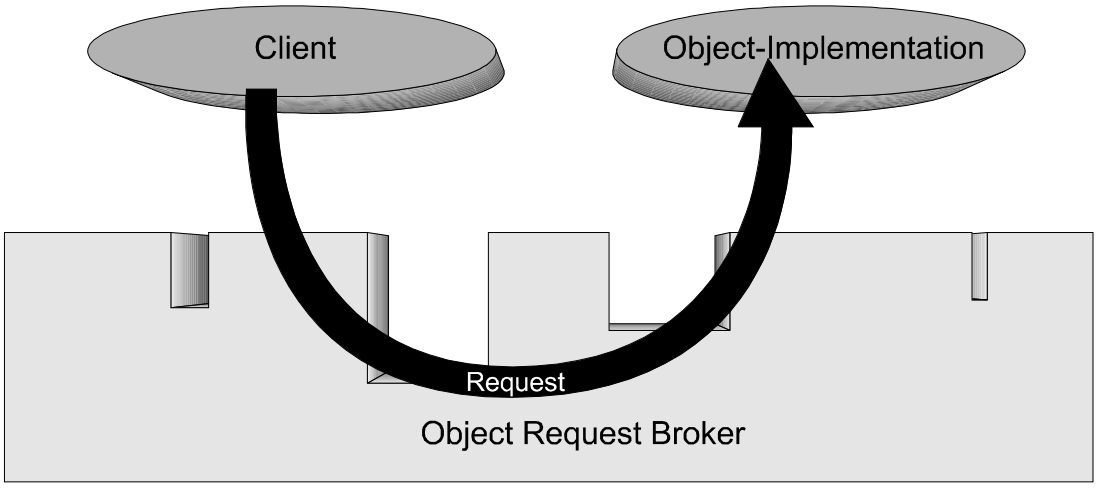


Abbildung 3-1: Kommunikation zwischen Client und Objektimplementation

Ein Client sendet einen Request an ein Objekt, welches im Netz beliebig verteilt werden kann, vorausgesetzt es kann identifiziert werden. Der Object Request Broker sorgt dann für die verteilungstransparente Weiterleitung von diesem Request.

Genau um diese Anforderungen zu erfüllen, wurde CORBA von der OMG spezifiziert.

3.1 CORBA Spezifikation

Das Word CORBA setzt sich aus den Anfangsbuchstaben von **C**ommon **O**bject **R**equest **B**roker **A**rchitecture zusammen, d.h. das „A“ in CORBA steht für Architektur und genau dies ist die Aufgabe der OMG, eine **Architektur** zu spezifizieren, mit deren Hilfe ein Produkt entwickelt werden kann, welches sich an die Architektur und damit an die Spezifikation hält.

Die Spezifikation umfaßt hierbei folgende Punkte:

- **Object Request Broker Core**
  Definition der Schnittstellen und Aufgaben des Kommunikations-Bus.

- **Interface Definition Language**
  Definition einer Sprache, um Schnittstellen zu beschreiben.

- **Abbildung der IDL auf Programmiersprachen**
  Vorschrift, wie die IDL Sprachkonstrukte auf verschiedene Programmiersprachen

umgesetzt werden müssen.

- **Stubs und Skeletons** (statischer Aufruf)
  Aufbau der durch den IDL-Compiler generierten Stubs (Client-Seite)und Skeletons (Server-Seite).

- **Dynamic Invocation Interface**
  Definition einer dynamischen Schnittstelle für Clients

- **Dynamic Skeleton Interface**
  Definition einer dynamischen Schnittstelle für Objektimplementationen

- **Object Adapter** (z.B. Basic-Object-Adapter)
  Definition einer Schnittstelle für die Regestrierung von Implementationsobjekten und Methodenaufrufe.

- **Interface und Implementation Repositories**
  Persistente Speicherung der Dynamic Invocation Interfaces und der Objektimplementationen in Datenspeichern (Repositories).

- **Interoperabilität**
  Kommunikation zwischen den ORBs verschiedener Hersteller, d.h. Client und Server können miteinander kommunizieren, obwohl die Entwickler unterschiedliche ORBs verwenden oder ORBs, die auf ganz verschiedenen Netzen liegen.

3.2 Architektur

In jedem Buch über CORBA gibt es auf jeden Fall ein Diagramm, welches in jedem Buch ziemlich ähnlich aussieht. Dieses Diagram stellt die grundlegende Architektur von CORBA dar. Da dieses Diagramm überall zu finden ist, fehlt es natürlich hier auch nicht:

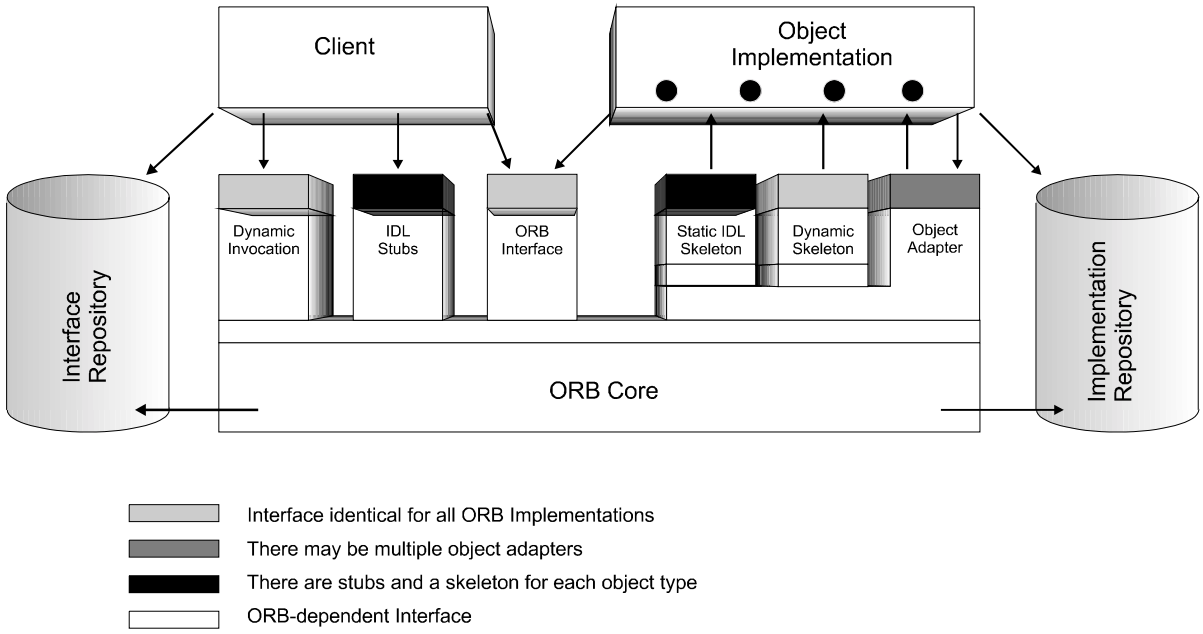


Abbildung 3-2: Aufbau von CORBA

Dieses Diagramm stellt die Verbindung zwischen den einzelnen Komponenten dar.

Nachfolgend eine kurze Beschreibung aus Sicht des Clients:

Ein Client besitzt zu folgenden Komponenten eine Verbindung:

- **Dynamic Invocation Interface**
  wenn zum Kompilierzeitpunkt das Interface noch nicht definiert wurde.

- **IDL Stubs**
  wenn das Interface zum Kompilierzeitpunkt definiert wurde.

- **ORB Interface**
  Dieses Interface wird benötigt, um die Verbindung zum ORB zu initiieren und um eine Objekt-Referenz zu erhalten.

  Der Client kann entweder über ein Dynamic Interface oder über ein IDL Stub einen Request zu einem **Implementationsobjekt** senden. Als Transportplattform steht der ORB Core zur Verfügung.

Nachfolgend eine kurze Beschreibung aus Sicht der Implementationsobjekte (im Server):

Ein Server besitzt zu folgenden Komponenten eine Verbindung:

- **Static Skeletons**
  wenn das Interface zum Kompilierzeitpunkt definiert wurde.

- **Dynamic Skeleton Interface**
  wenn zum Kompilierzeitpunkt das Interface noch nicht definiert wurde.

- **Object Adapter**
  Registrierung des Implementationsobjektes, damit einem Request ein Ziel zugeordnet werden kann. Ein Request wird vom OA zum entsprechenden Implementationsobjekt weitergeleitet.

- **ORB Interface**
  Dieses Interface wird benötigt, um die Verbindung zum ORB zu initiieren, eine Referenz zum Object Adapter zu erhalten und um mit dem ORB zu kommunizieren.

3.3 ORB Schnittstellen / Core

3.3.1 ORB Schnittstellen

In Bezug auf den Object Request Broker sind drei verschiedene Typen von Schnittstellen definiert:

- standardisierte Schnittstellen, die für alle ORBs identisch sind (ORB, DII, DSI)

Über diese Schnittstellen wird mit dem ORB kommuniziert um:

- den ORB zu initialisieren
- eine Referenz auf den Objektadapter zu erhalten
- dynamisch zur Laufzeit einen Request zu generieren, obwohl zum Kompilierzeitpunkt das Interface noch nicht definiert war
- dynamisch zur Laufzeit ein Implementationsobjekt zu generieren, obwohl zum Kompilierzeitpunkt das Interface noch nicht definierbar war.

- Objekttyp spezialisierte Schnittstellen, die unabhängig vom ORB sind (Skeletons, Stubs)
  Diese Schnittstellen werden durch den IDL-Compiler generiert und hängen deshalb vom Implementationsobjekt ab. Der IDL- Compiler generiert folgende Schnittstellen:

- Stubs
  Diese Schnittstelle bietet einen netz-transparenten Zugriff auf das Implementationsobjekt, d.h. der Zugriff erfolgt unabhängig davon, ob sich das Implementationsobjekt lokal oder entfernt befindet.
- Skeletons
  Diese Schnittstelle definiert den Rahmen des Implementationsobjektes und den Zugriff auf den Object Adapter, um das Implementationsobjekt zu registrieren.

  Das Implementationsobjekt muß den Rahmen mit der Implementierung füllen.

- Schnittstellen für die verschiedenen Arten von Objekt Adaptern
  (Hängt vom ORB ab)
  Um die verschiedenen Object Adapter ansprechen zu können, werden diese Schnittstellen definiert. Die Aufgabe des Object Adapters ist es, das Implementationsobjekt zu registrieren, um bei einem Request das angesprochene Objekt lokalisieren zu können.

  Die Art der Registrierung hängt vom Object Adapter ab.

3.3.2 ORB Core

- Stellt den Kommunikationsmechanismus zum Austausch von Requests zwischen lokalen und entfernten Objekten bereit

- Transparente Client/Server Kommunikation in Bezug auf:
  - Ort des Objektes
  - Implementierung des Objektes
  - Ausführungszustand des Objektes
  - verwendeter Kommunikationsmechanismus
  - Erzeugung eines Objektes

Aus prozeßtechnischer Sicht gibt es verschiedene Methoden, wie der Client, das Implementationsobjekt und der ORB zusammenarbeiten:

- vollständig getrennte Prozesse
- getrennte Prozesse für Client/Server
- ein einziger Prozeß

3.3.2.1 vollständig getrennte Prozesse

Hier befinden sich der Client, das Implementationsobjekt und der Object Request Broker in drei verschiedenen Prozessen.

Dieses Verfahren ist gegeben, wenn der ORB z.B. als Demon ein Bestandteil des **Betriebssystemes** ist. Hierbei besteht schon die Kommunikation zwischen Client und ORB und Implementationsobjekt und ORB aus einer **Interprozesskommunikation**.

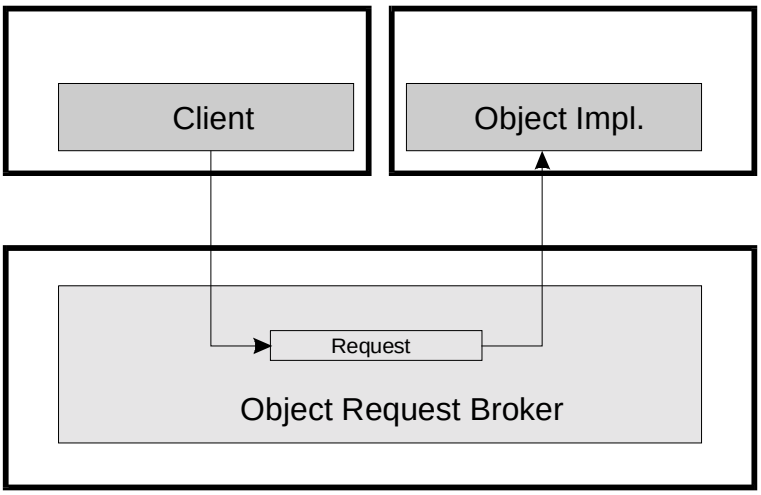


Abbildung 3-3: vollständig getrennte Prozesse

3.3.2.2 getrennte Prozesse für Client/Server

Eine weitere Art für die Gruppierung von Client, Implementationsobjekt und ORB liegt bei der Aufteilung des ORB's, so daß der ORB sich im gleichen Prozeß wie der Client und im **gleichen Prozeß** wie das Implementationsobjekt befindet.
Bei dieser Methode wird der ORB in Form einer Bibliothek an die beiden Prozesse gebunden und ist kein Bestandteil des Betriebssystems.

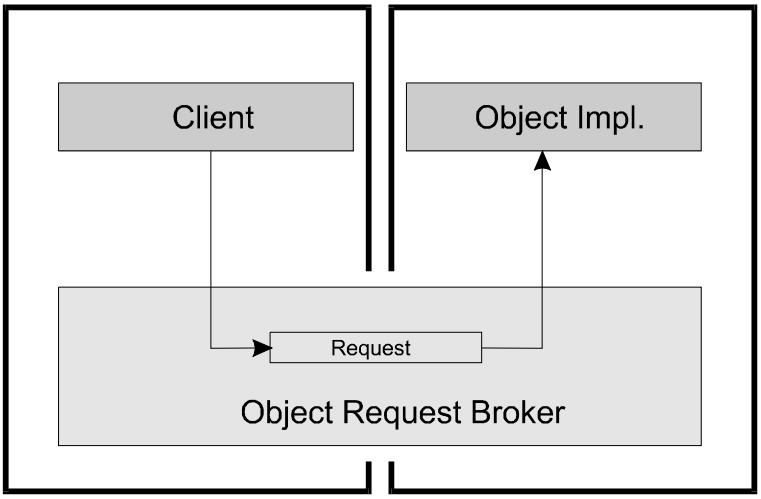


Abbildung 3-4: getrennte Prozesse Client/Server

3.3.2.3 ein einziger Prozeß

Eine Variation des obigen Punktes ist, daß sich alle drei Komponenten Client, Implementationsobjekt und ORB im **selben Prozeß** befinden.

Auch hier wird der ORB in Form einer Bibliothek an den Prozeß gebunden und ist nicht Bestandteil des Betriebssystemes.

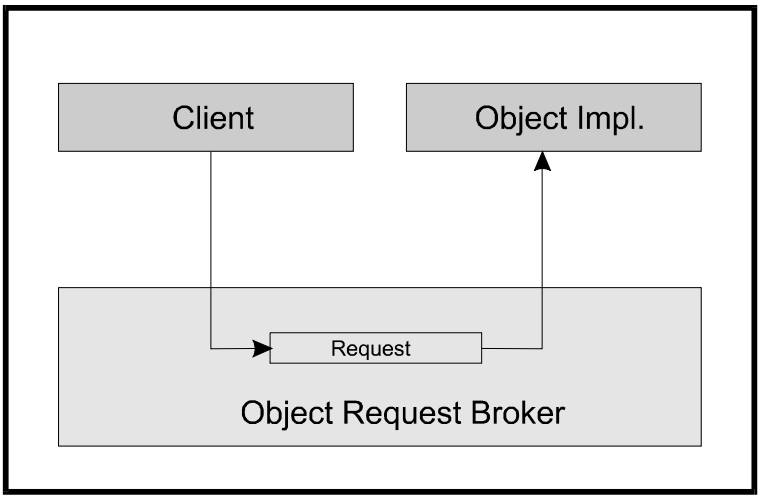


Abbildung 3-5: gemeinsamer Prozeß

3.4 IDL und die verschiedenen Programmiersprachen

Damit Client und Server miteinander kommunizieren können, müssen beide eine **Vereinbarung** über die Schnittstelle treffen, d.h. beiden müssen die gleiche Sicht auf die Schnittstelle besitzen.

Damit eine Schnittstelle definiert werden kann, wird eine **Sprache** benötigt, die im Falle von CORBA einen **objektorientierten** Ansatz besitzen muß und die möglichst einfach auf verschiedene Programmiersprachen umgesetzt werden kann, da die Definition einer Schnittstelle nur die **Beschreibung**, aber nicht die Realisierung beinhaltet, d.h. eine Schnittstellenbeschreibung muß in eine Programmiersprache übersetzt und mit der Programmiersprache implementiert werden.

Für CORBA wurde eine „Schnittstellen-Beschreibungssprache“ definiert, die Interface Definition Language (IDL).

Nachfolgendes Bild zeigt diesen Punkt. Der Client und der Server verwenden die Schnittstellenbeschreibung, um daraus die Stubs und Skeletons zu generieren. Hierbei können der Client und der Server unterschiedliche Programmiersprachen verwenden.

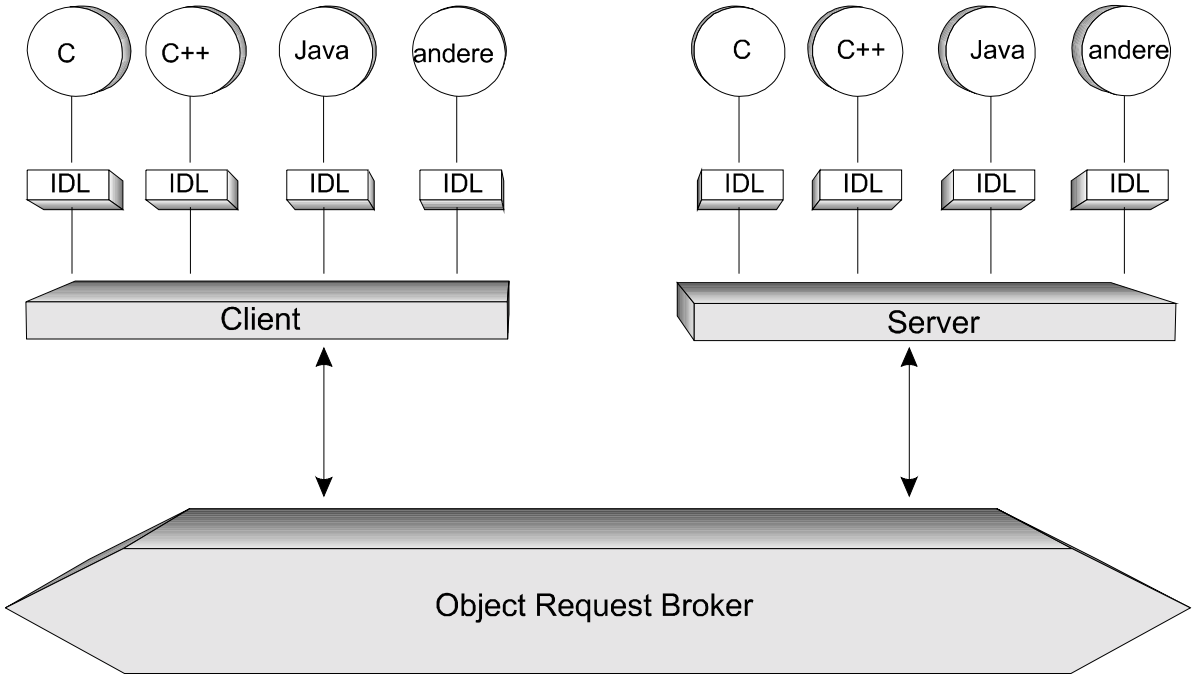


Abbildung 3-6: Einbindung verschiedener Programmiersprachen

3.5 Definition von Diensten

Wie wird nun ein Dienst definiert und realisiert.

Kochrezept:

Die in der **plattformunabhängigen** Beschreibungssprache IDL definierten Schnittstellen werden durch den IDL Compiler in einen Stub und in einen Skeleton in der gewählten Programmiersprache, z.B. Java, **übersetzt**. Der Skeleton wird **implementiert**, d.h. der vom IDL-Compiler erzeugte Code wird „mit Leben gefüllt“. Danach wird der Stub für den Client, das Skeleton und das **Implementationsobjekt** für den Server kompiliert und Schwups kommt ein kleiner Client und ein kleiner Server heraus.

Dies sind im Wesentlichen die Schritte, um einen Client und Server zu generieren.

Nachfolgendes Diagramm zeigt diesen groben Ablauf:

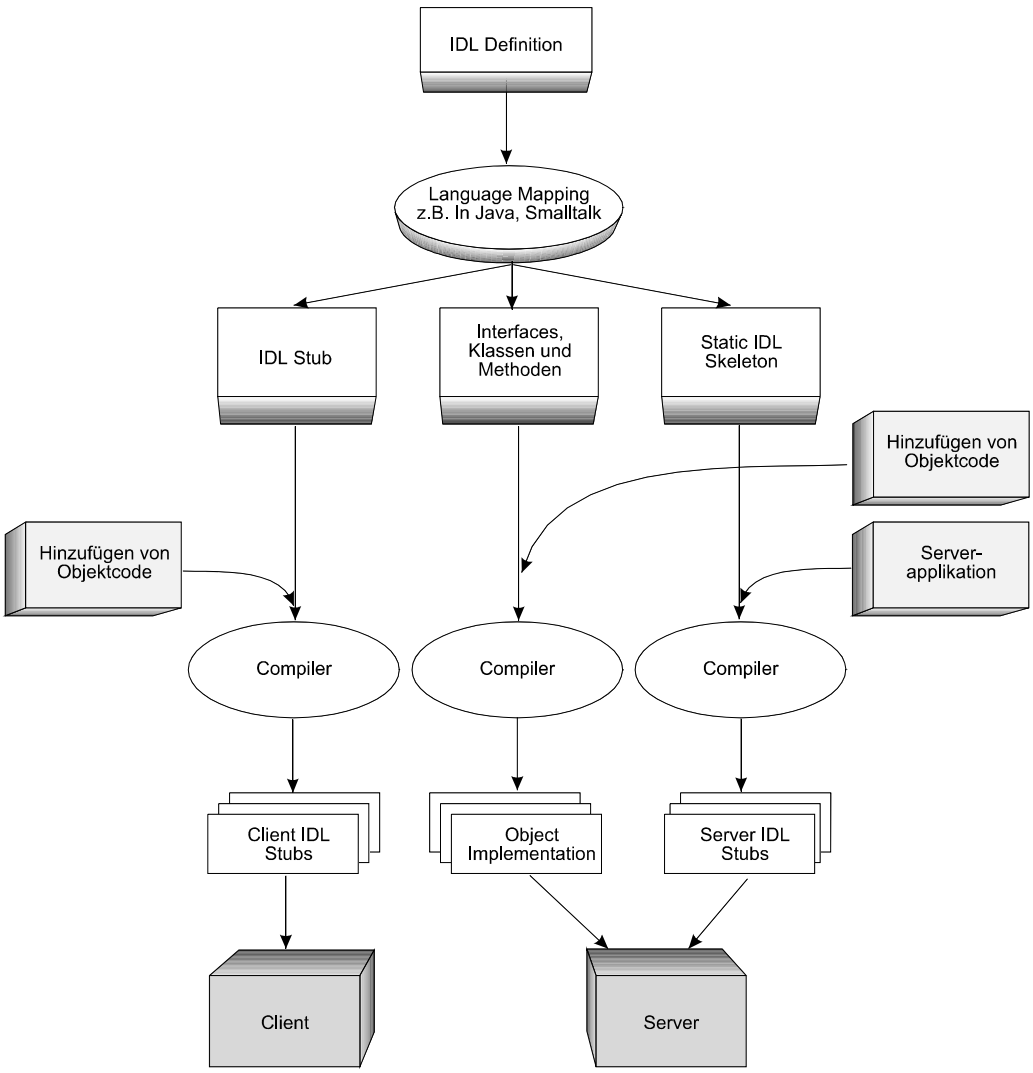


Abbildung 3-7: Generierung von Programmen

3.6 Statischer / Dynamischer Objektaufruf

Um einen Request vom Client zu einem Implementationsobjekt zu senden, stehen zwei Aufrufe zur Verfügung:

- statischer Objektaufruf
- dynamischer Objektaufruf

3.6.1 statischer Objektaufruf

Der statische Objektaufruf wird verwendet, wenn das Interface zum Kompilierzeitpunkt definiert ist, d.h. der IDL-Compiler verwendet werden kann, um einen Stub zu generieren.

- Verwendung von Client Stubs (Proxy)

- Verfügbare Methoden zur Implementierungszeit bekannt

- Hauptvorteile
  - typsicher
  - effizient
  - einfach zu bedienen

- Pendant auf der Server Seite ist das Skeleton

3.6.2 dynamischer Objektaufruf

Wenn zum Kompilierzeitpunkt die Schnittstelle nicht definiert wird, besteht die Möglichkeit, zur Laufzeit einen Request dynamisch zu generieren.

- Verwendung generischer Stubs
  - Schnittstelle(DII) zur Erzeugung der Anfrage zur Laufzeit
  - Nutzung von „Metadaten“ aus dem Interface Repository

- Flexibler als der statische Objektaufruf
  - Methoden müssen nicht zur Laufzeit bekannt sein
  - Nutzung bei Browsern, Gateways

- Erlaubt „Deferred Synchronous Invocation“
  d.h. asynchroner Request, bei dem das Ergebnis später geholt werden kann.

- komplex, weniger typsicher und effizient

- DSI für Objekte, die kein IDL-Skeleton besitzen
  - wird genutzt zur Interoperabilität zwischen ORBs

3.7 Object Adapter (OA)

Während der ORB als Kommunikationsbus benötigt wird, stellt ein **Object Adapter** die **Laufzeitumgebung** für die Objekt Implementationen zur Verfügung.

Dabei können für verschiedene Aufgaben, verschiedene OA bereitgestellt werden.

Im Wesentlichen umfaßt das **Aufgabengebiet** des Object Adapters folgende Punkte:

- **Registrierung** von Objekten
- Erzeugung und Interpretation von **Objektreferenzen**
- Aufruf der Methoden der **Objektimplementation**

- **Sicherheit** (Zugangsberechtigung eines Clients)
- **Aktivierung** und Deaktivierung von Objektimplementationen
  - Kontrolle der Serverprozesse
- zwei standardisierte OAs
  - Basic Objekt Adapter (für normale verteilte Anwendungen)
  - Object-Oriented Database Adapter (für Datenbanken)

Nachfolgendes Bild zeigt die Integration des Objekt Adapters innerhalb der OMA.

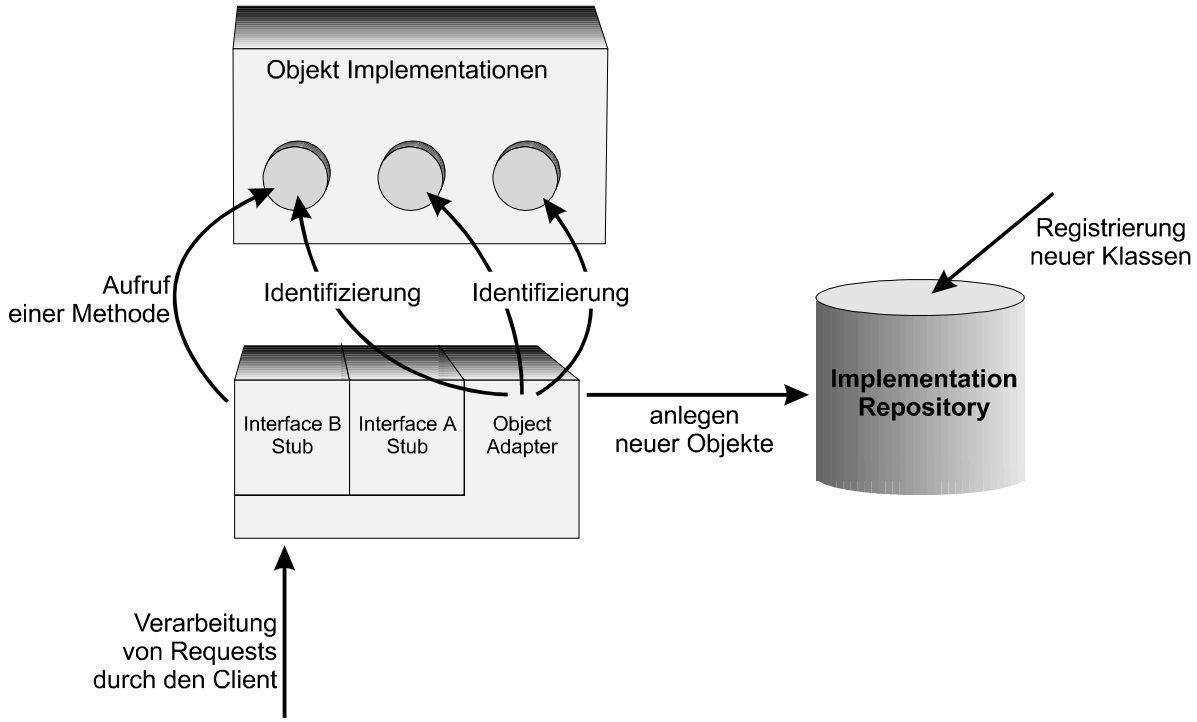


Abbildung 3-8: Einbindung des OA

3.8 Interoperabilität

Bei größeren Projekten/Systemen kann es vorkommen, daß verschiedene Firmen innerhalb einer Kooperation ein System entwickeln.

Im Rahmen dieser Kooperation erfolgt die **Kommunikation** zwischen den verschiedenen Teilen über den Object Request Broker, wobei die beteiligten Firmen nicht erfreut wären, wenn sie zu einem bestimmten Produkt gezwungen würden.

Aus diesem Grund ist es notwendig, daß eine Kommunikation zwischen ORBs möglich ist, damit der Client und Server unter **verschiedenen ORBs** entwickelt werden kann.

Nachfolgend einige Punkte zu der **Interoperabilität** zwischen verschiedenen ORBs:

- erst seit CORBA 2.0

Die OMG hat es in der Spezifikation des CORBA 1.0-Standards versäumt, die ORB übergreifende Kommunikation zu definieren. Dieses wurde mit der Verabschiedung des CORBA 2.0-Standards nachgeholt.

- zwei konkurrierende Ansätze
  - OSF-DCE-basiert (DP, DEC, Microsoft) - **abgelehnt**
  - UNO (IONA, SunSoft, IBM)

- General Inter-ORB Protocol (GIOP)
  Das **GIOP** definiert den Aufbau eines Inter-ORB-Protokolls. Eine Ausprägung dieses Protokolls über TCP/IP ist das **Internet Inter-ORB-Protokoll**, das im weiteren als IIOP bezeichnet wird. Die Beziehung zwischen GIOP und IIOP ist z.B. gleich der Beziehung zwischen SGML und HTML.

  - einheitliches Nachrichtenformat für alle Semantiken der Kommunikation über ORBs
  - einheitliche Typkodierung für IDL Datentypen (siehe XDF)

Für die Interoperabilität gibt es drei Ansätze:

- Interoperabilität über eine **Halbbrücke**
- Interoperabilität über eine **Vollbrücke**
- Interoperabilität über ein **gemeinsames Protokoll**

3.8.1 Interoperabilität über eine Halbbrücke

Hierbei wird das Protokoll, das ein ORB verwendet, über eine Halbbrücke in ein **allgemeines Protokoll** gewandelt, um dann ebenfalls über eine weitere Halbbrücke in das Protokoll des anderen ORBs umgesetzt zu werden, d.h. für die eigentliche Kommunikation wird ein allgemeines Protokoll verwendet. Das allgemeine Protkoll könnte zum Beispiel das TCP/IP sein.

Die Brücke stellt dabei ein Stück Soft- oder Hardware dar.

Nachfolgendes Diagramm zeigt diesen Aufbau:

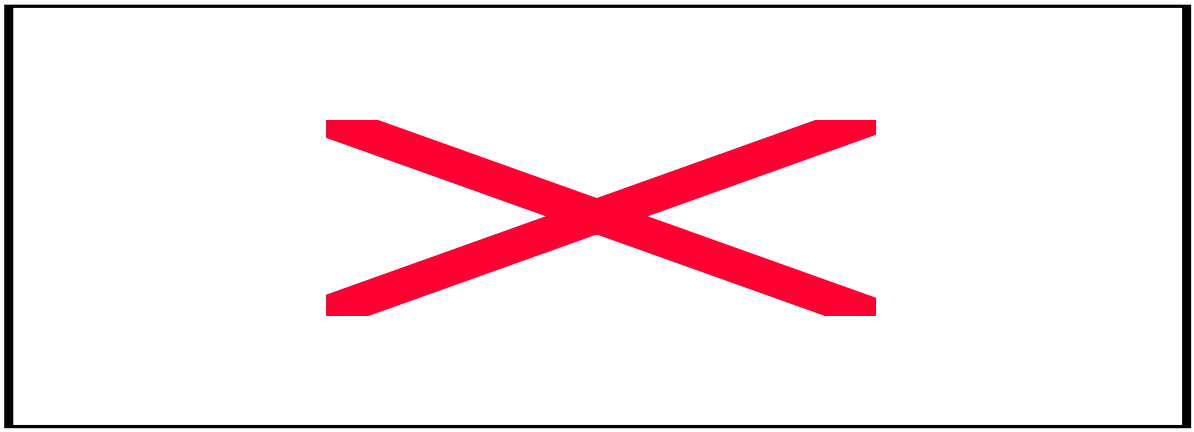


Abbildung 3-9: Interoperabilität über eine Halbrücke

3.8.2 Interoperabilität über eine Vollbrücke

Eine weitere Methode für die Interoperabilität zwischen zwei ORBs stellt die **Vollbrücke** dar. Diese Brücke wandelt das Protokoll des einen ORBs direkt in das Protokoll des anderen ORBs um.

Nachfolgendes Diagramm zeigt den Aufbau der Vollbrücke:

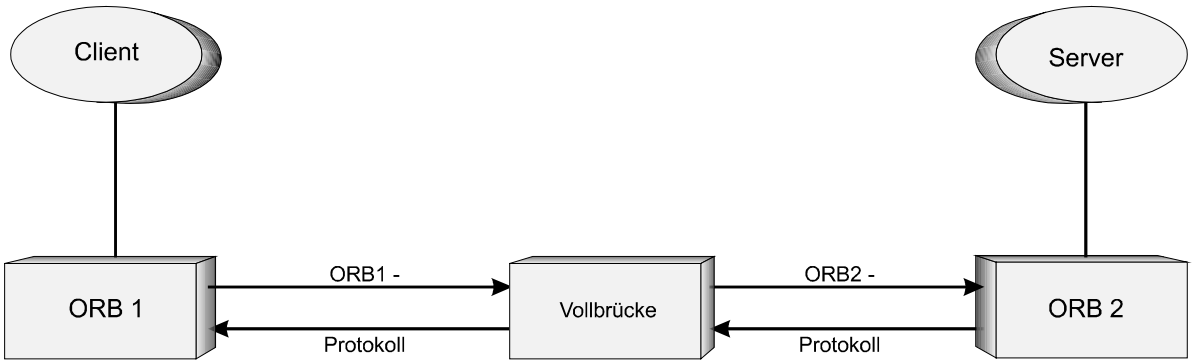


Abbildung 3-10: Interoperabilität über eine Vollbrücke

3.8.3 Interoperabilität über gemeinsames Protokoll

Als dritte Methode steht für die Interoperabilität zwischen den ORBs ein **gemeinsames Protokoll** zur Verfügung.

Dabei erfolgt die Kommunikation zwischen zwei verschiedenen ORBs über ein von beiden ORBs gemeinsam verwendetes Protokoll. Ein solches Protokoll stellt das Internet Inter-ORB Protokoll (IIOP) dar, welches eine Realisierung des GIOP für ein TCP/IP Netz ist.

Nachfolgendes Diagramm zeigt den Aufbau über das gemeinsame Protokoll:

Abbildung 3-11: Interoperabilität über gemeinsames Protokoll

4 IDL - Interface Definition Language

4.1 Zweck der IDL

Die IDL dient dem Ziel, innerhalb von CORBA den unterschiedlichen Softwareobjekten die Möglichkeit zu geben, betriebssystem-, hersteller- und programmiersprachenübergreifend zu kooperieren. Dazu wurde ganz bewußt die Spezifikations- von der Implementationsseite getrennt.

IDL-Compiler übersetzen die rein beschreibenden IDL-Definitionen in Prozedurrümpfe herkömmlicher Programmiersprachen, auch Stubs bzw. Skeletons genannt. Das heißt auch, daß Objekte nicht in der IDL implementiert oder aus einem IDL-Quelltext heraus aufgerufen werden können. Dafür müssen Programmiersprachen verwendet werden (z.B. Java, Smalltalk, C++...).

Wenn ein Client einen Dienst vom Server abfragt (Request), besteht dieser Request aus mehreren Informationen. Ein Request besteht dabei aus der Operation, dem angefragten Zielobjekt, einer Menge von Parametern und optional können noch Mengen von Ausnahmen (Exceptions) mitgeliefert werden.

Die IDL Interface Definition besteht aus einem Kopf und einem Rumpf. Im Kopf werden die Namen vergeben und eventuelle Vererbungs-hierarchien. Geerbte Definitionen können auch wieder überschrieben werden.

4.2 Struktur einer IDL

```
module <identifier>
{
<type declarations>;
<constant declarations>
<exception declarations>;

     interface <identifier> [:<inheritance>]
     {
          <type declarations>;
          <constant declarations>;
          <attribute declarations>;
          <exception declarations>;

          [<op type>] <identifier> (<parameter>)
          [raises exception] [context];
                    .
                    .
          [<op type>] <identifier> (<parameter>)
          [raises exception] [context];
     };

     interface <identifier> [:inheritance>]
          .
          .
}; // end of module
```

4.3 Module

Sie stellen einen separaten Namensbereich zur Verfügung. Ein Name XYZ aus dem Modul MODUL_NAME wird mit MODUL_NAME::XYZ identifiziert.
Ein Bezeichner, der in einem bestimmten Namensraum definiert wurde, kann nur dort verwendet werden.
Anm.: siehe auch Namensräume

4.4 Interfaces

Ein Interface hat selbst wieder einen Namen und kann eigene Exceptions und Attribute besitzen. Es enthält einen Menge von Operationen (Methoden). Interfaces können auch voneinander abgeleitet sein, d.h. Methoden können auch vererbt werden. Ähnlich einem Modul eröffnet eine Interface-Definition einen separaten Namensraum, so daß alle Bezeichner wieder neu vergeben werden können.

4.5 Vererbung

Ein Interface kann von beliebig vielen Interfaces erben.
Erbt ein Interface A von einem Interface B, dann sind automatisch alle in B enthaltenen Definitionen Bestandteil von A.

```
interface A : B {...};
```

Unzulässig ist, wenn ein und derselbe Bezeichner von mehreren Basisinterfaces geerbt wird. Stichwort: Konflikt der Namensräume.

Mehrfach-Vererbung ist ebenfalls möglich :

```
interface A : B,C { ... };
```

4.6 Attribute

Attribute sind Bestandteile von Interfaces. Sie sind äquivalent zu je einem Paar von Zugriffsoperationen; eine zum Setzen des Attributwerts und eine zum Abfragen des Attributwerts.

```
attribute short x;
```

Es können auch mehrere, durch Komma getrennte Namen angegeben werden, wodurch gleichzeitig mehrere Attribute vom gleichen Typ definiert werden.

```
attribute string x1,x2,x3;
```

Ein Attribut kann als nicht modifizierbar gekennzeichnet werden.
Dazu wird das Schlüsselwort *readonly* vorangestellt.

```
readonly attribute a1,a2,a3
```

Durch readonly wird verhindert, daß ein Client den Wert des Attributs direkt beeinflussen kann.

4.7 Operationen

Operations-Deklarationen sind ähnlich aufgebaut wie in C. Optional kann der Definition das Schlüsselwort *oneway* vorangestellt werden, um anzuzeigen, daß die Operation asynchron ausgeführt wird.

```
short op1 (inout short arg);
oneway void op2 (in long arg1, in char arg2);
short op3();
short op3(void);    // Falsch
```

Die Argumentenliste wird in runde Klammern eingeschlossen. Sie kann leer sein oder durch Kommas getrennte Argumente enthalten.
Innerhalb der Argumentenliste ist eine Verwendung der Schlüsselwörter `in, inout, out` möglich.

`in` bedeutet dabei: Der Wert des Arguments wird vom Client zum Server übertragen. Er darf vom Server nicht verändert werden.
`out` bedeutet: Der Server bekommt eine Kopie des Argumenten-Werts

übertragen, welche vom Server modifiziert an den Client zurückgeschickt wird.

`inout` bedeutet: Die Daten werden zunächst vom Client zum Server, anschließend vom Server zum Client übertragen.

Beachte: `oneway` Operationen dürfen nur `in` Parameter besitzen.

Exceptions (siehe 4.9 Exceptions) können im Zusammenhang mit Operationen optional angegeben werden. Die Definition wird mit dem Schlüsselwort `raises` eingeleitet. Ihm folgt in runden Klammern eine Aufzählung der Bezeichner der möglichen Exceptions.

```
// angenommen, die Exceptions X,Y,Z sind bereits
// definiert

raises (X,Y,Z)        // richtig
raises()              // falsch
raises(BAD_PARAM)     // falsch
```

4.8 Datentypen

Sie dienen zur Beschreibung der Werte für Parameter, Exceptions, Attribute und Rückgabeparameter.

Basic types: short, long, unsigned short, unsigned logn, float, double, char, boolean, octet

Constructed types: enum, string, struct, array, union, sequence, any

4.8.1 Constructed types

In der IDL sind die Basic types vordefiniert. Aus den Constructed types lassen sich eigene Datentypen bilden.

4.8.1.1 struct (Struktur)

Eine Struktur ist ein Datensatz, der mehrere Elemente zu einer Einheit zusammenfaßt. Die Elemente besitzen einen Typ und einen Namen.

```
 struct s1 {
      short element1;
      string element2;
};
struct s2 {
      s1 element1;
      any element2;
};
```

4.8.1.2 union

Eine Union ist eine Struktur, deren Elemente sich denselben Hauptspeicherplatz teilen müssen, so daß zu einem Zeitpunkt immer nur eines der Elemente aufbewahrt werden kann. Um zu wissen, um welches Element es sich handelt, wird in der IDL eine Union grundsätzlich mit einem Selektor gestartet. Der Selektor kann vom Typ char, boolean, short, long sein. Nimmt der Selektor ihren Wert an, dann bedeutet das, daß sich das dadurch markierte Element gerade in der Union befindet.

```
union UnionName switch(char) {
     case 1: short short_val;
     case 2: string string_val;
};
```

4.8.1.3 enum (Aufzählungen)

Eine Aufzählung “enum” besteht aus einer geordneten Liste von Bezeichnern. Sie beginnt mit dem Schlüsselwort enum, dem der Bezeichner für die Aufzählung und, eingeschlossen in Klammern, die Liste der Bezeichner folgt:

```
enum EnumName {ID1, ID2, ID3};
```

4.8.1.4 sequence (Sequenz)

Eine Sequenz ist eine Folge von Elementen desselben Typs. Ihre Länge kann entweder durch einen Maximalwert beschränkt werden oder nicht.

```
typedef sequence <<Typ> <Name>>
typedef sequence <<Typ>,<Länge> <Name>>
z.B.:
typedef sequence <long> LongSeq;
typedef sequence <string,10> StringSeq10;
```

4.8.1.5 array (Vektor)

Folgen von Elementen desselben Typs mit fester Länge werden als array bezeichnet. Sie können mehrdimensional sein.

```
typedef float array [10];
typedef float array [10][2];
```

4.9 Exceptions

Eine Exception ist ein Datentyp, dessen Wert dazu benutzt wird, dem Aufrufer einer Operation das Auftreten von Fehlern mitzuteilen. Wie Strukturen enthalten auch Exceptions Elemente. Über sie kann derjenige, der die Exception auslöst, zusätzlich Informationen über den aufgetretenen Fehler bereitstellen.

```
exception Beispiel {
     short info1;
     string info2;
};
```

Wie eine Exception ausgelöst bzw. abgefangen wird, ist eine Frage der jeweiligen Programmiersprache und wird nicht von der IDL beschrieben.

4.10 Schlüsselwörter

Folgende Schlüsselwörter sind reserviert und dürfen nicht als Bezeichner verwendet werden:

any, attribute, boolean, case, char, const, context, default, double, enum, exception, FALSE, float, in, inout, interface, long, module, Object, octet, oneway, out, raises, readonly, sequence, short, string, struct, switch, TRUE, typedef, unsigned, union, void

4.11 Namensräume

Namensräume können ineinander verschachtelt werden. In äußeren Namensräumen bekannte Bezeichner sind in den inneren Namensräumen automatisch verfügbar. Die lokale Definition überdeckt dann die äußere. Der überdeckte Bezeichner kann jedoch in der originalen Bedeutung verwendet werden, wenn er durch den Bezeichner des äußeren Namensraums und dem Scope-Operator :: qualifiziert wird.

Beispiel :

```
module X {
          const short i = 1;
          interface Y {
               const long i = X::i * 2;
               .....
          };
         };
```

Innerhalb der Definition des Interfaces Y überdeckt die lokale Definition der Konstanten i den gleichnamigen Bezeichner aus der Definition des Moduls X. Auf diesen kann innerhalb von Y jedoch noch über den qualifizierten Bezeichner X::i zugegriffen werden.

Soll auf einen global definierten Bezeichner zugegriffen werden, der lokal überdeckt wird, so ist ihm der Operator :: voranzustellen:

Beispiel :

```
typedef string T;
interface Y {
      void T(in short arg1);
      ::T op();
};
```

4.12 Beispiel einer IDL

```
module MyAnimals
{
      // Class definition of Dog
      interface Dog: Pet, Animal
      {
            attribute integer age;
            exception NotInterested (string explanation);

            void Bark(in short how_long)
                  raises (NotInterested);

            void Sit(in string where)
                  raises (NotInterested);

            void Growl(in string at_whom)
                  raises (NotInterested);
      };
      // class Definition of cat
      interface Cat: Animal
      {
            void Eat();
            void HereKitty();
      };
}; // end of MyAnimals
```

4.13 Verarbeitung der Interfaces

4.13.1 Interface Repository (IR)

Ein IR ist eine Online Datenbasis, in der IDL Definitionen eingetragen werden können. Die in einer IDL-Definition enthaltenen Informationen werden
im IR durch CORBA Objekte repräsentiert, so daß Clients und Server von entfernten Rechnern auf sie zugreifen können.

4.13.2 Funktion des IR

Man kann Objektinformationen mit in die Stub-Routinen hineinkompilieren,
dann sind sie fest oder man benutzt das IR, in dem diese Informationen
dynamisch abrufbar sind.

das IR prüft Types von Methodensignaturen
Der ORB kann z.B. anhand der im IR aufbewahrten Informationen überprüfen, ob ein vom Client gesetzter Request korrekt ist, oder

nicht.

das IR hilft den ORBs, Objekte über ORB Grenzen hinaus zu übersetzen und zu lokalisieren (dazu muß das Objekt aber in allen IR`s bekannt sein.)

4.14 Implementation Repository

Neben der reinen Definitionsseite (Interface Repository), gibt es noch die Implementierungsseite. CORBA benutzt den Objektadapter OA, um die Objektimplementierungen zu verwalten. Alle Implementierungsinformationen sind im Implementation Repository vermerkt:

Registrieren von Server Klassen mit dem Implementation Repository
Instanziierung neuer Objekte zur Laufzeit
Generieren und Verwalten von Objektreferenzen
Bekanntgabe seiner selbst und der Dienste, die er anbietet
Abarbeiten eingehender Methodenaufrufe des Clients
Umlenken der Methodenaufrufe zu entsprechenden Server-Objekten

Objektimplementierungen können dadurch aktiviert und auch deaktiviert werden, wobei die Methodenaufrufe dann durch Stubs erfolgen.

4.15 Statischer Aufruf

Hier wird die IDL in ihrer eigentlichen Form genutzt und beim Kompilieren in einen IDL Stub der gewünschten Programmiersprache gemapped. Dadurch kennt der Client zur Compilierzeit die Objektreferenz und kann damit über den ORB direkt die Objektimplementierung aufrufen.

4.16 Dynamischer Aufruf

Hat ein Client keine Objektreferenz, muß sie beim Request dynamisch generiert werden. Beim Dynamic Invocation Interface (DII) wird mittels einer Standard-Operations-Signatur die gewünschte Methode zur Laufzeit aufgerufen. Auch hier ist man unabhängig von der jeweiligen Objektimplementierung, da nur das Interface dynamisch angepaßt wird.

5 Java Mapping Grundlagen

Das Language Mapping ist eines der wichtigsten Konzepte in der CORBA-Spezifikation. Dies ermöglicht erst die **Unabhängigkeit** und **Flexibilität** von CORBA. Eine in der Interface Definition Language (IDL) beschriebene Schnittstelle (interface) kann daher programmiersprachenneutral formuliert werden. Über den sogenannten idl2xx-Compiler wird eine Zuordnung - der in IDL beschriebenen Module, Interfaces, Operationen und Attributen - in eine bestimmte Programmiersprache erreicht.

Im folgenden soll nun das Language Mapping von IDL nach Java dargestellt werden.

5.1 Module

CORBA: modules

Ein CORBA IDL `module` wird in ein Java `package` gemappt.

```
// CORBA-IDL
module MyModule
{
   ...
};

// generated Java Code
package MyModule;
  ...
```

5.2 Konstanten

CORBA: const (Konstanten) bei der Definition außerhalb eines Interfaces

CORBA-`const` (-Konstanten) werden zu `static final` Instanzen in Java gemappt. Der Name der IDL Konstanten wird zum Namen der Java Klasse. Innerhalb dieser Klasse gibt es die Instanz-Variable value, die den konstanten Wert enthält.

```
// CORBA-IDL
const long MyConst = - 12345;

// generated Java Code
public final class MyConst
{
   final public static int value = (int) -12345;
}

// Zugriff erfolgt über ...
int MyResult = MyConst.value;
```

CORBA: const (Konstanten) bei der Definition innerhalb eines Interfaces

CORBA-`const` (-Konstanten) die in einem IDL Interface deklariert werden, werden im zugehörigen Java Interface auf Attribute vom Typ `public final static` gemappt

```
// CORBA-IDL
module example
{
   interface Face
   {
      const long alongOne = -321;
   };
};

// generated Java Code
package example;
interface Face
{
   public static final int value = (int) -321;
}
```

5.3 Basisdatentypen

CORBA- Basistypen: short, ushort, long, ulong, long long (Ganzzahlen)

Die CORBA-IDL enthält 6 Integer (Ganzzahl) -Datentypen: `short` und `long`, `long long` jeweils mit und ohne Vorzeichen (`signed` und `unsigned`). Java kennt nur 3 Integer Datentypen: `short`, `int` und `long`.

Die CORBA-Typen werden in die entsprechenden Java-Typen gemappt.

```
// CORBA-IDL
const short MyShort = -1;
const unsigned short MyUnsignedShort = 15907;
const long MyLong = -12345;
const unsigned long MyUnsignedLong = 901008;
const long long MyLongLong = 1035541

// generated Java Code
final public class MyShort
{
   final public static short value = (short) (-1L);
}
final public class MyUnsignedShort
{
   final public static short value = (short) 15907;
}
final public class MyLong
{
   final public static int value = (int) -12345;
}
```

```
final public class MyUnsignedLong
{
   final public static int value = (int) 901008;
}
final public class MyLongLong
{
   final public static long value = (long) 1035541
```

CORBA- Basistypen: float, double (Gleitkommazahlen)

CORBA kennt 2 floating point-Formate: `float` und `double`. Diese werden in die gleichnamigen Java-Typen `float` und `double` gemappt.

```
// CORBA-IDL
const float MyFloat = 3.14159;
const double MyDouble = 2.71828182845904;

// generated Java Code
final public class MyFloat
{
   final public static float value = (float) (3.14159);
}
final public class MyDouble
{
   final public static float value = (double)
2.71828182845904;
}
```

CORBA-Basistyp: char

CORBA-`chars` werden durch das Language Mapping in den Java-Typ `char` gemappt.

```
// CORBA-IDL
const char MyChar = 'A';

// generated Java Code
public final class MyChar
{
   final public static char value =  (char) 'A';
}
```

CORBA-Basistyp: boolean

Die Ausdrücke für `boolean` sind in IDL und Java identisch. Die IDL-Konstanten TRUE und FALSE werden in die Java-Konstanten `true` und `false` gemappt.

```
// CORBA-IDL
const boolean MyBoolean = TRUE;

// generated Java Code
public final class MyBoolean
{
   final public static boolean value = true;
}
```

CORBA- Basistyp: octet

Der CORBA-Typ `octet` wird mittels Language Mapping in den Java-Typ `byte` gemappt.

```
// CORBA-IDL
void myOctet(in octet input);

// generated Java Code
public void myOctet(byte input)
```

5.4 Aufzählungstyp

CORBA-Typ: enum (Aufzählung)

Der CORBA-IDL-Typ enum wird in eine Java-Klasse des selben Namens vom Typ final public gemappt.In dieser Klasse wird eine wird eine value Methode, 2 `public static final` Attribute pro Element und ein Konstruktor mit dem Zugriffsmodifizierer `private` deklariert.

```
// CORBA-IDL
enum MyEnum { none, first, second, third, fourth };

// generated Java Code
final public class MyEnum
{
   public static final int _none = 0;
   public static final MyEnum none = MyEnum (_none);
   public static final int _first = 1;
   public static final MyEnum first = MyEnum (_first);
   public static final int _second = 2;
   public static final MyEnum second= MyEnum (_second);
   public static final int _third = 3;
   public static final MyEnum third= MyEnum (_third);
   public static final int _fourth = 4;
   public static final MyEnum fourth= MyEnum (_fourth);

   public int value();
   public static MyEnum from_int (int value);
   private MyEnum(int);
```

5.5 Strings

CORBA-Typ: string

Ein CORBA IDL-Typ `string` wird in den Java-Typ `java.lang.String` gemappt.

```
// CORBA-IDL
const string MyString = ‘Hello World’;

// generated Java Code
```

```
final public class MyString
{
   final public static String value = „Hello World“;
}
```

5.6 Strukturen

CORBA-Typ: struct

Der CORBA-IDL-Typ `struct` wird in ebenso in eine Klasse gemappt. Diese Klassen enthalten Instanzvariable für die Felder und einen Konstruktor zum initialisieren der Felder. Weiterhin wird ein Standardkonstruktor zur Verfügung gestellt, der die Variablen „default“-mäßig initialisiert.

```
// CORBA-IDL
struct MyStruct
{
   long myLong;
   string myString;
};

// generated Java Code
final public class MyStruct
{
   // Constructors
   public MyStruct(int _myLong, String _myString) { ... };
   public MyStruct(){ ... };

   // Data Members
   public int myLong;
   public String myString;
}
```

CORBA-Typ: union

Eine CORBA-IDL `union` wird in eine Java-Klasse gemappt, welche einen default-Konstruktor, eine Zugriffsmethode für den Diskriminator (Unterscheider) und Zugriffsmethoden für jedes Element bereitstellt. Die Zugriffsmethoden für die Elemente haben die gleichen Namen, jedoch unterschiedliche Signaturen.

```
// CORBA-IDL
union MyUnion switch (EnumType)
{
   case first:   long win;
   case second:  long place;
   case third:   long show;
   case fourth:
   default:      long other;
};

// generated Java Code
final public class MyUnion
```

```
{
   public MyUnion() { ... };
   public int discriminator() { ...};
   public int win() { ...};
   public void win(int discriminator, int _val) { ... };
   public int place() { ... };
   public void place(int discriminator, int _val) { ... };
   public int win() { ... };
   public void win(int discriminator, int _val) { ... };
   public int show() { ... };
   public void show(int discriminator, int _val) { ... };
   public int other() { ...};
   public void other(int discriminator, int _val) { ... };
}
```

5.7 Felder

CORBA-Typ: sequence

Eine CORBA `sequence` wird in ein Java `array` gemappt.

```
// CORBA-IDL
typedef sequence<MyObject> MySequence;
MySequence myStuff;

// generated Java Code
public MySequence[] myStuff;
```

CORBA-Typ: array

Ein CORBA `array` wird ebenso wie eine `sequence` nach Java gemappt.

5.8 Any-Typ

CORBA-Typ: Any

Der CORBA-IDL Typ `Any` wird in die Java-Klasse `CORBA.Any` gemappt. Der CORBA-Typ `Any` ist ein sehr mächtiger und nützlicher Typ. Man kann jede typisierte Information als `Any` darstellen. Die CORBA Type-Codes runden den Any-Typ ab. Man kann den Datentyp damit zur Laufzeit erfahren.

5.9 Interfaces

CORBA: interface

Der CORBA-IDL-Typ `interface` wird in ein Java `interface` gleichen Namens gemappt.

```
// CORBA-IDL
interface myInterface
{
   ...
};

// generated Java Code
public interface myInterface
{
   ...
};
```

5.10 Attribute

CORBA: attribute

In CORBA-Interfaces können `attribute` angegeben werden. Die Umsetzung in Java erfolgt auf die Art und Weise, daß Methoden gleichen Namens generiert werden, welche sich nur durch ihre Signatur unterscheiden. Diese Methoden dienen zum setzen und lesen (set und get) von den Attributen. Wenn ein Attribut als `readonly` markiert wird, generiert der idl2java-Compiler nur die get-Methode.

```
// CORBA-IDL
attribute long attribute1;
readonly attribute long attribute2;

// generated Java Code
public int attribute1();
public void attribute1(int value);
public int attribute2();
```

5.11 in, out und inout parametern

CORBA: Parameter-Modi

Die CORBA-IDL definiert 3 Modi die Parameter zu übergeben: `in`, `out`, `inout`. Da die Semantik für die `out` und `inout` Parameter pass-by-reference sind, jedoch Java für Parameter nur pass-by-value zur Verfügung stellt, sind zusätzliche Mechanismen notwendig

5.12 Exceptions

CORBA: exception

Der CORBA-Mechanismus erlaubt es eigene `exceptions` zu definieren und zu werfen. Die CORBA-IDL `exceptions` werden so ähnlich wie die Strukturen (`struct`) in die Zielsprache Java gemappt. Es wird eine Klasse mit Instanz-Variablen und Konstuktoren generiert. Es werden 2

Arten von CORBA-Exceptions unterschieden: `checked` und `unchecked exceptions`. Alle benutzerdefinierten Exceptions werden indirekt von der Klasse `java.lang.Exception` abgeleitet und sind `checked exceptions`. CORBA `system exceptions` sind indirekt von der Klasse `java.lang.RuntimeException` abgeleitet und sind `unchecked`.

```
// CORBA-IDL
module MyException
{
   exception ex1 { string reason; };
};

// generated Java-Code
package MyException;
final public class ex1 extends org.omg.CORBA.UserException
{
    public String reason;
    public ex1() { ... }
    public ex1(String reason) { ... }
}
```

5.13 Zusammenfassung

Als Zusammenfassung soll die Darstellung aller CORBA-to-Java Umsetzungen mittels einer Tabelle erfolgen.

CORBA-IDL	**Java**
module	package
interface	interface
boolean, TRUE, FALSE	boolean, true, false
char	char
octet	byte
string	java.lang.String
(unsigned) short	short
(unsigned) long	int
(unsigned) long long	long[3]
float	float
double	double
enum	Klasse mit static final enum
struct	Klasse mit Instanz-Variablen und Konstruktoren
union	Klasse mit get/set-Methoden für die Felder
sequence	array
array	array
any	CORBA.Any-Klasse
exception	Klasse mit Instanz-Variablen[4]

[3] Dies ist ein neuer CORBA-Typ, welcher von SUN einführt wurde.

typedef	Verwendung des original Namens[5]

[4] Diese Klasse wird stets von CORBA.UserException abgeleitet.
[5] Java unterstützt keine typedefs.

6 C++ Mapping Grundlagen

Die **Programmiersprache C++** ist Im Gegensatz zu C immer noch nicht standardisiert[6], so daß eine Standardisierung für das Mapping von IDL nach C++ mit Schwierigkeiten verbunden war. Das Ergebnis ist ein Mapping, das den Entwicklern von IDL Compilern im Zweifelsfall mehrere **Optionen** einräumt. Solche Optionen waren zum Beispiel für die folgenden Sprachelemente von C++ erforderlich:

Exception Handling: Nicht alle Compiler unterstützten zum Zeitpunkt der Standardisierung das **Exception-Handling**, wie es von Stroustrop definiert wurde. Aus diesem Grund definierte der OMG Standard eine Möglichkeit, Exceptions auch über einen Environment-Parameter übergeben zu können.

Namespace: **Namespacing**, obwohl in der C++ Standardisierung angedacht, wird praktisch noch von keinem C++ Compiler unterstützt.

Templates: Auch die Unterstützung von **Templates** war zur Zeit der Spezifizierung durch die OMG so unterschiedlich, daß in der CORBA Spezifizierung gänzlich auf Templates verzichtet wurde.

Das Language Mapping für C++, wie es im folgenden beschrieben werden soll, wurde im März 1996 als Version 1.1 veröffentlich. Da es sich um einen relativ neuen Standard handelt, ist davon auszugehen, daß die derzeit verfügbaren Produkte noch nicht alle im Standard festgeschriebenen Eigenschaften besitzen.

6.1 Module

In IDL ist es möglich über das Schlüsselwort `module` einen Namensraum zu definieren. Damit ist es möglich eigene Definitionen abzugrenzen, um so eventuelle **Namenskonflikte** zu vermeiden.

In C++ wird ein IDL `module` in ein `namespace` Konstrukt übersetzt.

```
// IDL
module M
{
   .....
};
```

```
// C++
namespace  M
{
   .....
};
```

Für C++ Compiler, die noch keinen `namespace` Konstrukt unterstützen, werden IDL Module in einfache C++ Klassen umgesetzt:

[6] Beziehungsweise war zum Zeitpunkt der Spezifizierung von CORBA noch nicht standardisiert.

```
// IDL
module M
{
   ....
};
```

```
// C++
class M
{
   ....
};
```

Der Zugriff auf Elemente eines Namensraums erfolgt in IDL und C++ gleichermaßen über qualifizierte Bezeichner:

```
// IDL
M::x
```

```
// C++
M::x
```

Von CORBA vordefinierte Bezeichner, z.B. für IDL Standardtypen (siehe Abschnitt 6.3), befinden sich im Adressraum **CORBA**.

6.2 Konstanten

IDL-Konstanten werden direkt in entsprechende C++ Konstanten überführt. In C++ ist es jedoch nicht möglich Konstanten innerhalb von Klassendeklarationen zu initialisieren. IDL Konstanten innerhalb einer `interface` - Deklaration werden deshalb in C++ als **Klassenattribute** umgesetzt, die erst in der Implementierungs-datei initialisiert werden:

```
// IDL
const string unit = “traffic_factory“;
interface TrafficFactory
{
   const short MaxNoOfClients = 10;
}
```

Aus der obigen IDL Definition würde ein IDL Compiler folgenden Code generieren[7]:

```
// C++ Header-Datei
static const char* const unit = “traffic_factory“;

class TrafficFactory
{
public:
static const CORBA::Short MaxNoOfClients;
}
```

In der „cc“ Datei muß nun die Konstante noch initialisiert werden:

```
// C++ Implementierungs-Datei
const CORBA::Short TrafficFactory::MaxNoOfClients = 10;
```

[7] Dieses C++ Code-Beispiel, wie auch alle folgenden, stellen nur Auszüge von dem tatsächlich generierten Code dar.

6.3 Basis Datentypen

Die Umsetzung von IDL Basistypen nach C++ wird in der folgenden Tabelle dargestellt. Da C++ Datentypen (im Gegensatz zu IDL Datentypen) nicht systemunabhängig sind, ist eine **direkte** Umsetzung von IDL Datentypen in C++ Typen (z.B. `int`, `long` oder `float`) nicht möglich.

Ein Ausweg bietet eine indirekte **Umsetzung** auf Bezeichner, denen je nach Zielplattform mittels `typedef` geeignete C++ Standardtypen zugewiesen werden. Damit wird garantiert, daß zum Beispiel der Datentyp `CORBA::ULong` unabhängig von der verwendeten Plattform immer ein 32 Bit Integer darstellt. Tabelle 6-1 zeigt die Umsetzung aller IDL Basistypen in die entsprechenden C++ Bezeichner:

IDL	C++
short	CORBA::Short
unsigned short	CORBA::Ushort
long	CORBA::Long
unsigned long	CORBA::ULong
float	CORBA::Float
double	CORBA::Double
char	CORBA::Char
boolean	CORBA::Boolean
octet	CORBA::Octet

Tabelle 6-1 Mapping von Basistypen

Mit Ausnahme der Basistypen `boolean`, `char` und `octet` sind die zugehörigen C++ Typen garantiert voneinander verschieden. Dies spielt dann eine Rolle, falls Methoden überladen werden sollen.

Für den Typ `CORBA::Boolean` sind im Standard nur die Werte „0“ (FALSE) und „1“ (TRUE) definiert. Die symbolischen Werte TRUE/FALSE müssen, falls gewünscht, vom Programmierer entsprechend selbst definiert werden.

6.4 Aufzählungstyp (enum)

Aufzählungen (`enum`) in IDL werden ohne Veränderung in C++ Aufzählungen umgesetzt:

```
// IDL
enum Color {RED, YELLOW, GREEN};

// C++
enum Color {RED, YELLOW, GREEN};
```

6.5 Strings

IDL - Zeichenketten (`string`) werden in C++ durch den Typ `char*` repräsentiert, unabhängig davon, ob eine maximale Länge festgelegt wurde oder nicht. Das heißt diese Information geht beim C++ Mapping verloren. Der Programmier ist selbst dafür verantwortlich ausreichend Speicher zur Verfügung zu stellen.
Damit der ORB die Verwaltung des Speichers selbst durchführen kann, dürfen für Variablen vom Datentyp `string`, nur die folgenden Funktionen zum Erzeugen, Duplizieren und Freigeben von Strings verwendet werden:

```
// C++
char* CORBA::string_alloc (CORBA::Ulong len)
char* CORBA::string_dup    (const char*);
char* CORBA::string_free  (char*);
```

Um die gängigsten Programmierfehler bei Verwendung von Zeichenketten in C++ zu vermeiden allokiert `string_alloc()` genau `len`+1 Bytes, so daß `len` Zeichen sowie das Endekennzeichen `'\0'` darin Platz finden. Weiterhin ist es auch möglich der Funktion `string_free()` einen `NULL` Zeiger zu übergeben.

6.6 Strukturierte Datentypen

Die IDL-Datentypen `struct`, `union` und `sequence` werden in C++ als Klassen repräsentiert, die bereits über einen Standard-Konstruktor, einen Kopier-Konstruktor einen Destruktor sowie einen Zuweisungsoperator verfügen.

Der **Standard-Konstruktor** initialisiert alle Elemente von Basistypen mit dem Wert Null; für Elemente zusammengesetzter Datentypen wird der entsprechende Standard-Konstruktor aufgerufen.

Der **Kopier-Konstruktor** erzeugt eine physikalische Kopie aller Elemente (deep copy). Dazu allokiert der Kopier-Konstruktor entsprechend Speicher und kopiert die Elementwerte. Für Elemente vom Datentyp `string` ruft der Kopier-Konstruktor die Funktion `string_alloc()` für Objekt-Referenzen deren Funktion A::`_duplicate()` auf, wobei `A` der Name der Interface-Klasse darstellt (siehe 6.9.1 Interfaces und Objekt-Referenzen).

Der Zuweisungsoperator garantiert, daß zunächst alle Elemente freigegeben werden und anschließend eine **physikalische** Kopie erzeugt wird.

Der Destruktor gibt alle Elemente frei. Für ein Element vom Typ `string` wird dabei die Funktion `CORBA::string_free(),` für Objekt-Referenzen die Funktion `CORBA::release()` aufgerufen.

6.6.1 Struct-Typen

Strukturen in IDL werden in gleichnamige C++ Strukturen übersetzt. Für Strukturen fester Länge ist dabei eine statische Initialisierung möglich:

```
// IDL
struct FixedStruct
{
float f;
short s;
};
```

```
// C++
struct FixedStruct
{
FixedStruct ();
FixedStruct (const FixedStruct&);
FixedStruct& operator = (const FixedStruct&);
~FixedStruct ();

CORBA::Float f;
CORBA::Short s;
};
```

Damit sind die folgenden Anweisungen möglich:

```
FixedStruct x;
FixedStruct y = {3.1414, 5};   // Statische Initialisierung
x.f = 123.456789;
x.s = 10;
..
y  =  x;                       // Zuweisungs-Operator
...
FixedStruct z = x;             // Copy-Konstruktor
```

Für Strukturen variabler Länge gelten dieselben Regeln, nur daß in diesem Fall keine statische Initialisierung möglich ist.

6.6.2 Union-Typen

Eine IDL-Union wird beim Mapping in eine C++ Klasse übersetzt, die für jede ihrer Elemente je eine Methode für den lesenden bzw. eine Methode für den schreibenden Zugriff definiert:

```
<typ> element (void);     // lesender Zugriff
void element (<typ>);     // schreibender Zugriff
```

Beim schreibenden Zugriff auf eine Union wird zuerst der vom bisherigen Element belegte Speicherplatz freigegeben bevor der neue Wert gesetzt wird. Anschließend wird der Selektor entsprechend neu gesetzt.

```
// IDL
struct CompoundId { short no; long id };

union Message switch (long)
{
case 1: long        id;
case 2: string      name;
case 3: CompoundId address;
```

```
};
// C++
CompoundId com_id = {10, 123}; // stat. Initialisierung
U u;                                  // noch nicht initialis.
u.id(4711);               // Zuweisung an Element id
cout << “Identification: “ << u.id() << endl;

char* name = “CORBA“;
u.name (name);      // Zuweisung an Element name
cout << “Identification:“ << u.name() << endl;
```

Der Stand des Selektor kann über die Methode `<SelektorTyp> _d() const` abgefragt werden.

6.6.3 Sequenz Typen

Der IDL Datentyp `sequence` wird in C++ als Klasse umgesetzt. Mit Hilfe des `[]` Operators kann wie bei einem Array auf die einzelnen Elemente der Sequenz zugegriffen werden. Im Unterschied zu Arrays muß die Länge bei einer Sequenz nicht fest sein. Die aktuelle Länge einer Sequenz kann über die Methode

```
CORBA::Ulong  length () const;
```

abgefragt werden und über die Methode

```
void  length (CORBA::ULong);
```

beliebig oft geändert werden. Eine Verkleinerung der Länge bedeutet, daß nicht mehr benötigter Speicherplatz automatisch freigegeben wird.

```
// IDL
typedef sequence <long> LongSeq;

// C++
LongSeq  long_seq;        // Standard Konstruktor
long_seq.length (10);     // Länge auf 10 setzen
long_seq[0] = 47;
long_seq[1] = 11;
...
long_seq[9] = 99;
long_seq.length (5);      // die letzten 5 Elemente freigeben
```

Für den Fall, daß bereits bei der Initialisierung die Größe der Sequenz festliegt stehen zwei weitere Konstruktoren zur Verfügung. Für eine Sequenz aus Elementen von einem beliebigen Typ `T` generiert der IDL Compiler den folgenden Code:

```
// IDL
typedef sequence <T> Tseq;
```

Aus dieser `typedef` Definition generiert der IDL Compiler nun den folgenden Code:

```
// C++
class TSeq  {
public:
   TSeq ();                    // Standard Konstruktor

   TSeq (CORBA::ULong maxlen);      // maximale Laenge

   TSeq (CORBA::ULong maxlen, // maximale Laenge
           CORBA::ULong length,    // Laenge bei Initial.
           T* data,                // Speicherbereich
           CORBA::Boolean release = 0);

   static T* allocbuf (CORBA::ULong len);
.......

};
```

Beim letzten Konstruktor wird neben der maximalen Länge der **Sequenz** auch die Anfangslänge beim Erzeugen der Sequenz sowie die Anfangswerte (als Zeiger auf AnyType) übergeben. Der letzte Parameter `release` gibt an, ob die Sequenz den als dritten Parameter übergebenen Speicherbereich auch modifizieren und löschen darf (TRUE). In diesem Fall muß der Speicherbereich mit der folgenden Funktion allokiert werden:

```
T* data = TSeq::allocbuf (length);
```

```
// C++
T* data = AnyTypeSeq::allocbuf (2); // T = short
data[0] = 123;
data[1] = 456;
TSeq seq (100, 2, data, 1 /*TRUE*/);

// KEIN delete auf data ausführen
// KEIN direkter Zugriff data erlaubt
```

Wurde der Speicherbereich z.B. als C++ Array angelegt, so muß dagegen der `release` Parameter auf FALSE gesetzt werden und der Programmierer ist für die Freigabe des belegten Speichers selbst verantwortlich.

```
static AnyType data2[2] = {123, 456};
AnyTypeSeq s2 (100, 2, data2, 0 /*FALSE*/);

// Aenderungen auf data2 bezogen auf die Sequenz s2 sind
// undefiniert.
```

Wird der release Parameter auf FALSE gesetzt, so ist etwas Sorgfalt von Seiten des Programmierers erforderlich, um Speicherlöcher zu vermeiden. Insbesondere bei der Übergabe von Sequenzen als Parameter sollte deshalb `release` auf TRUE gesetzt werden.

6.7 Arrays

IDL-Arrays werden in gleichlautende Definition von C++ Arrays übersetzt oder in einem Datentyp, der sich äquivalent zu einem C++ Array verhält. Die statische Initialisierung von Arrays in C++ muß in beiden Fällen möglich sein. Der Zugriff auf Elemente des Arrays erfolgt wie gewohnt mit dem `[]` **Operator**;

```
// IDL
typedef short ShortArray[3];

// C++
const ShortArray sa = {1, 2, 3};
cout << sa[0] << endl;
```

Ist der Basistyp eines Arrays entweder vom Typ `string` oder `Object`, so wird bei einer Wertzuweisung an ein Element des Arrays zuerst der alte Wert freigegeben (für String mit `CORBA::string_free()`, bei Objektreferencen mit `CORBA::release()`) und anschließend eine physikalische Kopie (deep copy) des neuen Wertes zugewiesen.

```
// IDL
typedef float FloatArray[10];
typedef string StringArray[10];

FloatArray float_array;
StringArray string_array;

// Initialisieren von beiden Arrays

string_array[0] = CORBA::string_dup (“abc“);
float_array[0] =  2.718281828459;
```

.....

```
// Wertzuweisungen
// alter Wert wird überschrieben
float_array[0] = 3.14159265359;

// alter Wert wird zuerst freigegeben anschließend wird
// eine Kopie des neuen Wertes erzeugt
string_array[0] = CORBA::string_dup (“xyz“);
```

Neben dem Array vom Datentyp `T` erzeugt der IDL-Compiler einen weiteren Datentyp mit der Bezeichnung `T_slice`. Der Datentyp `T_slice` entspricht dem Datentyp der Elemente des Arrays. Bei einem `short`-Array entspricht `T_slice` also dem Datentyp `short`.

Die nachfolgenden Operationen dienen dem dynamischen Erzeugen, Duplizieren und Freigeben von Arrays des Datentyps `T`:

```
T_slice*      T_alloc ();                 // T erzeugen
T_slice*      T_dup (const T_slice*)      // T duplizieren
```

```
void          T_free (T_slice*);          // T freigeben
```

6.8 Any und T_var Datentypen

Das C++ Mapping generiert für jeden Datentyp T - mit Ausnahme der Basisdatentypen - einen weiteren Datentyp T_var. Die Besonderheit bei Verwendung von T_var Datentypen besteht darin, daß der von Variablen dieses Typs belegte **Speicherplatz** automatisch wieder **freigegeben** wird, sobald diese Variablen ihren Gültigkeitsbereich verlassen oder ein neuer Wert zugewiesen wird.

Dies ist vor allem bei Datentypen variabler Länge und vor allem bei Verwendung als out bzw. inout Parameter in Operationen vorteilhaft. Nach außen hin verhalten sich T_var Datentypen wie Zeiger auf den Datentyp T. Dies bedeutet, daß zum Beispiel bei Strukturen die Zeigerschreibweise (->) anstelle der Punktschreibweise (.) verwendet werden muß.

```
// IDL
struct S
{
   string     name;
   float      age;
};
```

.....

```
// C++
S a;
S_var b;
   .....
a = b;         // deep copy !
cout << „Names: „ << a.name  << „ and „ << b->name;
cout << endl;
}  // memory for b is freed!!
```

6.8.1 Der any Datentyp

Der IDL Datentyp any kann, wie der Name schon besagt, jeden Datentyp annehmen. In C++ wird ein solcher Datentyp in eine Struktur umgewandelt, die sich vereinfacht folgendermaßen darstellt:

```
// C++
struct any
{
   type_code*       _type;
   void*      _value;
   any();
   any (const any&);
   any (const char* type, const void* val);
   ~any (=;
   any& operator= (const any&);
   // stream like interface to fill and extract
```

```
};
```

Der Datentyp `any` wird vor allem bei Verwendung des **DII** *(*dynamic invocation interfaces) eingesetzt, wenn es darum geht während der Laufzeit CORBA-Anfragen zu erstellen und an das Zielobjekt zu senden.

6.9 Mapping von Interfaces

Ein Interface in IDL wird beim C++ Mapping in eine C++ Klasse umgesetzt mit allen Datentypen, Konstanten, Operationen und Exceptions wie sie auch innerhalb des IDL-Interfaces definiert wurden. Alle Definitionen innerhalb der C++ Klasse sind dabei als `public` deklariert.

Bei C++ Klassen, die aus IDL generiert werden, sind die folgenden Einschränkungen zu beachten:

- Es ist nicht möglich, direkt eine Instanz dieser Interfaceklassen zu erzeugen oder eine Referenz auf eine Interfaceklasse zu besitzen.
- Es ist nicht möglich, von einer Interfaceklasse abzuleiten. Statt dessen muß bereits in der IDL Beschreibung abgeleitet werden.
- Es ist nicht erlaubt, einen Zeiger (`A*`) oder eine Referenz (`A&`) auf eine Interfaceklasse zu besitzen. Statt dessen sind die zwei Datentypen `A_var` und `A_ptr` zu verwenden (siehe 6.9.1.).

Die Absicht, die hinter diesen Einschränkungen für den CORBA Anwender steckt, ist, den Entwicklern von CORBA Produkten einen großzügigen Freiraum bei der Implementierung des C++ Mappings zu geben Daraus ergeben sich zum Teil erhebliche Unterschiede für das C++ Mapping zwischen den verschiedenen Anbietern von Object Request Brokern, um die sich der Anwendungsprogrammierer jedoch nicht zu kümmern braucht.

6.9.1 Interfaces und Objekt-Referenzen

CORBA Objekte werden durch Objekt-Referenzen weltweit eindeutig identifiziert. Der Aufbau dieser Objekt-Referenzen bleibt dem Programmierer jedoch in der Regel verborgen, nur der ORB arbeitet intern mit ihnen. Der Programmierer benutzt statt dessen Instanzen von C++ Klassen (Stubs), die der IDL Compiler automatisch aus dem IDL-Code generiert.

Um den Zusammenhang zwischen IDL-Interface, den Stub-Klassen und den Objekt-Referenzen zu verdeutlichen betrachten wir die IDL-Definition von einem Interface `A`:

```
// IDL
interface B;                    // Vorwaertsdeklaration

interface A
{
   void getB (in string name, out B);
}
```

Aus dieser Definition generiert der IDL-Compiler eine gleichnamige C++ Interface- oder **Stub-Klasse**. Methodenaufrufe an diese Interface-Klasse werden vom ORB an die **Implementierungsklasse** weitergeleitet.

Wie die IDL-Definition zeigt, ist es auch möglich, einen Interface-Datentyp als Parameter einer Methode zu übergeben. Beim Aufruf der Methode wird jedoch nicht das CORBA Objekt selbst, sondern ein **Referenz** auf dieses Objekt übergeben. Empfängt der ORB der Client-Applikation eine Objekt-Referenz als Parameter, so wird für sie automatisch eine Interface-Klasse (in diesem Fall eine Instanz der Klasse B) generiert, über die dann auf wieder auf die Implementierungs-Klasse zugegriffen werden kann.

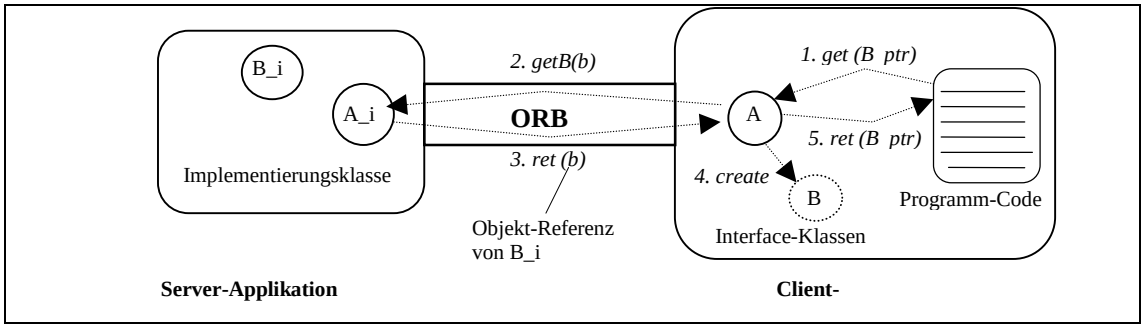


Abbildung 6-1 Interface-Klassen und Objekt-Referenzen

Objekt-Referenzen können auch für den Anwendungsprogrammierer interessant werden, falls zum Beispiel eine Referenz persistent gespeichert werden soll. Dazu wird die Objekt-Referenz mit Hilfe der Methode `object_to_string()` zunächst in eine **Zeichenkette** gewandelt, die daraufhin z.B. in einer Datei gespeichert werden kann. Die Methode `object_to_string()` wandelt die Zeichenkette wieder in ein Objekt-Referenz zurück, was bedeutet daß wiederum automatisch eine Instanz der entsprechenden Interface-Klasse erzeugt wird.

Sehr häufig werden Objekt-Referenz und Interface-Klasse begrifflich gleichgestellt werden. Dabei müssen uns jedoch einige Besonderheiten bewußt sein. Nehmen wir an eine Client-Applikation empfängt **mehrfach** dieselbe Objekt-Referenz. In diesem Fall kann es sein, daß der ORB für jede Objekt-Referenz eine eigene Interface-Klasse erzeugt. Die verschiedenen **Interface-Klassen** repräsentieren zwar dasselbe CORBA Objekt, sind aber aus C++ Sicht unterschiedliche Klassen.

Um zu vermeiden, daß für dieselbe Objekt-Referenz mehrfach Instanzen der entsprechenden Interface-Klasse erzeugt werden, besitzt jede Interface-Klasse einen Referenz-Zähler. Dieses Verfahren nennt sich **reference counting**, d.h. es wird von der Interface-Instanz mitgezählt, wieviele Verweise es innerhalb eines Programms gerade auf sie gibt. Erreicht diese Zahl den Wert 0, so wird die Instanz automatisch freigegeben.

Über die beiden Methoden

```
A* A::_duplicate (A* obj);
void CORBA::release (Object* obj);
```

kann dieser Zähler inkrementiert, beziehungsweise dekrementiert werden.

Zusätzlich zu der Interface-Klasse `A` generiert der IDL Compiler noch die beiden Typen `A_ptr` und `A_var`. Nur über diese beiden Typen darf der Programmierer Interface-Klassen referenzieren. Diese Festlegung gewährt den ORB Herstellern einen großen Spielraum in der Art und Weise, wie Objekt-Referenzen **repräsentiert** werden.
Variablen vom Typ `A_ptr` und `A_var` dürfen überall dort eingesetzt werden, wo Objekt-Referenzen vom Typ `A*` erwartet werden. Der Unterschied zwischen beiden Datentypen besteht in der Art der Speicherverwaltung. Wie auch bei den anderen Datentypen beinhaltet der `A_var` Datentyp eine **automatische Speicherverwaltung**.

6.10 Mapping von Attributen

Neben Operationen ist es in IDL ebenfalls möglich, **Attribute** innerhalb eines Interfaces zu definieren. Bei der Umsetzung in C++ werden pro Attribut eine **get-** und eine **set** -Methode mit dem Namen des Attributs erzeugt. Die Unterscheidung beider Methode erfolgt über die Signatur der Methode:

- Die set Methode erwartet einen Parameter vom Typ des entsprechenden IDL Attributs und gibt keinen Wert zurück.
- Die get Methode gibt einen Wert vom Datentyp des Attributs zurück und erwartet keinen Parameter.

```
// IDL definitions
interface Person
{
   attribute string first_name;
}
```

IDL nach C++ Compiler würden obigen IDL Code in etwa folgendermaßen umsetzen:

```
// C++ generated code
class Person
{
   void  first_name (char*);
   char*      first_name ();
}
```

Setzt man das Schlüsselwort **readonly** vor dem `attribute` Schlüsselwort, so wird lediglich die **get** Methode generiert.

6.11 Mapping von Operationen

Operationen, die innerhalb eines IDL Interfaces definiert wurden, werden in C++ Methoden mit entsprechendem Rückgabewert und Parametern umgewandelt.

Aus der folgenden IDL Definition

```
// IDL definition
interface Bank
{
```

```
   float withdraw ( in long pin, in float amount );
};
```

wird zum Beispiel der folgende Code generiert:

```
// C++
class Bank
{
   float withdraw (CORBA::Long pin, CORBA::Float amount );
};
```

Dabei ist zu beachten, daß der obige Code nur das Wesentliche zeigen soll und von dem tatsächlichen generierten Code je nach CORBA Anbieter etwas abweichen kann.

6.12 Mapping von in, out und inout Parametern

Das Mapping von Parametern und Rückgabewerte bei Methoden hängt im wesentlichen von den verwendeteten Datentypen ab. Nachfolgend soll für alle zuvor besprochenen Datentypen das Mapping anhand von Beispielen dargestellt werden.

6.12.1 Basistypen

Für alle Basistypen werden `in`-Parameter als Kopie, sowie `inout` bzw `out`-Parameter als Referenz übergeben:

```
// IDL
T operation (in T a1, inout T a2, out T a3);

// C++ generiert
T operation (T a1, T& a2, T& a3);
```

6.12.2 Zeichenketten

Zeichenketten werden in C++ in den Datentyp `char*` umgewandelt, wobei die Länge der Zeichenkette, wie in C üblich, durch das Endezeichen `'\0'` bestimmt wird. Dies erfordert, daß der Programmierer sowohl auf Client, wie auch auf Server-Seite stets genügend **Speicherplatz** zur Verfügung stellen muß, um die gewünschte **Zeichenkette** aufzunehmen. Das Mapping für Zeichenketten als Parameter zeigt das folgende Beispiel:

```
// IDL
string operation (in string a1, inout string a2, out string
a3);

// C++
char* operation (const char* a1, char*& a2, char*& a3);
```

Um Speicherfehler (insbesondere Speicherlöcher) zu vermeiden, sind bei der Parameterübergabe von Zeichenketten folgenden Regeln zu beachten:

Für in bzw. inout Parameter muß auf Client-Seite genügend Speicher zur Verfügung gestellt werden. Die Übergabe eines nicht initialisierten char* Parameters wird vom ORB mit einem Fehler beantwortet. Das **Allokieren** von Speicher muß mit den Funktionen string_alloc() beziehungsweise string_dup() geschehen.
Für Parameter vom Typ inout oder out allokiert der ORB selbständig Speicher für die Rückgabewerte. Der Client ist dann dafür verantwortlich diesen Speicher über die Methode CORBA::string_free() wieder freizugeben. Out Parameter sollten grundsätzlich mit dem Wert NULL initialisiert werden, da vom out-Parameter bereits belegter Speicher nicht automatisch freigegeben wird. Für Rückgabewerte gilt dasselbe wie für out Parameter.

```
{
   String_var str_inout = CORBA::string_dup (“abc“);
   String_var str_out = NULL;
   char* result;

   result = operation (“Do not bug me“, str_inout, str_out);

   cout << “Inout Paramater: “ << str_inout;
   cout << “Out Parameter: “   << str_out;

   // Speicher fuer result muss explizit freigegeben werden
   CORBA::string_free (result);

}  // Speicher fuer str_inout und str_out wird automatisch
   // freigegeben
```

Im obigen Beispiel werden die beiden inout/out Parameter als Variablen vom Typ String_var deklarariert. Damit wird der von beiden Variablen belegte Speicher automatisch beim Verlassen des Blocks freigegeben, während der Rückgabewert explizit freigegeben werden muß um ein Speicherloch zu vermeiden.

6.12.3 Strukturierte Typen als Parameter

Werden strukturierte Typen (Strukturen, Unions oder Sequenzen) als Parameter einer Operation übergeben, so generiert der IDL Compiler die Datentypen const T&, T*& bzw. T*, je nachdem, ob es sich um einen in-, out- oder inout-Parameter handelt und ob die Parameter eine feste oder variable Größe besitzen:

```
// IDL
struct T { long l; };          // Struktur fester Größe
struct V { string s };         // Struktur variabler Größe

T operation_1 (in T t1, inout T t2, out T t3);
V operation_2 (in V v1, inout V v2, out V v3);

// C ++ generierter Code
T operation_1 (const T& t1, T& t2, T& t3);
V* operation_2 (const V& v1, V& v2, V*& v3);
```

Im obigen Beispiel wurden Strukturen verwendet. Es gilt analog für Sequenzen und Unions, wobei für Sequenzen nur die Variante mit variabler Größe besteht.

Bei der Verwendung von strukturierten Datentypen als Parameter bietet sich der T_var Typ an. Dieser Typ besitzt Operatoren zur automatischen Anpassung an die von den Parametern geforderten Typen (type cast), so daß sich der Programmierer nicht mehr um solche Details kümmern muß.

In dem folgenden Beispiel können somit alle drei Parameter als T_var Typen deklariert werden:

```
// C++ Beispiel

T_var t1, t2, t3;   // Definiere Parameter als T_var Typen
T result;           // Definiere Rueckgabewert vom Typ T

t1 = new T;              // Erzeuge Struktur
t1->l = 123;t2 = t1;     // Zuweisung einer Kopie von t1
t3 = new T;              // Erzeuge neue Struktur fuer out

result = operation_1 (f1, t2, t3);

cout << t2,->l << t3->l << result.l;     // Ausgabe der
neuen Werte
```

Für strukturierten Datentypen fester Größe muß Speicherplatz für alle Parameter von Operationen vom Client-Programmierer zur Verfügung gestellt werden. Werden dabei T_var Typen verwendet, muß der Programmierer mittels dem new Operator Speicher allokieren.

Die Verwendung von strukturierten Datentypen variabler Länge unterscheidet sich in der Behandlung des out-Parameters sowie des Rückgabewertes. In beiden Fällen muß Client-Programmierer keinen Speicher allokieren. Dieser wird **automatisch** vom ORB beim Eintreffen der Werte im Client angelegt und muß im Anschluß vom Programmierer wieder **freigegeben** werden.

6.12.4 Arrays als Parameter

Ist T ein Array-Datentyp so werden Funktionsparameter für T in C++ in eine Form des Datentyps T_slice umgesetzt. Die exakte Form hängt dabei sowohl vom Parametertyp als auch davon ab, ob Elemente des Arrays eine fest oder variable Größe besitzen:

```
// IDL

//Array mit Elementen fester Groesse
typedef long   T[10];

//Array mit Elementen variabler Groesse
typedef string V[10];
```

```
T operation_1 (in T t1, inout T t2, out T t3);
V operation_2 (in V v1, inout V v2, out V v3);


// C++ generierter Code
typedef long        T_slice;   // Datentyp eines Elementes
typedef stringV_slice;   // Datentyp eines Elementes


T_slice* operation_1 (const T t1, T t2, T t3);
V_slice* operation_2 (const V v1, V v2, V_slice*& v3);
```

Und hier ein kurzes Beispiel für den Umgang von Arrays als Parameter von Operationen:

```
// C++ Code
T_var t1, t2, t3, result;
t1 = T_alloc ();         // Speicher fuer t1 allokieren
t1[0] = 123; t[1] = 456; ....  // Array t1 initialisieren
t2 = t1;            // Kopie von t1 an t2 zuweisen
t3 = T_alloc();     //  Speicher  allokieren  fuer  den  out-
Parameter

result =  operation_1 (t1, t2, t3);

cout << t1[0] << t2[0] << result[0];  // Ausgabe der Werte
```

Der Speicherplatz für `in` als auch `out`-Parameter (**!**) `t1-t3` muß vom Client-Programmierer zur Verfügung gestellt werden, während der Speicherplatz für den Rückgabewert automatisch vom ORB allokiert wird.
Man beachte zudem, daß bei der Zuweisung t2=t1 der alte Wert von t2 freigegeben wird (hier hatte t2 noch keinen Wert) und anschließend eine Kopie von t1 an t2 zugewiesen wird.

Für Arrays mit Elementen variabler Größe wird der Speicher für die Aufnahme von `out`-Parametern automatisch vom ORB allokiert und an die als Referenz übergebene Variable zugewiesen. Ansonsten kann das obige Beispiel auch für Arrays variabler Länge herangezogen werden.

6.12.5 Objekt-Referenzen als Parameter

Objekt-Referenzen werden als Parameter von Operationen wie Basistypen behandelt. Im nachfolgenden Beispiel steht der Bezeichner A für ein beliebiges zuvor definiertes Interface:

```
// IDL
A operation (in A a1, inout A a2, out A a3);

// C++
A_ptr operation (A_ptr a1, A_ptr& a2, A_ptr & a3);
```

Nachfolgend ein Beispiel für die Verwendung von Objekt-Referenzen als Parameter von Operationen:

```
// C++
A_var a1, a2, a3, result;
a1 = .. ;
a2 = A::_duplicate (a1);
a3 = A::_nil();

result = operation (a1, a2, a3);
```

6.13 Mapping von Exceptions

Die Behandlung von Ausnahmen (exceptions) ist in IDL ebenfalls vorgesehen. In C++ werden Exceptions als eigene Klassen codiert. CORBA unterscheidet dabei zwei Arten von Ausnahmen: **systembedingte** Ausnahmen (SystemException) und vom Anwender **definierte** Ausnahmen (UserException). Beide Klassen leiten sich dabei von der gemeinsamen Basisklasse **Exception** ab, so daß man die folgende Hierachie erhält:

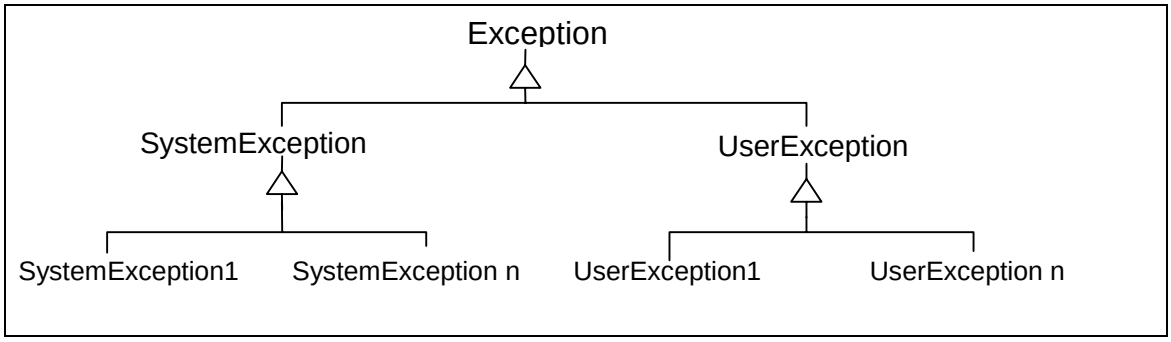


Abbildung 6-2: Exception Hierachie

Aufgrund dieser Hierachie ist es möglich alle Ausnahmen in einer einzigen catch-Anweisung abzufangen.

```
// C++ code
try {
.......
}
catch (Exception exception& ex) {
   .....
}
```

Auf ähnliche Art und Weise kann der C++ Programmierer alle systembedingten bzw. vom Anwender definierte Ausnahmen getrennt abfangen und bearbeiten.

7 CORBAservices und CORBAfacilities

Mit der Spezifikation von CORBA wurde das Fundament gelegt, aber CORBA verbindet lediglich Objekte, nicht aber Applikationen. Die Grundlage für die Integration von Applikationen wurden von der **OMG** in der Object Management Architecture (OMA) festgelegt.

Die OMA ist in zwei Hauptkomponenten unterteilt, den low-level **CORBAservices** und den intermediate-level **CORBAfacilities**, die im nachfolgenden besprochen werden sollen.

7.1 CORBAservices

CORBAservices bieten grundlegende Dienste für CORBA Objekte und stellen somit eine **Weiterführung** des reinen ORB dar. CORBAservices sind äußerst generisch. Zum einen sind sie völlig unabhängig von den jeweiligen Applikationen und zum anderen wurden sie nach dem Bauhaus Prinzip entwickelt. Das bedeutet, jeder Dienst besitzt nur eine einzige, dafür aber klar definierte **Aufgabe**. Weiterhin muß es möglich sein, diese Dienste auf unterschiedliche Art und Weise in den eigenen Applikationen zu integrieren.

Bei der Spezifikation der CORBAservices wurden - wie auch beim ORB - nur die **Schnittstellen** standardisiert nicht aber wie diese Dienste konkret implementiert werden sollen. Die OMG hat bis heute Standards für die folgenden Dienste veröffentlich:

Der **Naming Service** erlaubt es Komponenten im Netzwerk anhand ihres Namens zu finden und wird deshalb oft als Telefonbuch Dienst bezeichnet.

Der **Life Cycle Service** definiert Operationen, um Komponenten im Netz zu erzeugen, zu löschen, zu kopieren oder zu verschieben.

Der **Persistence Service** stellt ein einziges Interface zur Verfügung, um Komponenten persistent in Objektdatenbanken, relationalen Datenbanken oder flachen Dateistrukturen zu speichern.

Der **Event Service** erlaubt es, daß sich Komponenten für bestimmte Ereignisse registrieren lassen können. Tritt ein solches Ereignis ein, so werden alle Komponenten, die sich dafür registriert haben, entsprechend informiert.

Der **Concurrency Control Service** stellt einen Lock-Manager für Transaktionen oder Threads zur Verfügung.

Der **Transaction Service** stellt ein 2-phase-commit Protokoll für Komponenten zur Verfügung.

Der **Relationship Service** ermöglicht es, dynamisch Beziehungen zwischen Komponenten aufzubauen, ohne daß diese Komponenten einander bekannt sind.

Der **Externalization Service** stellt eine standardisierte Möglichkeit zur Verfügung, um Daten einer Komponente über einen `stream`-ähnlichen Mechanismus zu lesen oder zu schreiben.

Der **Query Service** stellt Abfrageoperationen für Objekte zur Verfügung, die ein Superset von SQL darstellen.

Der **Licensing Service** stellt eine Möglichkeit dar, die Inanspruchnahme von Komponenten zu kontrollieren.

Der **Properties Service** erlaubt es, bestimmte Eigenschaften mit beliebigen Komponenten zu verbinden.

Der **Time Service** stellt ein Interface zur Verfügung, das es erlaubt, verteilte Komponenten zeitlich zu synchronisieren.

Der **Security Service** stellt ein komplettes Rahmenwerk für Sicherheitsaspekte verteilter Komponenten zur Verfügung.

Der **Trader Service** repräsentiert die Gelben Seiten im Netzwerk. Er erlaubt es Objekten, für sich und ihre Dienste zu „werben“.

Der **Collection Service** bietet CORBA Interfaces, um die gängigsten „Collections“ für Komponenten zu erzeugen und zu manipulieren.

Eine weiterführende Beschreibung aller CORBAservices würde bei weitem den Rahmen dieses Manuskriptes sprengen. Daher sei an dieser Stelle auf die Literatur sowie der Home Page der OMG (www.omg.org) verwiesen. Im folgenden werden die wichtigen Dienste **Naming Service** und **Event Service** etwas näher beschrieben.

7.1.1 Der Naming Service

7.1.1.1 Begriffe

Der **Naming Service** stellt einen Basisdienst für Objekte dar, um andere Objekte im Netzwerk zu lokalisieren. Der **Naming Service** assoziiert Namen mit Objekt-Referenzen. Eine solche Namen-zu-Objekt Assoziation wird als **Namensbindung** (engl. name binding) bezeichnet. Ein Namenskontext (engl. naming context) ist ein Namensraum, in dem der Name eines Objektes eindeutig ist. Es ist möglich mehr als einen Namen einer Objekt-Referenz zuzuordnen. Namen werden immer relativ zu ihrem Kontext definiert.

Der Naming Service wurde im September 1993 zum OMG Standard. Ziel dieses Standards ist es, bestehende **Namens**- und **Verzeichnisdienste**, wie zum Beispiel DCE CDS, ISO X.500 oder Sun NIS+, unter einer einheitlichen Schnittstelle zu integrieren. Der Name, mit dem ein CORBA Objekt referenziert werden kann, setzt sich dabei aus einer Sequenz von Namen zusammen, die einen hierarchischen **Namensbaum** bilden, siehe Abbildung 7-1.

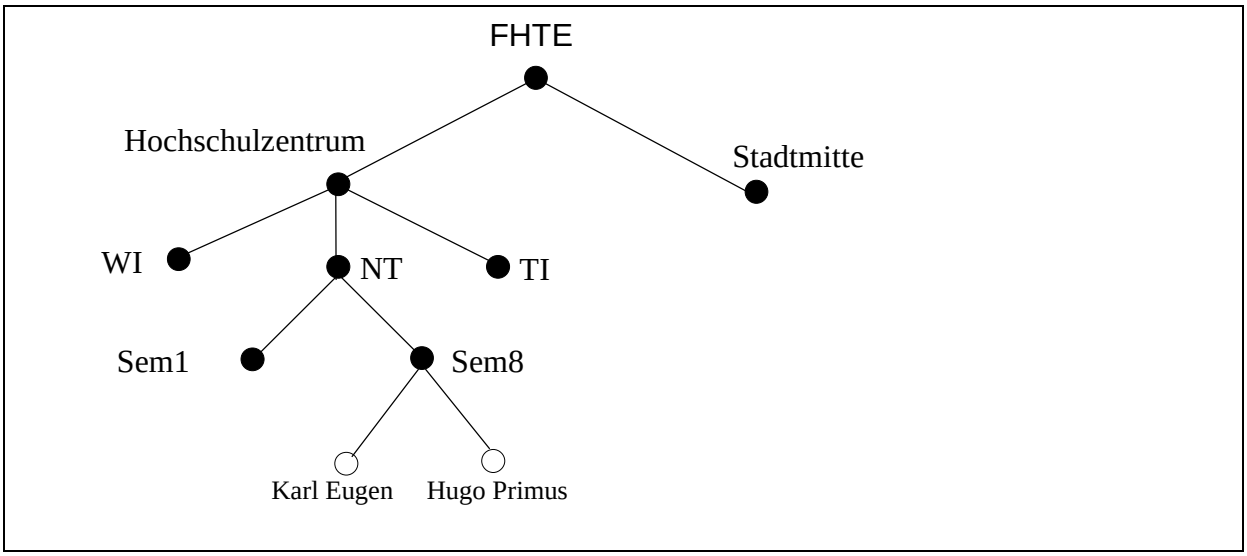


Abbildung 7-1: Namenshierachie

Jeder ausgefüllte Knoten in Abbildung 7-1 stellt einen Namenskontext dar, jeder nicht ausgefüllte Knoten den **einfachen Namen** (simple name) eines Objektes. Den vollständigen Namen eines Objektes erhält man durch Aneinanderreihen der Kontextnamen beim Durchlaufen der Hierarchie. Der vollständige (compound name) des Objektes „Karl Eugen“ lautet somit:

```
FHTE - Hochschulzentrum - NT - Sem8 - Karl Eugen
```

Über diesen Namen kann die dazugehörige Objekt-Referenz des Karl Eugen aufgelöst (engl. resolve) werden. Der umgekehrte Vorgang, also das Erzeugen einer Namen-zu-Objekt Assoziation wird auch als **Binden** (engl. bind) bezeichnet.

Jeder Knoten in der Namenshierachie ist eine Struktur, bestehend aus zwei Attributen:

1. **identifier** ist der Name des Knoten und
2. **kind** ist eine Zeichenkette, um den Knoten näher zu beschreiben

Die Informationen des **kind** Attributs werden dabei vom Naming Service in keiner Weise interpretiert oder manipuliert, sondern dienen lediglich dem Benutzer des Naming Service.

7.1.1.2 Die Interfaces

Im Naming Services Standard wurden zwei Interfaces definiert: **NamingContext** und **BindingIterator**. Instanzen vom Typ NamingContext enthalten einen Satz von Namensbindungen, in welchen jeder Name eindeutig ist. Diese Instanzen können auch ihrerseits an einen Namen gebunden sein und in anderen Namenskontexten erscheinen, um so eine Namenshierarchie zu bilden. Abbildung 7-2 zeigt die Definition beider Interfaces.

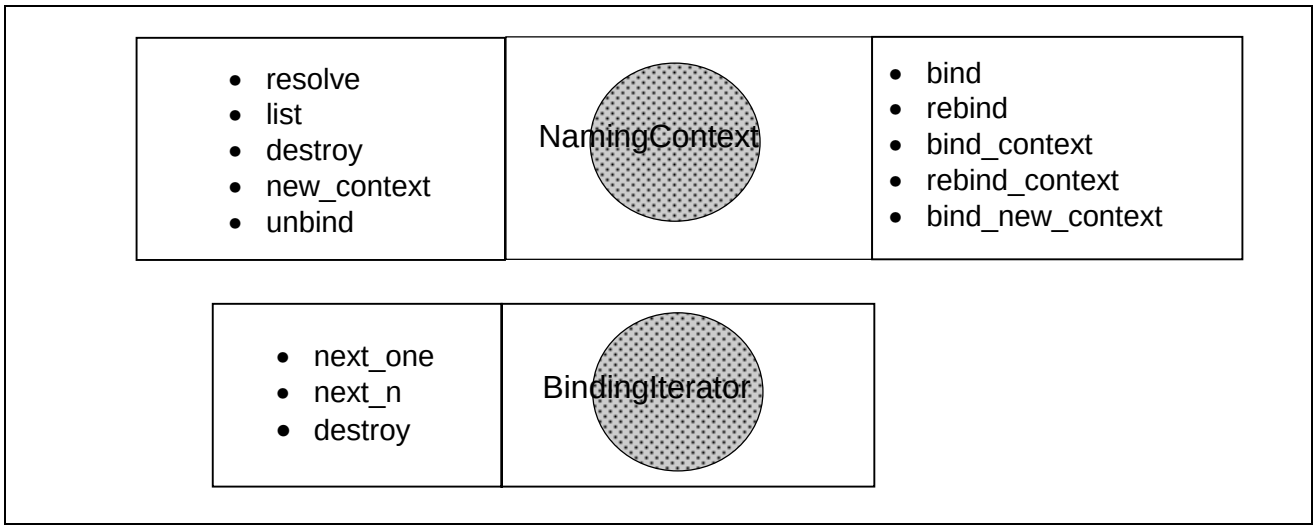


Abbildung 7-2: Interfaces des Naming Services

Die `bind()` Methode innerhalb des **NamingContext** Interfaces ermöglicht es, den Namen eines Objektes mit einem **binding context** zu verknüpfen. Die Methode `rebind()` hat denselben Effekt mit dem Unterschied, daß kein Fehler zurückgegeben wird, falls der Name bereits an einem anderen Objekt gebunden ist; das Objekt wird stattdessen einfach an den neuen Namen gebunden. Auf diese Art und Weise kann ein Namensbaum generiert werden.

Mit der Methode `unbind()` kann ein Name von einem **naming context** entfernt werden. Die `new_context()` Methode liefert den Namenskontext zurück. Die Methode `bind_new_context()` erzeugt einen neuen Kontext und bindet diesen an einen Namen, der als Parameter übergeben werden muß. Die `destroy()` Methode erlaubt es schließlich, einen Namenskontext zu löschen.

Objekte, denen zuvor ein Name zugewiesen wurde, können über die `resolve()` Methode aufgespürt werden. Mit Hilfe der `list()` Methode ist es möglich, über eine Menge von Namen zu iterieren. Die Methode liefert eine Instanz von `BindingIterator` zurück. Über die Methoden `next_one()` und `next_n()` kann über die Liste der Namen iteriert werden. Mit der Methode `destroy()` wird der Iterator wieder freigegeben.

7.1.2 CORBA Event Service

Ein normaler CORBA-Request ist eine synchrone Ausführung einer Operation an einem Objekt. Der Event Service entkoppelt die **Kommunikation** zwischen Objekten. Der Event Service stellt ein **Supplier-Consumer-Kommunikationsmodell** zur Verfügung, welches mehreren Supplier-Objekten erlaubt, Daten asynchron - mit Hilfe eines Event-Channels - an mehrere Consumer-Objekte zu senden. Dieses Supplier-Consumer-Kommunikationsmodell erlaubt einem Objekt einen wichtigen Zustandswechsel, beispielsweise daß eine Festplatte voll ist, jedem Objekt zu melden, welches an so einem Event Interesse hat.

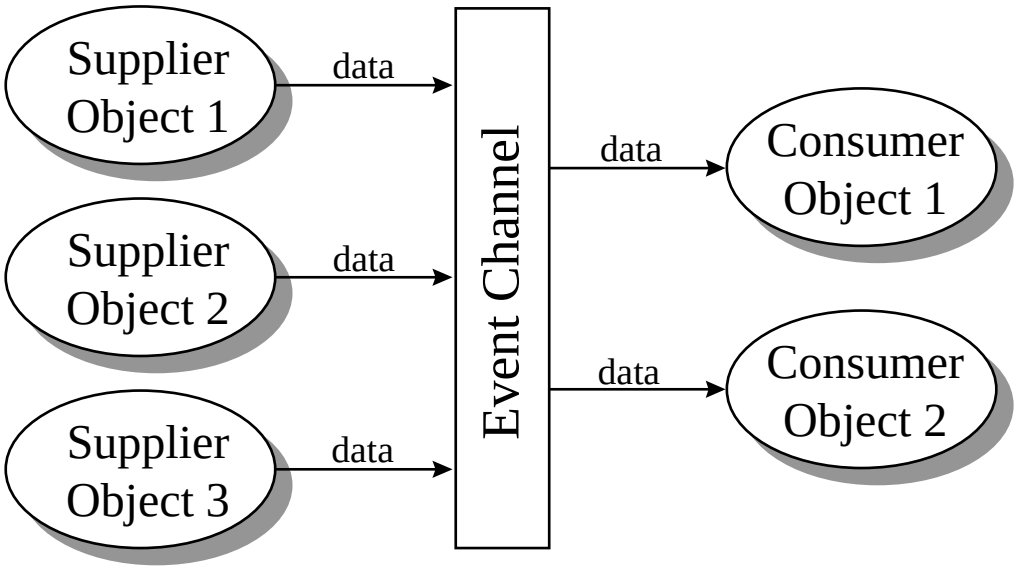


Abbildung 7-3: Supplier-Consumer-Kommunikationsmodell

Die Abbildung 7-3 zeigt drei **Supplier-Objekte** die über einen **Event-Channel** mit zwei **Consumer-Objekten** kommunizieren.

Der **Datenfluß** in den Event-Channel wird von den Supplier-Objekten gesteuert, während der Datenfluß aus dem Event-Channel von den Consumer-Objekten gesteuert wird.

Wenn jeder Supplier eine Nachricht pro Sekunde versendet, dann erhält jeder Consumer drei Nachrichten pro Sekunde und der Event-Channel leitet insgesamt 6 Nachrichten pro Sekunde weiter.

Der Event-Channel ist beides, ein Consumer und ein Supplier von Events.
Supplier und Consumer kommunizieren durch den Event-Channel mittles standard CORBA-Requests.

7.1.2.1 Proxy Consumer und Proxy Supplier

Consumer und Supplier sind komplett durch **Proxy-Objekte** (Stellvertreter Objekte) voneinander entkoppelt. Anstatt direkt miteinander zu interagieren, erhalten sie ein Proxy-Objekt vom Event-Channel und kommunizieren mit diesem Stellvertreter-Objekt. Das Supplier-Objekt erhält ein Consumer-Proxy-Objekt und das Consumer-Objekt erhält ein Supplier-Proxy-Objekt. Der Event-Channel lenkt den **Datentransfer** zwischen den Proxy-Objekten. Die Abbildung 7-4 zeigt, wie ein Supplier-Objekt über einen Event-Channel Daten an drei Consumer-Objekte verteilen kann.

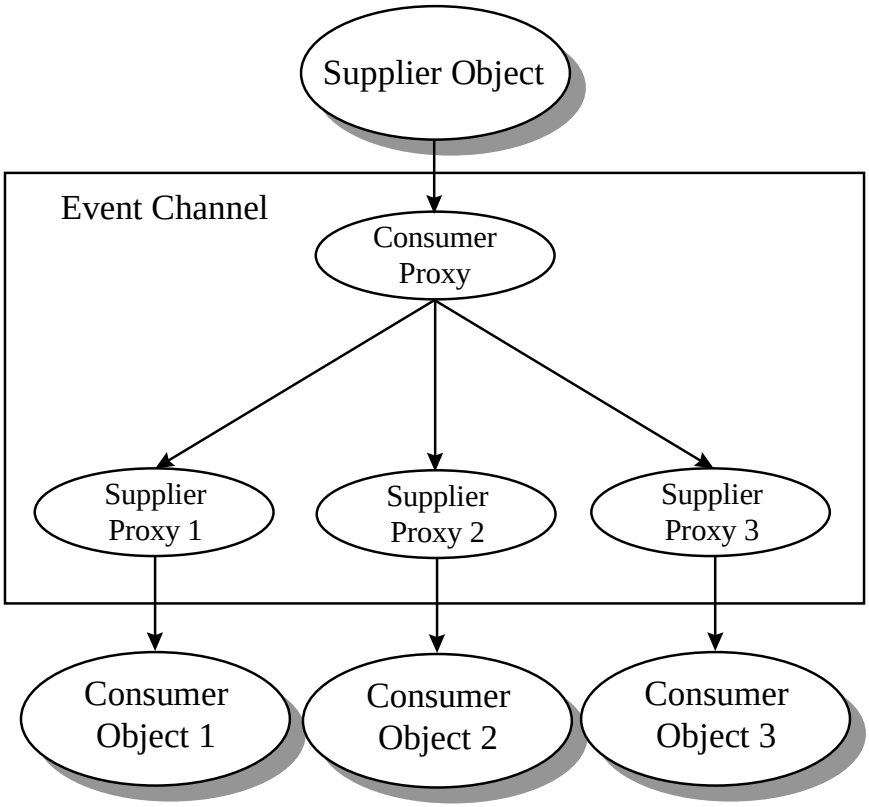


Abbildung 7-4: Consumer und Supplier Proxy Objekte

7.1.2.2 Kommunikations-Modelle

Der Event-Service ermöglicht beides - ein Pull- und ein Push-Modell für Supplier und Consumer-Objekte.
Im **Push-Modell** kontrollieren die Supplier-Objekte den Datenfluß, indem sie Daten zum Consumer „pushen“ (drücken).
Dagegen kontrollieren die Consumer-Objekte den Datenfluß im **Pull-Modell**, indem sie die Daten vom Supplier „pullen“ (ziehen).
Der Event Channel erwirkt dabei, daß es für die Supplier- und Consumer-Objekte unwichtig ist, welches Modell die anderen Objekte verwenden. Das bedeutet, daß ein Pull-Supplier einem Push-Consumer, wie auch ein Push-Supplier einem Pull-Consumer Daten übergeben kann.

7.1.2.2.1 Push-Modell

Das Push-Modell ist das häufiger verwendete der beiden Modelle.
Ein Beispiel des Push-Modells ist ein Supplier, der ständig den freien Speicherplatz auf einer Festplatte kontrolliert und **Meldung** an alle interessierten **Consumer** gibt, sobald dieser freie Platz eine Mindestgrenze unterschreitet.

Der Push-Supplier sendet daher Daten an sein ProxyPushConsumer-Objekt, falls dieser Zustand eintritt.

Der Push-Consumer läuft den größten Teil seiner Zeit in einer **Ereignisschleife** und wartet auf eintrefende Ereignisse von seinem ProxyPushSupplier.

Der Event Channel steuert den Datentransfer zwischen dem ProxyPushSupplier und dem ProxyPushConsumer.

Die Abbildung 7-5 zeigt einen Push-Supplier wie auch sein zugehöriges ProxyPushConsumer-Objekt. Weiterhin sind auch 3 Push-Consumer mit ihren zugehörigen ProxyPushSupplier dargestellt.

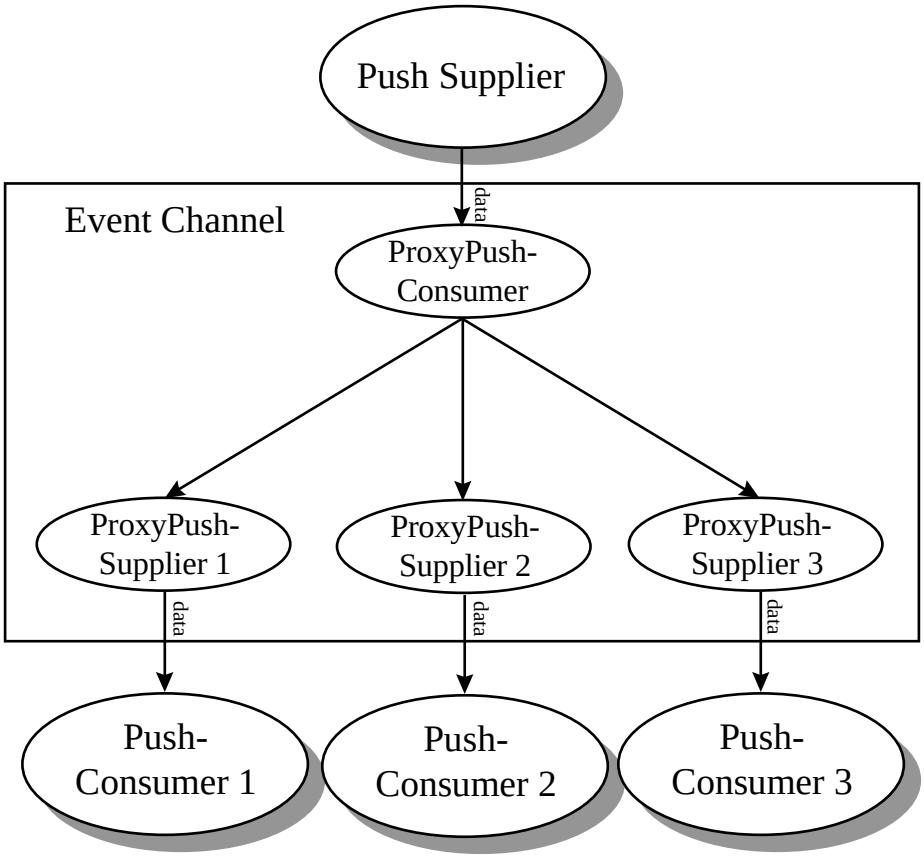


Abbildung 7-5: Push-Modell

7.1.2.2.2 Pull-Modell

Im Pull-Modell, „pullt“ der Event Channel regelmäßig das Supplier-Objekt, legt die Daten in eine **Warteschlange** (engl. queue) und macht diese Daten daher verfübar, um von einem Consumer-Objekt gepullt zu werden.

Ein Beispiel für einen Pull-Consumer ist ein Netzwerk-Monitor, der in regelmäßigen zeitlichen Abständen verschiedene Netzwerkgeräte (Bridge, Router) pollt, um Netzwerkstatistiken aufzustellen.

Der **Pull-Supplier** läuft die meiste Zeit in einer Ereignisschleife und wartet auf Datenanforderungen (data requests) von seinem **ProxyPullConsumer**.

Der Pull-Consumer fordert Daten von seinem **ProxyPullSupplier** an, wenn er bereit ist, (weitere) Daten aufzunehmen.
Der Event Channel pullt die Daten vom Supplier in eine Warteschlange und macht sie daher dem ProxyPullSupplier verfügbar.

Die Abbildung 7-6 zeigt einen Pull-Supplier und sein zugehöriges ProxyPullConsumer-Objekt. Weiterhin sind auch 3 Pull-Consumer mit ihren zugehörigen ProxyPullSupplier-Objekten dargestellt.

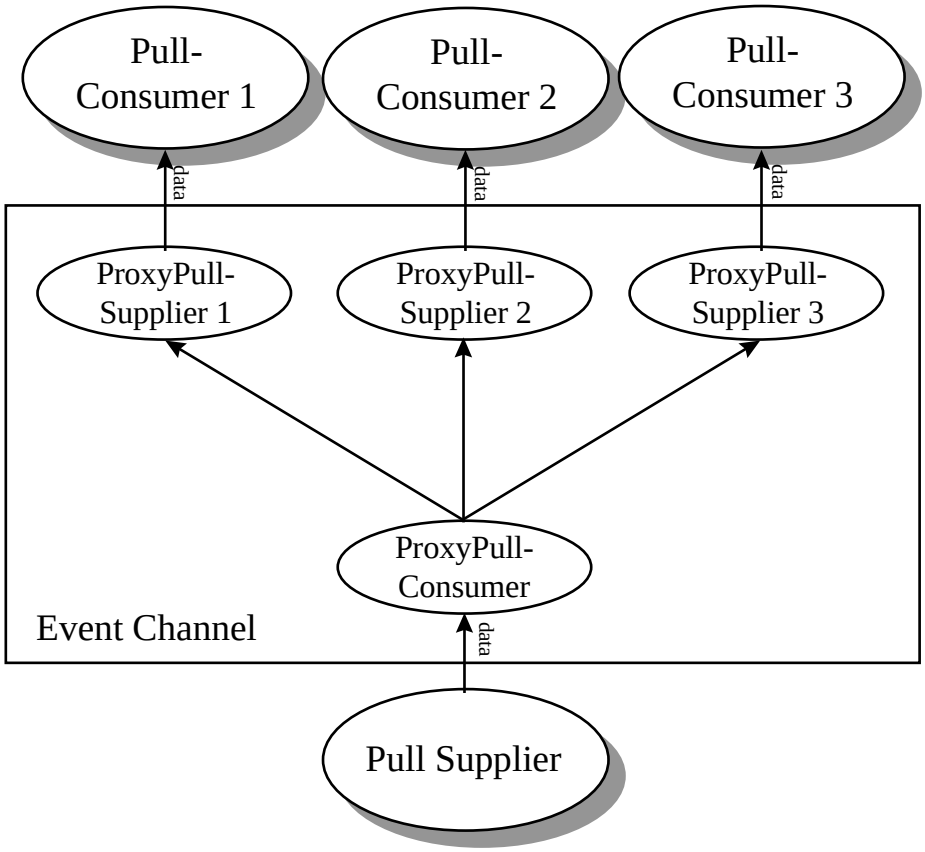


Abbildung 7-6: Pull-Modell

7.2 CORBAfacilities

7.2.1 Einleitung

Wie oben schon beschrieben, widmet sich die Object Management Group (OMG) der **Bereitstellung** und **Spezifizierung** von kommerziell verfügbaren **objektorientierten** Applikationen. Dabei bildet die Object Management Architecture die Grundlage für OMG Spezifikationen. Das Modell für diese Architektur ist hierbei das Referenz-Modell, welches die Komponenten, die Schnittstellen und die Protokolle bildet. Das Referenz Modell besteht aus den folgenden Teilen:

- **Object Request Broker**
  Der ORB bietet einen Software-Bus, über den verschiedene Objekte, die im Netz verteilt sind, miteinander kommunizieren können.

- **Object Services**
  Die Object Services bestehen aus einer Sammlung von Diensten (bestehend aus Schnittstellen und Objekten), die verschiedene **Grundfunktionalitäten** zur Verfügung stellen. Zu den Object Services gehören nur Dienste, die unabhängig von Applikationen sind und von ihrer Funktionalität nahe am ORB stehen. Hierzu gehört z.B. der Naming-Service, mit dem Objekte im Netz mit Hilfe einer Identifikation gefunden werden können.

- **Common Facilities**
  Die Common Facilities bestehen aus einer Sammlung von Diensten, die Funktionalitäten zur Verfügung stellen, die nicht von **grundsätzlicher Natur** sind, d.h. die Common Facilities stellen Dienste zur Verfügung, die für verschiedene Applikationen benötigt werden. Zu den Common Facilities gehört z.B. ein Dienst für den Dokumentenaustausch zwischen verschiedenen Applikationen.

- **Application Objects**
  Application Objects sind alle Objekte, die für Systeme oder **kommerzielle Produkte** entwickelt werden. Die Application Objects werden dementsprechend nicht von der OMG verwaltet.
  Es kann jedoch vorkommen, daß ein Application Object zu einem Common Facility wird, wenn seine Anwendung nicht auf ein spezielles Produkt zugeschnitten ist.

7.2.2 Die Rolle der Common Facilities innerhalb der OMA

Wie oben beschrieben, besteht das Referenzmodell der OMG aus den Teilen Object Request Broker, Object Services und Common Facilities.

Die Common Facilities bilden hierbei den Abschluß der Definition der Object Management Architecture, d.h. sie bilden den **höchsten Level** der OMA und sind deshalb eine konzeptuelle Verbindung zwischen den Object Services und den Application Objects.

Um die verschiedenen Dienste innerhalb der Common Facilities definieren zu können, wurden die Common Facilities in zwei Kategorien aufgeteilt:

- Horizontale Common Facilities
- Vertikale Common Facilities

7.2.2.1 Die horizontalen Common Facilities

In dieser Kategorie werden alle Dienste zusammengefaßt, die von grundsätzlicher Natur sind und von vielen oder den meisten Systemen benötigt werden.

Da relativ viele Dienste zu dieser Kategorie gehören, war es notwendig, vier verschiedene Gruppen zu bilden:

- User Interface
- Information Management
- System Management
- Task Management

7.2.2.2 Die vertikalen Common Facilities

In dieser Kategorie werden verschiedene Dienste zusammengefaßt, die verschiedenen **kommerziellen Bereichen**, wie z.B. Airportsysteme, Verkehrssysteme, Geosysteme, Telekommunikation usw., zugeordnet werden können.

Dabei gehört ein Dienste zu einem speziellen Bereich und nicht zu mehreren. Ein Dienst, der für verschiedene Marktsegmente benötigt wird oder verwendet werden kann, ist ein potentieller Kandidat für ein horizontales Common Facility.

Nachfolgend eine Übersicht über die Lage der Common Facilities innerhalb der Object Manangement Architekture:

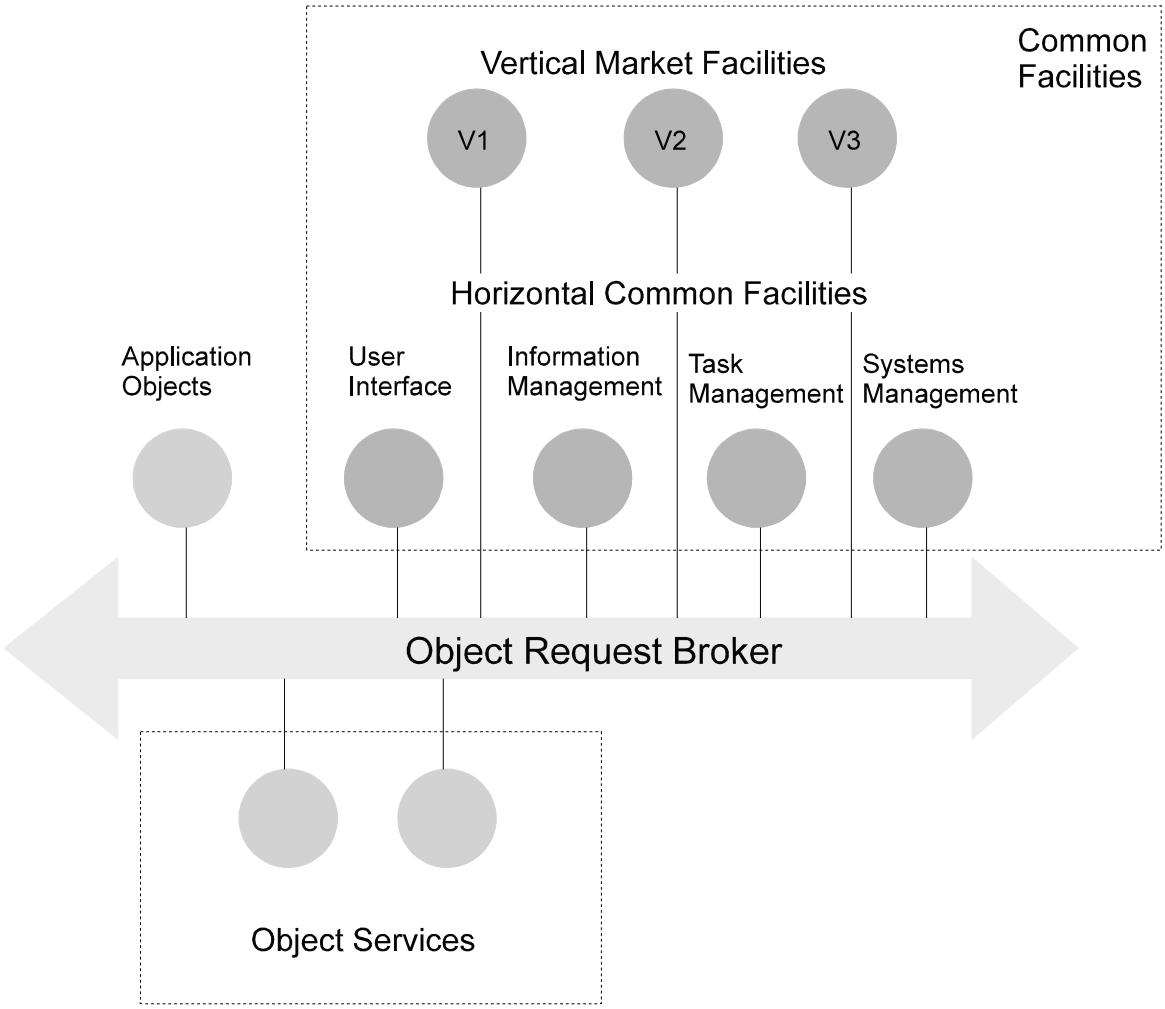


Abbildung 7-7: Aufteilung Common Facilities

Die Grenzen zwischen **Common Facilities** und **Object Services** sind fließend, d.h. der derzeitige Stand der Common Facilities und der Object Services beruht auf der heutigen Sichtweise der OMG im Standardisierungsprozeß.

Wenn die OMG mehr Erfahrungen sammelt, dann ist es durchaus möglich, daß ein heutiges vertikales Common Facility zu einem Horizontalen Facility wird (z.B. wenn es für verschiedene kommerzielle Bereiche benötigt wird) oder daß ein Common Facility zu einem Object Service wird.

7.2.3 Beschreibung der horizontalen Common Facilities

Wie oben schon beschrieben, beinhalten die horizontalen Common Facilities **Applikations**- und **User spezifische Funktionen**, die durch ihren grundsätzlichen Charakter nicht einem bestimmten Marktsegment zugeordnet werden können.

Diese Dienste werden nicht für die Dienstleistung zwischen Applikationen und Object Request Broker benötigt (hierfür gibt es die Object Services), sondern für die Dienstleistungen zwischen den **Applikationen/Systemen** untereinander.

Es gibt inzwischen eine ganze Reihe von horizontalen Common Facilities, die nach folgenden Gebieten gruppiert werden.

- **User Interface**
  Ermöglicht den Zugriff eines Users auf die Applikation oder das System. Zum Beispiel gehört die Compound Presentation Facility in die Gruppe User Interface.

- **Information Management**
  Gruppiert die Modellierung, Definition, Speicherung, Wiederherstellung, Verwaltung und der Austausch von Informationen. Das Facility Compound Interchange gehört z.B. zur Gruppe Information Management.

- **System Management**
  Gruppiert die Verwaltung von komplexen Informationssystemen durch Dienstanbieter.

- **Task Management**
  Gruppiert die Prozeßautomatisierung und beinhaltet sowohl die Automatisierung von User-Prozessen als auch System-Prozessen.

Die nachfolgend beschriebenen horizontalen Dienste sind teilweise noch nicht vollständig definiert und können sich somit noch ändern. Außerdem unterliegen die gesamten Common Facilities noch einer starken Evolution.

7.2.3.1 User Interface Dienste

Nachfolgend eine Liste der derzeitigen, von der OMG beschriebenen, User Interface Common Facilities:

User Interface Common Facility	Beschreibung
Rendering Management	Beinhaltet Dienste für die Darstellung und Ausgabe von Informationen auf Geräten wie z.B. Monitor, Drucker, Plotter, Soundkarten und für die Eingabe von Daten durch Geräte wie z.B. Tastatur, Maus, Scanner, Mikrofone. Hierbei werden folgende Dienste zur Verfügung gestellt: • Window Verwaltung • Klassenbibliothek für User-Interface-Objects • Dialogobjekte wie zum Beispiel Menübars, Scrollbars usw. • Bereitstellung von virtuellen Geräten, um die verschiedenen Eingabe- und Ausgabegeräte abstrahieren zu können.
Compound Presentation Management	Dieser Dienst stellt einen Rahmen zur Verfügung, um Fenster auf dem Display aus verschiedenen Teilen zusammensetzen und verteilen zu können, d.h Fenster können aus verschiedenen grafischen Elementen (z.B.

	Schaltflächen, Textzeilen, Listen usw.) oder wiederum aus Fenstern zusammengesetzt sein. Diese Teile können wiederum aus anderen Teilen zusammengesetzt sein, d.h. der gesamte Aufbau eines Fensters kann tief verschachtelt werden. Dieses Facility beschreibt den Display-Teil einer allgemeinen Architektur für zusammengesetzte Dokumente (Compound Document Architekture).
User Support Facilities	Zu den User Support Facilities gehören alle Dienste, die in verschiedenen Applikationen gleichartige Aufgaben erfüllen. Der Sinn der User Support Facilities ist, dem Benutzer eine gleichartige Oberfläche und ein gleichartiges Verhalten (Look & Feel) für dieselbe Aufgabe innerhalb verschiedener Applikationen bereitzustellen und dem Entwickler einen wiederverwendbaren Code mit standardisierten Schnittstellen zur Verfügung zu stellen. Derzeit sind folgende Aufgaben als Dienst in der User Support Facilities definiert, aber in der Zukunft werden noch weitere Dienste hinzukommen: • Hilfesystem • Rechtschreibprüfung Folgende Dienste sind in Vorbereitung: • Versionsverwaltung • Funktionen für Tabellenkalkulationen und Grafik
Desktop Management Facilities	Die Desktop Management Facilities stellen eine generelle Struktur für die Darstellung der Benutzerumgebung zur Verfügung.
Scripting Facilities	Zur Scripting Facilities gehören folgende Dienste: Eine Interpreter Sprache für die funktionale Zerlegung, die benötigt wird, um Skripte als Agenten zu versenden. Bereitstellung von Tastaturschnittstellen zur Verwendung in Common Facilities, Object Services und ORB Facilities auf Sprachebene. Eine grafische Entwicklungsumgebung, wie derzeit z.B. Microsoft Visual Basic, zur Verfügung. Hierzu gehören auch Funktionen für die Programmierung und Anwendung

	von Makros innerhalb von Applikationen.

7.2.3.2 Information Management Dienste

Nachfolgend eine Liste der derzeitigen, von der OMG beschriebenen, Information Management Common Facilities:

Information Management Common Facilities	Beschreibung
Information Modeling Facilities	Die Information Modeling Facilities definieren Dienste für die Erzeugung und Darstellung von Informationsmodellen. Die Information Modeling Facilities werden z.B. für objekt-orientierte Datenbanken, für Case-Tools und das Software-Engineering benötigt.
Information Storage and Retrieval Facilities	Stellt Dienste für die persistente Speicherung von Informationen und die Wiederherstellung der Informationen zur Verfügung. In die Kategorie dieser Facilities gehören z.B. folgende Standards und Referenzen: • SGML • HTML • HTTP • SQL • ODBC • WAIS • WWW
Compound Interchange Facilities	Definiert Dienste für den Austausch von Daten innerhalb verschachtelter Dokumente. Hierzu gehören: • Bindung der Datenobjekte an einen Presentation Manager • Die Verknüpfung dieser Datenobjekte mit zusätzlichen Informationen • Die Konvertierung der Datenobjekte in verschiedene Typen • Austausch der Datenobjekte sowohl online (z.B. über

	Drag-and-Drop oder die Zwischenablage) und Offline (z.B. E-Mail oder Disketten).
Data Interchange Facilities	Stellt Dienste für den generellen Austausch von Daten zur Verfügung. Dies wird z.B. durch die Definition von gemeinsamen Datenformaten (z.B. TIFF, GIF, EPS, RTF, usw.) erreicht.
Information Exchange Facilities	Stellt Dienste für den Austausch von Informationen zwischen Informationseinrichtungen bereit.
Data Encoding and Representation Facilities	Stellt Dienste für die Verschlüsselung und Übersetzung von Daten zur Verfügung. Hierzu gehören z.B. folgende Standards: • MIME • MPEG • XDR • V.42bis • ASN.1
Time Operations Facilities	Stellt Dienste für die Verarbeitung von Kalender- und Zeitdaten zur Verfügung.

7.2.3.3 System Management Dienste

Nachfolgend eine Liste der derzeitigen, von der OMG beschriebenen, System Management Common Facilities:

System Management Common Facility	Beschreibung
Policy Management Facility	Wird für die Kontrolle zum erzeugen, löschen und modifizieren von managebaren Objekten verwendet.
Quality of Service Management Facility	Dieser Dienst stellt ein Interface zur Unterstützung der Auswahl verschiedener Dienstmerkmale bereit. Wählbare Dienstmerkmale sind die Verfügbarkeit, der Durchsatz, die Zuverlässigkeit und die Möglichkeit zur Wiederherstellung.
Instrumentation Facilities	Stellen Interfaces für die Erfassung, Handhabung und Verbreitung von resoucenspezifischen Daten bereit, um Systemmanagement zu unterstützen.
Data Collection Facilities	Diese Facilities bieten eine Schnittstelle für die Informationserfassung in Unterstützung des Systemmanagements. Derzeitig existieren 2 Facilities: • Logging Management

	• History Management
Security Facilities	Die Security Facilities stellen ein allgemeines Interface für das Handling der Sicherheitsmechanismen der Systemressourcen bereit. Diese unterscheiden sich von der Implementation der Security Mechanisms.
Instance Management Facilities	Definiert grundlegende Operationen, die das Handling von multiplen Instanzen einer Klasse ermöglichen.

Die System Management Facilities realisieren Dienste, wie z.B. SNMP (Simple Network Management Protocol) innerhalb der OMA.

7.2.3.4 Task Management Dienste

Nachfolgend eine Liste der derzeitigen, von der OMG beschriebenen, Task Management Common Facilities:

Task Management Common Facility	Beschreibung
Workflow Facilities	Bei komplexeren Applikationen ist es oftmals notwendig, Objekte innerhalb verschiedener Arbeitsprozesse zu koordinieren. Als einfaches Beispiel sei hier die Auftragsbearbeitung in einem Unternehmen aufgeführt. Ein neuer Auftrag muß verschiedene Arbeitsprozesse durchlaufen, bis am Ende das fertige Produkt beim Kunde ist. Die Verschiedene Prozesse sind hierbei stark voneinander abhängig. Um dies zu koordinieren, werden Workflow Dienste benötigt.
Agent Facilities	Diese Facilities stellen Dienste für statische und dynamische Agenten zur Verfügung.
Rule Management Facilities	Diese Facilities stellen Dienste für regelbasierende Systeme zur Verfügung. In regelbasierenden Systemen wird eine Situation mit Hilfe von verschiedenen Regeln bewertet. Als Beispiel sei hier die Konflikterkennung auf einem Flughafenrollfeld dargestellt. Hierbei genügt es nicht einen Konflikt nur dadurch zu erkenne, ob z.B. zwei Flugzeuge sehr nah beieinander stehen oder sich bewegen, es könnte ja sein, daß diese Flugzeuge sich parallel zueinander bewegen -> und schon ist eine Regel geboren.

	Um z.B. ein Flughafenrollfeld in Bezug auf mögliche Konflikte, die dann dem Kontroller dargestellt werden, bewerten zu können, ist also ein System nötig, welches Regeln versteht und verarbeiten kann. Hierbei können verschiedene Dienste zur Situationsbewertung, Festlegung und Wartung von Regeln mittels eines Tools die Aufgabe erleichtern.
Automation Facilities	Ermöglicht den Zugriff auf Schlüsselfunktionen zwischen verschiedenen Objekten, um z.B. Abläufe zu automatisieren.

7.2.4 Beschreibung der vertikalen Common Facilities

Wie oben schon beschrieben, beinhalten die vertikalen Common Facilities Dienste, die für spezielle **Marktsegmente** benötigt werden.

Diese werden zwar von der Industrie entwickelt, die OMG hilft jedoch den Firmen dabei, diese Dienste in die Common Facilities zu integrieren.

Zum Beispiel gibt es derzeit folgende vertikale Facilities:

- **Information Superhighways Facility**
  Stellt Dienste für verteilte Multiuser Informationssysteme über Wide-Area-Networks (WAN) zur Verfügung.

  Hierzu gehört z.B. Telekonferenzen.

- **Accounting Facility**
  Stellt Dienste für kommerzielle Transaktionen zur Verfügung.

  Hierzu gehören z.B. Geschäftsabwicklungen über das Internet (Bestellung und Bezahlung) und Übermittlung von Aufträge an Banken (Überweisungsauftrag, setzen einer Order für Aktiengeschäfte usw.)

- **Mapping Facility**
  Stellt Dienste für die Representation und Verarbeitung von geographischen Daten eines GIS (Geographical Information System) - Tools bereit.

  Hierzu gehören z.B. Fahrzeugnavigationsysteme

8 Strategien zur Integration von Interfaces

Üblicherweise steht beim Design von Software die Funktionalität im Vordergrund, d.h. man entwickelt auf verschiedene Arten ein Softwaremodell, das den Problembereich möglichst genau abbildet.

Während der **Analysephase** und der frühen Entwurfsphase werden also die Schnittstellen (Interfaces) zum System nicht, oder fast nicht betrachtet. Dies ist auch nicht notwendig, da die Interfaces lediglich eine bestimmte Sicht auf ein System zur Verfügung stellen.

Darüber hinaus steht während der frühen Designphase oft noch nicht fest, wer auf welche Art mit dem abgebildeten System kommunizieren soll und es steht nicht fest, welche Teile eines Gesamtsystems auf welche Art verteilt werden sollen.

Es ist also oftmals notwendig, nachträglich Interfaces zu definieren, die einen **Zugriff** auf das System zur Verfügung stellen. Da Interfaces nur eine Sicht auf ein System beschreiben, ist es möglich, verschiedene Interfaces für dasselbe System zu definieren.

Damit dies möglich ist, muß das System unabhängig von Interfaces modelliert werden, d.h. im Modell wird nur die **Funktionalität** und nicht der Zugriff auf das System betrachtet.

Um ein Interface in ein System zu integrieren, stehen drei Möglichkeiten zur Verfügung:

- Funktionalität und Interface-Implementation innerhalb einer Klasse
- Interface-Implementation als Spezialisierung einer bestehenden Klasse
- Integration der Interface-Implementation in ein bestehendes Objektmodell

Unter **Interface-Implementation** wird hierbei die konkrete Realisierung des definierten Interfaces verstanden, d.h. Methoden und Klassen, die bisher nur durch ihre Attribute und Parameter definiert sind, werden „mit Leben gefüllt“.

Nachfolgend werden die einzelnen Möglichkeiten zur Integration von Interfaces in ein System beschrieben.

8.1 Funktionalität in der Interface-Implementation

Es kann durchaus Sinn machen, die **Funktionalität** einer Applikation oder Bibliothek direkt innerhalb der **Implementationsklasse** zu realisieren, d.h. nicht nur die Schnittstelle zu implementieren, sondern auch die Funktionalität, auf die die Schnittstelle aufsetzt, direkt zu realisieren.

Hierbei sieht das OMT-Modell sehr einfach aus. Die abstrakte (und generierte) Implementations-Basisklasse wird zu einer reellen Implementationsklasse abgeleitet. Innerhalb dieser reellen Implementationsklasse wird dann nicht nur die Funktionalität für die Schnittstellen, sondern auch die Applikationsfunktionalität realisiert.

Nachfolgend das OMT-Modell:

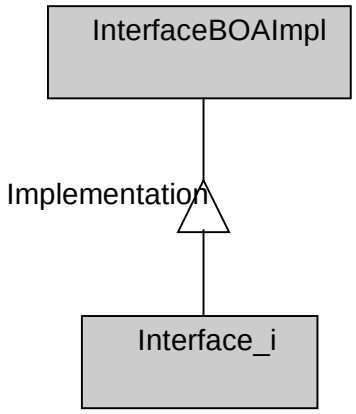


Abbildung 8-1: Funkionalität in der Interfaceimplementation

Diese Vorgehensweise macht dann hauptsächlich Sinn, wenn sichergestellt ist, daß nur eine Schnittstelle definiert werden muß.

Dies ist z.B. der Fall, wenn eine **Dienstleistung** für CORBA realisiert werden soll. Hierbei ist die Funktionalität direkt mit CORBA verknüpft, d.h. die Funktionalität wird nur für die Verwendung von CORBA benötigt und ansonsten nicht.

Als konkretes Beispiel sei hier eine **Watchdog-Funktionalität** genannt. Dieser Watchdog soll sicherstellen, daß

- der Client erkennt, wenn der Server beendet/neugestartet wurde
- der Server erkennt, wenn der Client beendet wurde

Die Funktion des Watchdog steht in direktem Zusammenhang mit CORBA, d.h. als Plattform kommt nur CORBA in Frage und somit gibt es nur eine Schnittstelle. Die Funktion und die Schnittstelle für die Kommunikation zwischen Watchdog-Server und Watchdog-Client gehören somit zusammen.

Sobald mehrere Schnittstellen für eine Applikation/Funktionalität definiert werden soll, d.h. wenn verschiedene Clients auf die Funktionalität zugreifen (d.h. sie benötigen), ist dieses Modell nicht mehr brauchbar.

8.2 Funktionalität in Basis-Klasse

Bei kleineren Applikationen/Bibliotheken kann es vorkommen, daß die gesamte Funktion innerhalb einer Klasse realisiert wird. Wird diese Klasse in verschiedenen Teilen (z.B. weil sie Teil einer Klassenbibliothek ist) benötigt, so ist das Interface dieser Applikationsklasse unterschiedlich zum „**Remote-Interface**“ (d.h. Interface für CORBA Zugriff), vor allem wenn verschiedene Clients auf die gleiche Applikation zugreifen.

Dies gilt auch, wenn eine bestehende Interfaceklasse von verschiedenen Clients verwendet wird. Hierbei gibt es dann für jeden Clienttyp ein eigenes Interface, welches den Zugriff des Clients auf das globale Interface matched.

Das OMT-Modell für diese Interface-Implementation ist folgend dargestellt:

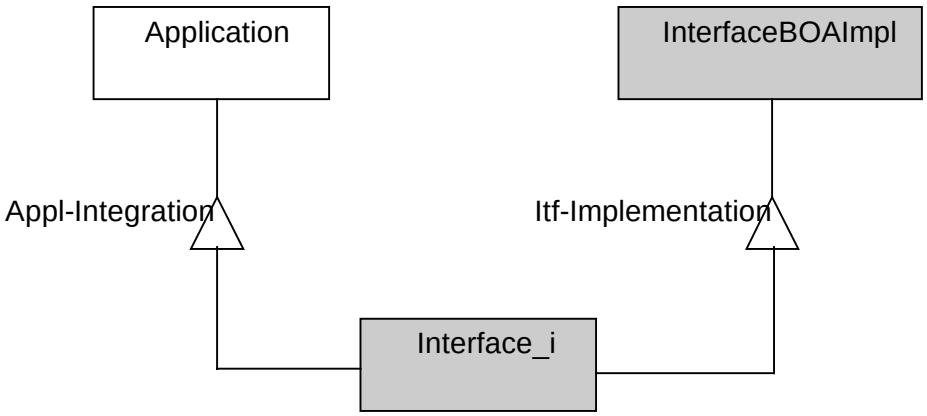


Abbildung 8-2: Funktionalität in der Basisklasse

Hierbei wird eine Mehrfachvererbung verwendet, d.h. die Implementation des Interfaces wird von folgenden Klassen abgeleitet:

- **Applikationsklasse**
  hierdurch erhält die Interface-Implementationsklasse die Funktionalität der Applikation oder die Schnittstellen, um auf die Applikation zugreifen zu können.

- **generierte (abstrakte) Implementationsklasse**
  hierdurch erhält die Interface-Implementationsklasse die Definition der Schnittstellen und die CORBA-Funktionalität.

Beides zusammen ergibt dann eine Klasse, die die Funktionalität der Applikation besitzt und via CORBA einen externen Zugriff ermöglicht.

Der Vorteil bei dieser Vorgehensweise liegt darin, daß

- die Funktionalität und die Implementierung des Interface **getrennt** ist, somit die Struktur verbessert wird.
- **mehrere** verschiedene **Interfaces** für eine Applikation/Funktion definiert und implementiert werden können.

- Die Applikation/Funktion **keinen CORBA Overhead** besitzt und somit ohne Verluste auch innerhalb verschiedener Teile ohne CORBA Zugriff verwendet werden kann.

Der Nachteil dieser Vorgehensweise liegt darin, daß

- **Mehrfachvererbung** verwendet wird, d.h. das Modell ist nicht ohne Veränderung auf Programmiersprachen übertragbar, die keine Mehrfachvererbung zulassen.

Die Probleme mit der fehlenden Mehrfachvererbung können z.B. dadurch verhindert werden, indem die Applikationsklasse in Form einer Aggregation an die Interface-Implementierung gebunden wird.

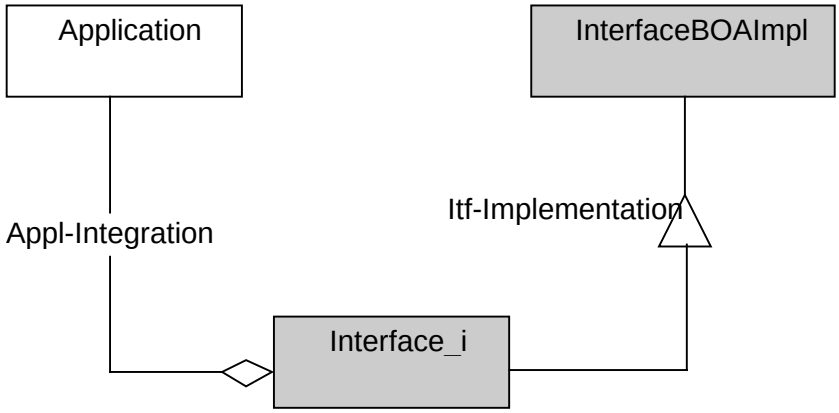


Abbildung 8-3: Funktionalität in der Basisklasse ohne Mehrfachvererbung

8.3 Funktionalität in Objektmodell

Im Normalfall sieht es so aus, daß ein Interface einen Zugriff auf ein komplexes System definiert.

Hierbei wird die Interface-Implementation in das Objektmodell eingefügt.

Das OMT-Modell für diese Interface-Implementation ist folgend dargestellt:

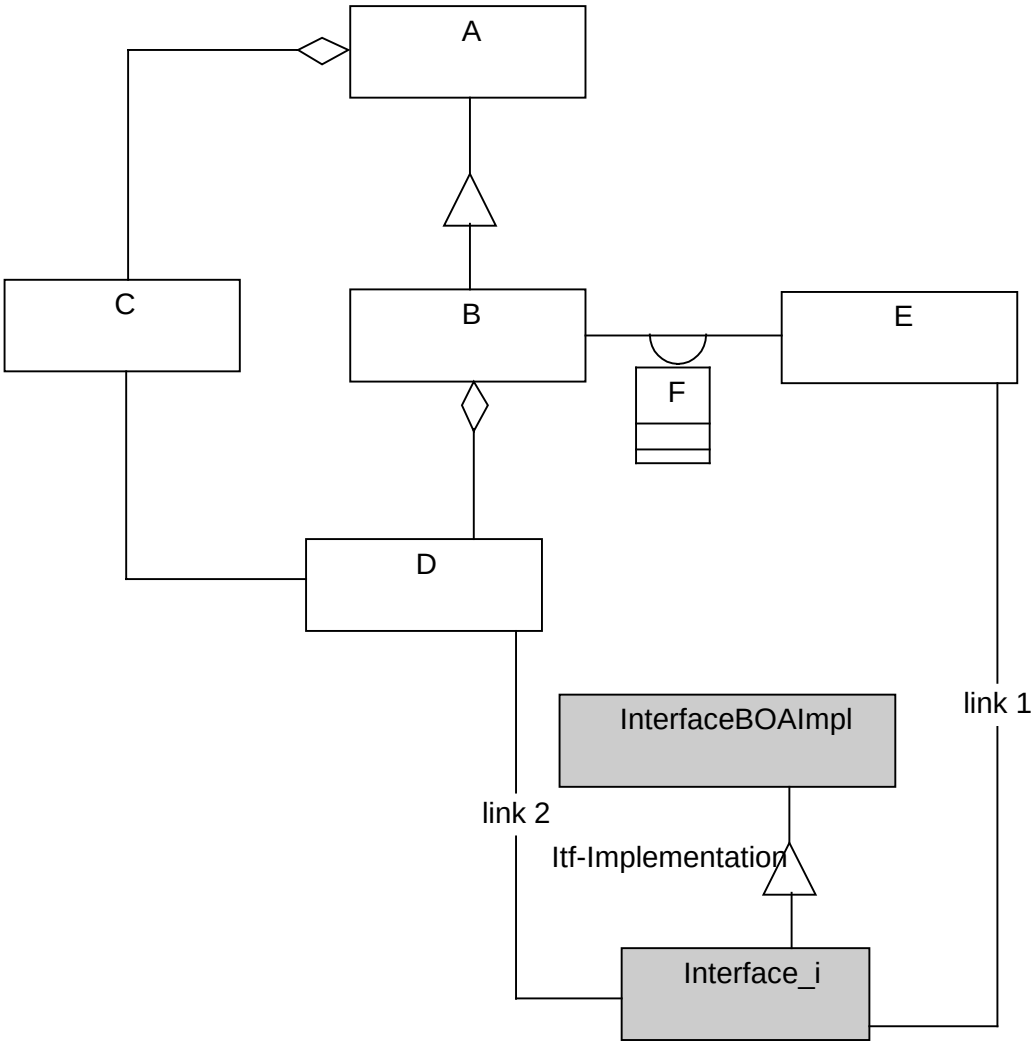


Abbildung 8-4: Funktionalität im Objektmodell

In diesem Beispiel besitzt die Interface-Implementation eine Beziehung zu den Klassen D und E, d.h. für die Realisierung der Schnittstelle werden Zugriffe auf Instanzen von D und E benötigt.

Bei der Realisierung der Interface-Implementations Klasse muß sichergestellt sein, daß die Verbindungen hergestellt werden.

8.4 Interface verschiedener Projekte

Nachfolgend ist noch die Integration verschiedener Interfaces in ein bestehendes Objektmodell dargestellt:

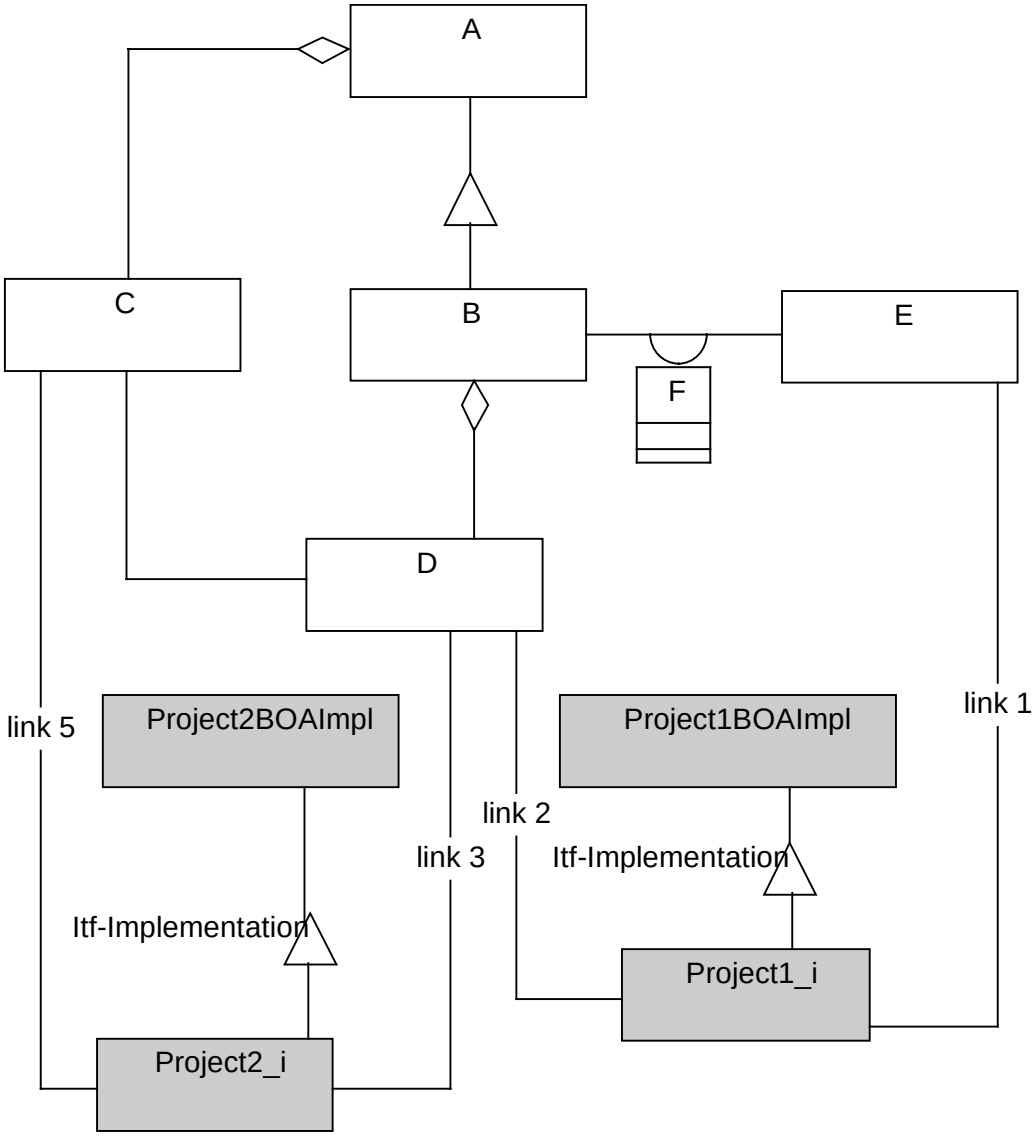


Abbildung 8-5: Integration in bestehendes Objektmodell

Die zwei Interfaces Project1 und Project2 haben nichts miteinander gemeinsam. Es ist jedoch durchaus denkbar, Interfaces und damit **Interface-Implementationen** voneinander abzuleiten oder in eine **Beziehung** zueinander zu setzen.

Wenn verschiedene Interfaces miteinander in Beziehung gesetzt werden, macht dies allerdings nur Sinn, wenn die **Lebenszeit** dieser Interfaces identisch ist, d.h. die Interfaces vom gleichen Client verwendet werden.

8.5 Beispiel

Nachfolgend ein einfaches Beispiel.

Das Beispiel definiert einen einfachen **Container**, der Integerwerte trägt und einen Iterator, mit dem lediglich ein **sequenzieller Zugriff** auf den Container möglich ist, d.h. auf die einzelnen Elemente des Containers kann nur nacheinander zugegriffen werden.

Es wird danach ein Interface definiert, welches über CORBA einen wahlfreien **Index-Zugriff** auf den Container ermöglicht, d.h. es kann auf ein bestimmtes Element des Containers mittels eines Index zugegriffen werden.

Das Beispiel wurde unter Solaris 2.5 getestet. Als Compiler wurde der Sparc Compiler 4.1 und als ORB der Orbix 2.1c verwendet.

8.5.1 OMT Modell

Nachfolgend das OMT Model des Beispiels. Das Modell ist ebenfalls in den funktionellen Teil und den Interface Teil aufgeteilt.

8.5.1.1 Funktioneller Teil

Beschreibung:

Die Klasse `Container` trägt eine Liste mit Objekten der Klasse `Item`. Die maximale Anzahl der verwalteten `Item`-Objecte wird durch ein Attribut innerhalb der Klasse `Container` festgelegt.

Die Klasse `Iterator` besitzt über eine Beziehung einen Zugriff auf die Elemente die in einer Instanz der Klasse `Container` gespeichert sind. Der Eingabeoperator der Klasse `Iterator` ermöglicht einen **sequentiellen Zugriff** auf die gespeicherten Elemente.

Die Klasse `Item` kapselt einen Integer-Wert und definiert verschiedene Schnittstellen für einen Lesezugriff auf den Wert.

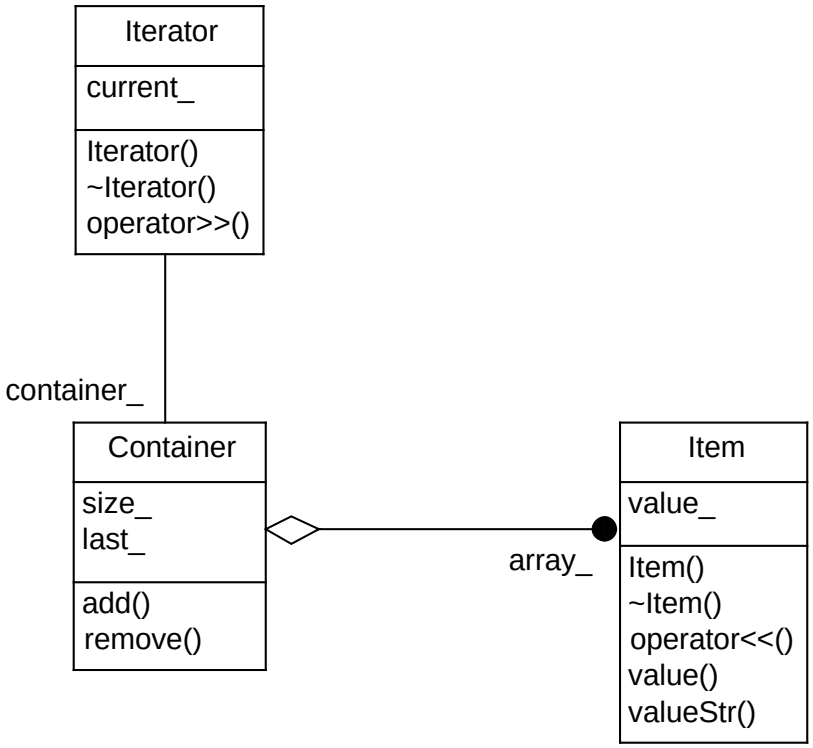


Abbildung 8-6: Funktionales Objektmodell

8.5.1.2 Interface Teil

Beschreibung:

Die Klassen `CORBA_Object`, `Riterator_var` und `RIteratorBOAImpl` werden durch den IDL-Compiler generiert und sind deshalb als extern definiert (Sichtbar am „e“ in der rechten oberen Ecke).

Die Klasse `RiteratorBOAImpl` kapselt die Registrierung der Instanz beim Basic Object Adapter (BOA).

Die Klasse `Riterator_i` realisiert einen **wahlfreien Zugriff** auf eine Instanz der Klasse `Container`, d.h. sie ermöglicht einen wahlfreien Zugriff auf die Elemente, die im Container gespeichert sind. Dieser Zugriff kann transparent über eine verteilte Architektur erfolgen.

Die Klasse `Riterator_var` kapselt einen Remote-Zugriff auf die Instanz der Klasse `Riterator_i` indem sie als Proxy arbeitet, d.h. ein Zugriff auf eine Instanz der Klasse `Riterator_var` wird über das Netz an eine Instanz der Klasse `Riterator_i` weitergeleitet. Dieser Zugriff erfolgt netz-transparent, d.h. es gibt keinen Unterschied zwischen dem lokalen Zugriff innerhalb einer Applikation oder einem remote Zugriff über Netzwerkgrenzen hinweg.

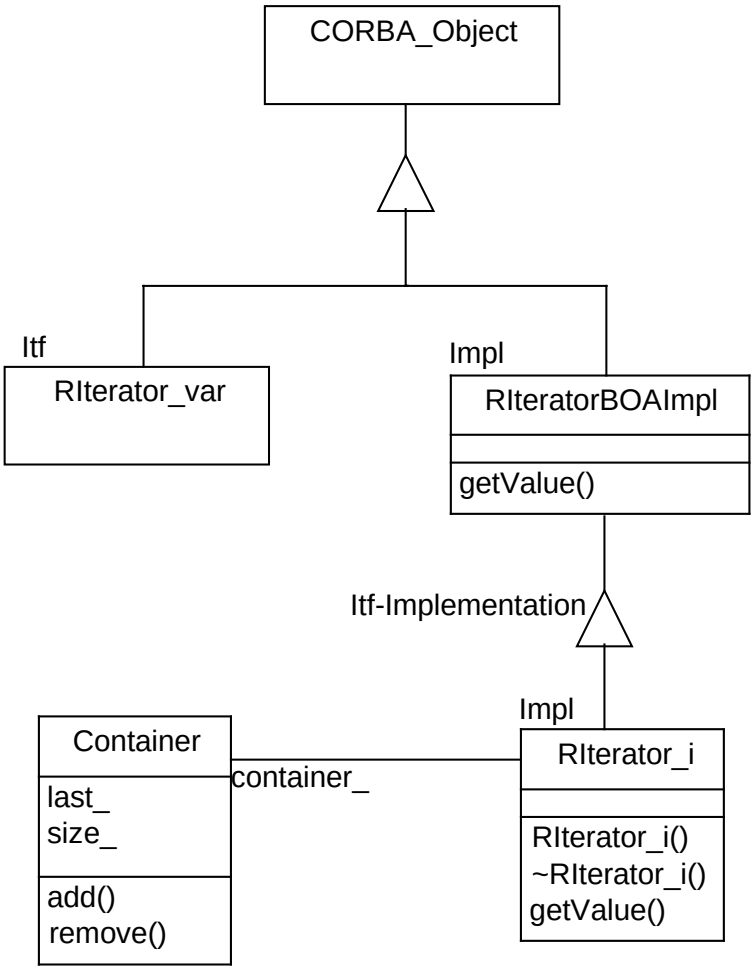


Abbildung 8-7: Objektmodell mit Interface Teil

8.5.2 Sourcecode Funktioneller Teil

Nachfolgend der Sourcecode für den funktionellen Teil des Beispiels, d.h. der Teil, der die Funktionalität realisiert.

8.5.2.1 Container-Headerfile

```
#ifndef __CONTAINER_H__
#define __CONTAINER_H__

# include <stdlib.h>
# include <iostream.h>
# include <strstream.h>

class Container
{
public:
  class Item
  {
    int value_;
  public:
```

```
    Item( int value ) : value_( value ) {};
    Item( const Item &i ) : value_( i.value_ ) {};
    ~Item() {};
    friend ostream &operator<<( ostream &os, const Item &i) {
      return os << i.value_;
    };

    int value() const { return value_; };
    const char* valueStr() const {
      ostrstream buffer;
      buffer << value_ << ends;
      return buffer.str();
    };
  };

  typedef Item *Item_ptr;

  friend class Iterator
  {
    size_t current_;
    Container &container_;

  public:
    Iterator( Container &container )
      : container_(container)
      , current_( 0 )
    {
    };
    ~Iterator() {};

    Iterator &operator>>( Item_ptr &item)
    {
      item = 0;
      if( current_ == container_.size_ ) return *this;
      item = container_.array_[current_];
      current_++;
      return *this;
    };

    operator int()
    {
      return ( current_ == container_.size_ ) ? 0 : 1;
    };
  };

private:
  Item_ptr *array_;
  size_t   size_;
  size_t   last_;

public:
  Container( size_t size );
  ~Container();

  size_t size() const { return size_; };
```

```
  int add( Item_ptr item );
  int remove( size_t from, size_t number );

  friend ostream &operator<<( ostream &os, const Container &c ) {
    if( !c.array_ ) return os;

    for( int i=0; i<c.size_; i++ ){
      os <<"[" << i << "]: "<<((c.array_[i])
                                ? c.array_[i]->valueStr()
                                : "NULL") << endl;
    }
    return os;
  };
};
#endif // __CONTAINER_H__
```

8.5.2.2 Container-Sourcefile

```
# include <Container.h>
# include <iostream.h>

Container::Container( size_t size )
  : size_( size )
  , array_( 0 )
  , last_( 0 )
{
  array_ = new Item_ptr[ size ];
  if( array_ ) {
    for( int i=0; i<size; i++ ) array_[i] = 0;
  }
}

Container::~Container()
{
  if( array_ ) {
    for( int i=0; i<size_; i++ ) {
      if( array_[i] ) delete array_[i];
      array_[i] = 0;
    }
    delete[] array_;
  }
  array_ = 0;
  size_  = 0;
  last_  = 0;
}

int Container::add( Item_ptr item )
{
  if( !array_ ) return 0;
  if( array_[last_] ) return 0;
  if( !item ) return 0;
  array_[last_] = item;
  last_++;
  return 1;
```

```
}

int Container::remove( size_t from, size_t number )
{
 if( !array_ ) return 0;
 if( !number ) return 0;
 if( !(from < size_) ) return 0;
 if( from < 0 ) return 0;
 for( int i=0; i<number; i++ ) {
   Item_ptr &item = array_[i+from];
   if( item ) delete item;
   item = 0;
 };
}
```

8.5.2.3 lokaler Zugriff auf Container

```
# include <Container.h>
# include <iostream.h>

int main( int argc, char **argv )
{

  Container container( 10 );
  for( int i=0; i<10; i++ ) {
    container.add( new Container::Item( i*10+i ) );
  }
  cout << container << endl;

  Container::Iterator iterator( container );
  Container::Item_ptr item;
  while( iterator >> item ) cout << item->value() << endl;

  return 0;
};
```

8.5.3 Sourcecode Interface Teil

Nachfolgend der Sourcecode für den Interface Teil des Beispiels, d.h. der Teil, der das Interface realisiert

8.5.3.1 IDL-File für Remote-Iterator

```
interface RIterator {
  long getValue( in long index );
};
```

8.5.3.2 Headerfile für Implementation des Remote-Iterator

```
#ifndef __CONTAINER_I_H__
```

```
#define __CONTAINER_I_H__

# include <Container.h>
# include <RIterator.hh>

class RIterator_i
  : public RIteratorBOAImpl
{
  Container &container_;

public:
  RIterator_i( Container &container );
  ~RIterator_i();

  CORBA::Long getValue( CORBA::Long index,
                        CORBA::Environment &IT_env=
                    CORBA::IT_chooseDefaultEnv())
    throw (CORBA::SystemException);
};

#endif // __CONTAINER_I_H__
```

8.5.3.3 Sourcefile für Implementation des Remote-Iterator

```
# include <Container_i.h>

RIterator_i::RIterator_i( Container &container )
  : RIteratorBOAImpl()
  , container_( container )
{
};

RIterator_i::~RIterator_i()
{
};

CORBA::Long
RIterator_i::getValue( CORBA::Long index,
                       CORBA::Environment &IT_env )
    throw (CORBA::SystemException)
{
  if( !(index < container_.size()) ) return -1;

  Container::Iterator iterator( container_ );
  Container::Item_ptr item;
  for( int i=0; i<index; i++ ) iterator>>item;
  if( item ) return item->value();

  return -1;
};
```

8.5.3.4 Server Sourcefile

```
# include <Container_i.h>
# include <iostream.h>

int main( int argc, char **argv )
{

  Container container( 10 );
  for( int i=0; i<10; i++ ) {
    container.add( new Container::Item( i*10+i ) );
  }
  cout << container << endl;

  CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv, "Orbix");
  CORBA::BOA_ptr boa = orb->BOA_init(argc, argv, "Orbix");

  RIterator_i *r_iterator=0;
  r_iterator = new RIterator_i( container );

  CORBA::Orbix.impl_is_ready( "TestIterator",
                              CORBA::Orbix.INFINITE_TIMEOUT);

  if( r_iterator ) delete r_iterator;

  return 0;
};
```

8.5.3.5 Client Sourcefile

```
# include <RIterator.hh>
# include <iostream.h>

int main( int argc, char **argv )
{
  if( argc < 2 ) return 1;
  char *hostname = argv[1];

  try{
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv, "Orbix");
    CORBA::BOA_ptr boa = orb->BOA_init(argc, argv, "Orbix");

    RIterator_var r_iterator_i = RIterator::_bind( ":TestIterator",
                                                   hostname );
    cout << "5. Element = " << r_iterator_i->getValue( 5 ) << endl;
    // automatic release of r_iterator_i
  }
  catch( CORBA::SystemException &ex ) {
    cerr << ex << endl;
  };

  return 0;
}
```

9 Ein einfaches CORBA-Programm

Am Anfang gab es „Hello World“ ...
Dieses bekannte Programm soll auch hier als einführendes Beispiel dienen. An ihm soll die generelle Vorgehensweise gezeigt werden, wie ein CORBA-Programm entwickelt wird.

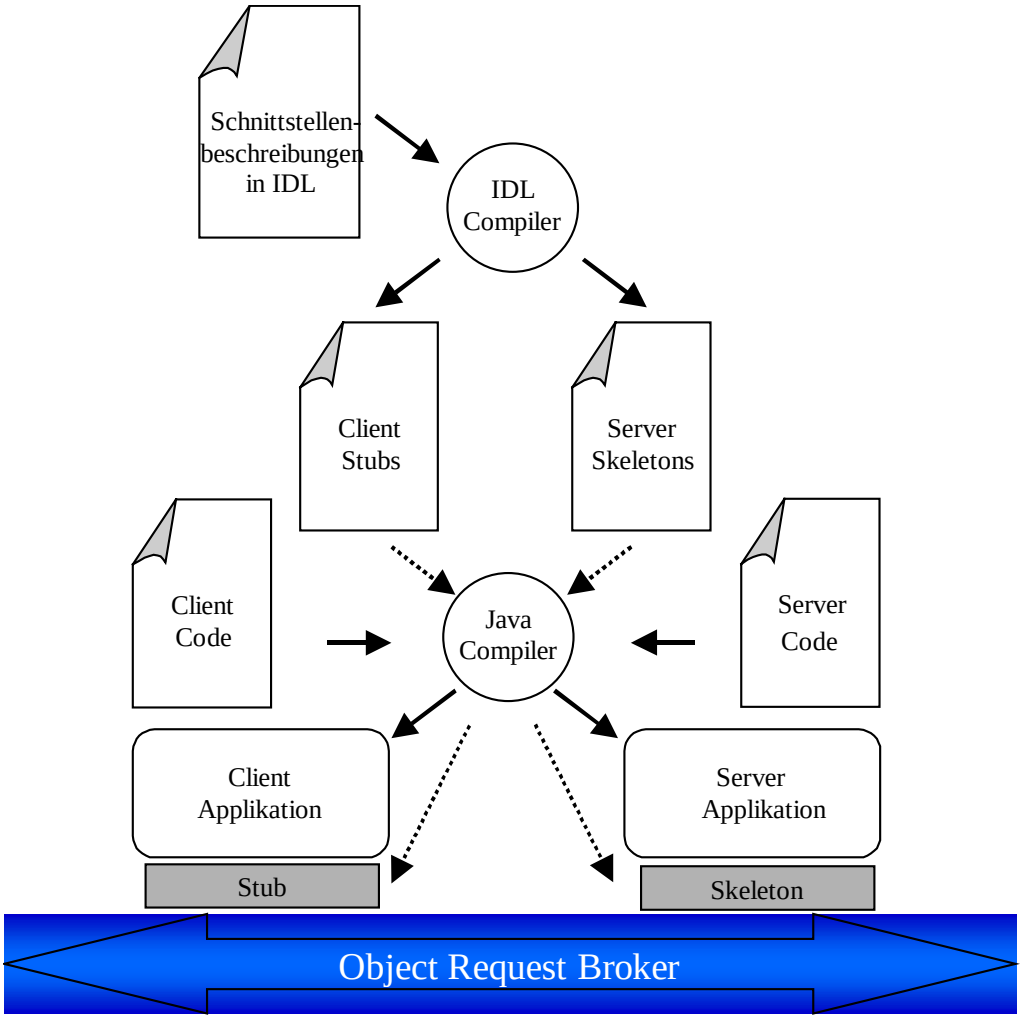


Abbildung 9-1: Generelle Vorgehensweise

9.1 Generelle Vorgehensweise

1) **Erstellung der Schnittstellenbeschreibungen mit Hilfe der Interface Definition Language**
Zuerst müssen sämtliche von den Objekten für den entfernten Zugriff angebotenen Operationen aufgezählt und ihre Aufrufschnittstellen beschrieben werden. Diese Beschreibung erfolgt mit der Interface Definition Language. Die IDL definiert die Typen der Objekte, ihre Attribute, ihre Methoden und ihre Übergabeparameter.

2) **Kompilierung mit dem IDL-Compiler**
Der IDL-Compiler erzeugt aus der IDL-Beschreibungsdatei die Client Stubs und die Server Skeletons. Diese Stubs und Skeletons sind programmiersprachenabhängig und dienen dem Client und dem Server als Gerüst, über welche ein Zugriff auf den ORB erfolgt.

3) **Implementierung des Clients, des Servers und der im IDL-File spezifizierten Methoden**
Der Programmierer muß nun noch den Client und den Server implementieren, welche sich auf das Gerüst der Stubs und Skeletons stützen.

4) **Kompilierung des Codes**
Der ganze Code wird jetzt mit einem Compiler zu lauffähigen Programmen compiliert. Man erhält jeweils eine Client- und eine Server-Version.

9.2 Die HelloWorld Interface Definition

```
// my first CORBA Hello World Program
module HelloWorld
{
 interface MyInterface
 {
  string getTheMessage();
 }
}
```

Der erste Schritt im Entwicklungsprozeß ist die Erstellung der programmiersprachenneutralen Interface Definitions. Ein IDL-File kann in einem normalen Texteditor erstellt werden und folgt gewissen Konventionen, die an die C++-Syntax angelehnt sind.

Im folgenden werden die im Beispiel benutzten Einheiten kurz erläutert:

module HelloWorld
Das Schlüsselwort module hat die Bedeutung, einen sogenannten Namespace bereitzustellen. Dies ist eine hierarchische Strukturierung, vergleichbar mit dem C++-Scope, um logische Einheiten zusammenzufassen.

interface MyInterface
Interfaces definieren eine Menge von Methoden. Sie sind mit Klassendefinitionen vergleichbar, jedoch ohne konkrete Implementationen. Im obigen Beispiel hat das Interface MyInterface nur die Methode `getTheMessage()`, welche einen Rückgabewert vom Typ String und keine Übergabeparameter besitzt.

Im zweiten Schritt werden die Definitionen des IDL-Files mit einem IDL-Compiler in eine konkrete Programmiersprache umgesetzt.

Für alle Beispiele wird der Visibroker for Java idl2java-Compiler eingesetzt.

Die Ergebnisse des Kompilierens werden durch das Language-Mapping bestimmt. Das Language-Mapping legt die Umsetzung von der Interface Definition Language in eine Programmiersprache fest. Diese Umsetzung wird von der Object Management Group festgeschrieben. Dieser Standardisierungsprozeß wurde mittlerweile für C++, Smalltalk und Ada beendet. Java befindet sich noch im Standardisierungsprozeß, Visigenic nutzt solange ein eigenes, nicht standardisiertes Language-Mapping.

```
prompt > idl2java –no_tie HelloWorld.idl
```

Das idl-file wird dem idl2java-Compiler als Argument übergeben.

In diesem Beispiel wird noch die Option *–no_tie* angegeben. Der IDL-Compiler von Visigenic bietet 2 Programmiermöglichkeiten:

inheritance-based mode: auf Vererbung basierend, wenn man objektorientierte Systeme neu entwirft und den Source-Code neu schreibt beziehungsweise die Mechanismen der Vererbung nützt.

delegation-based mode: auf Delegation basierend, wenn ganz alte nicht-objektorientierte Systeme mitintegriert werden müssen. Es werden besondere Stubs generiert die diese Integration erleichtern.

Die Option *–no_tie* unterdrückt die Generierung von diesen delegation-based Stubs.

Die module-Anweisung im IDL-File erwirkt, daß der idl2java-Compiler ein Java Package namens HelloWorld erzeugt.
Darunter verbergen sich verschiedene automatisch generierte Java Klassen und Interfaces, die im folgenden erläutert werden.

- **HelloWorld._sk_MyInterface** ist eine Java Klasse, welche die CORBA Server-Seite von HelloWorld implementiert. Sie übernimmt das Dekodieren der Argumente (unmarshalling) für das HelloWorld-Objekt. Zusätzlich werden in dieser Klasse die CORBA und Java Objektmodelle zusammengeführt (siehe HelloWorld - Klassenhierarchie).

- **HelloWorld._st_MyInterface** ist eine Java Klasse, welche den Client Stub für das HelloWorld Objekt implementiert. Sie ist eine interne Klasse, in der Methoden für das Kodieren (marshalling) zur Verfügung stehen.

- **HelloWorld.HelloWorldHelper** ist eine Java Klasse, die die bind-Methode zur Verfügung stellt. Diese Methode wird von den Clients für die Lokalisierung von Objekten dieser Klasse benötigt. Diese Klasse enthält weiterhin das Java-Mapping von IDL `out` und `inout` Übergabeparametern (siehe Kapitel 4). Java unterstützt standarmäßig nur call-by-value und nicht call-by-reference.

- **HelloWorld.MyInterface** ist ein Java-Interface. Die Aufgabe dieses Interfaces ist es, das HelloWorld-Interface auf das entsprechende Java-Interface abzubilden. Die Implementierung dieses Interfaces muß vom Programmierer in der Server-Klasse erfolgen (siehe HelloWorld-Klassenhierarchie).

- **HelloWorld._example_MyInterface** ist eine Java Klasse, die dem Programmierer als Erleichterung zur Verfügung gestellt wird. Sie stellt ein Gerüst dar, in welchem der Programierer „nur noch“ die eigentlichen Methoden ausprogrammieren muß.

Im folgenden wird die Klassenhierarchie des Servers aufgezeigt. Daraus werden die Zusammenhänge und Abhängigkeiten sichtbar.
Die Klasse *MyInterfaceImpl*, welche die Implementation des Servers beinhaltet, leitet sich von der abstrakten Klasse *_sk_MyInterface* ab. Diese Klasse muß vom Programmierer erstellt werden und beinhaltet die eigentlichen Methoden. Die Klasse *_sk_MyInterface* wird von der Klasse *org.omg.CORBA.portable.skeleton* abgeleitet und implementiert das Interface *MyInterface*. Diese Klasse, wie auch das Interface *MyInterface*, wird vom idl2java-Compiler automatisch generiert. Alle Skeletons werden von der Klasse *org.omg.CORBA.portable.skeleton* abgeleitet.

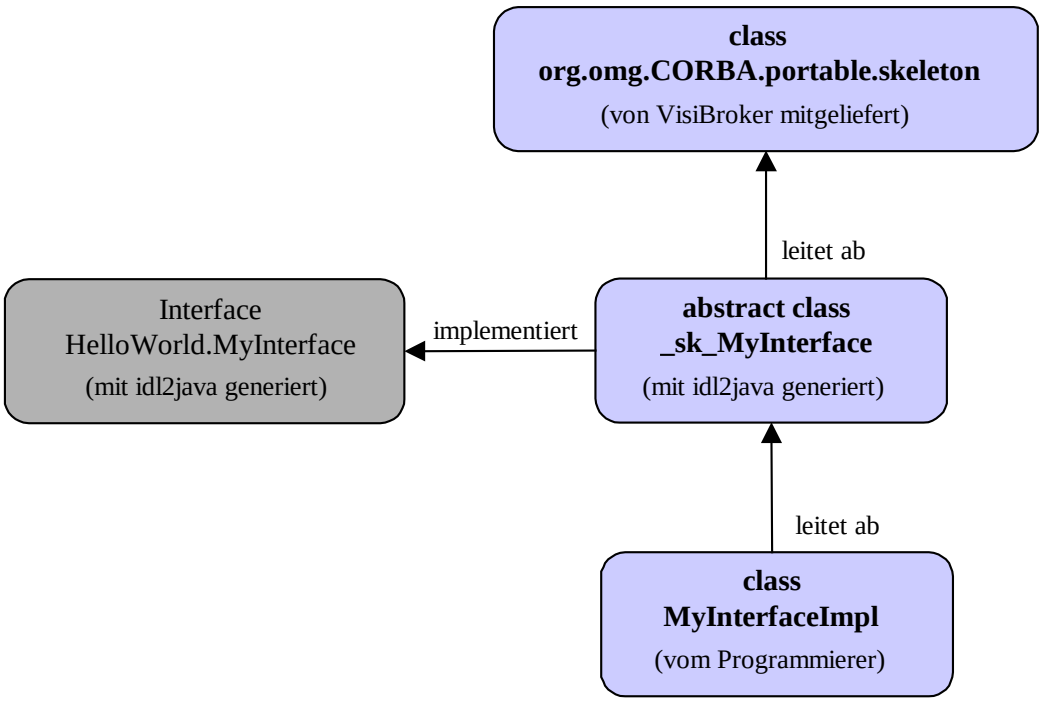


Abbildung 9-2: Klassenhierarchie des Servers

Jeder CORBA-Server muß ein Hauptprogramm haben, welches den ORB initialisiert und die Objekte startet. In diesem Fall wird dies die Klasse *HelloWorldServer* sein. Zusätzlich muß der Server die im IDL-File definierten Interfaces implementieren. Hier wird nur das Interface *HelloWorld.MyInterface* implementiert. Dazu wird eine Klasse *MyInterfaceImpl* geschrieben.

9.3 Die Implementation der Server-Klasse

```
package HelloWorld;

public class MyInterfaceImpl extends HelloWorld._sk_MyInterface {
  // Construct a persistently named object
  public MyInterfaceImpl(java.lang.String name) {
    super(name);
    System.out.println("World Object is here");
  }
  // Construct a transient object
  public MyInterfaceImpl() {
    super();
  }
  public java.lang.String getTheMessage() {

    // implement operation...
    String iGiveYouTheMessage = "Hello World";
    System.out.println("I got the message");
    return iGiveYouTheMessage;
 }
}
```

Die Server-Implementation wird bei VisiBroker immer von der Skeleton-Klasse, in diesem Fall von _sk_MyInterface, abgeleitet.
In diesem Fall wurde die _example_MyInterface-Datei als Vorlage verwendet, welche automatisch vom idl2java-Compiler generiert wurde.
Die Operation `getTheMessage()` wurde im IDL-File definiert und muß hier lediglich ausprogrammiert werden. In diesem Fall wird auf Server-Seite eine Meldung auf dem Bildschirm ausgegeben und der String „Hello World“ als Rückgabewert der Methode zurückgegeben.

9.4 Das Server-Hauptprogramm HelloWorldServer

Das Hauptprogramm muß – wie oben bereits erwähnt – verschiedene Initialisierungen vornehmen und die Objekte starten. In unserem Fall muß das Hauptprogramm folgendes tun:

1. den Object Request Broker (ORB) initialisieren
2. den Basic Object Adapter (BOA) initialisieren
3. das HelloWorld-Objekt starten
4. das neue Objekte dem ORB bekannt machen
5. auf Aufrufe (requests) warten

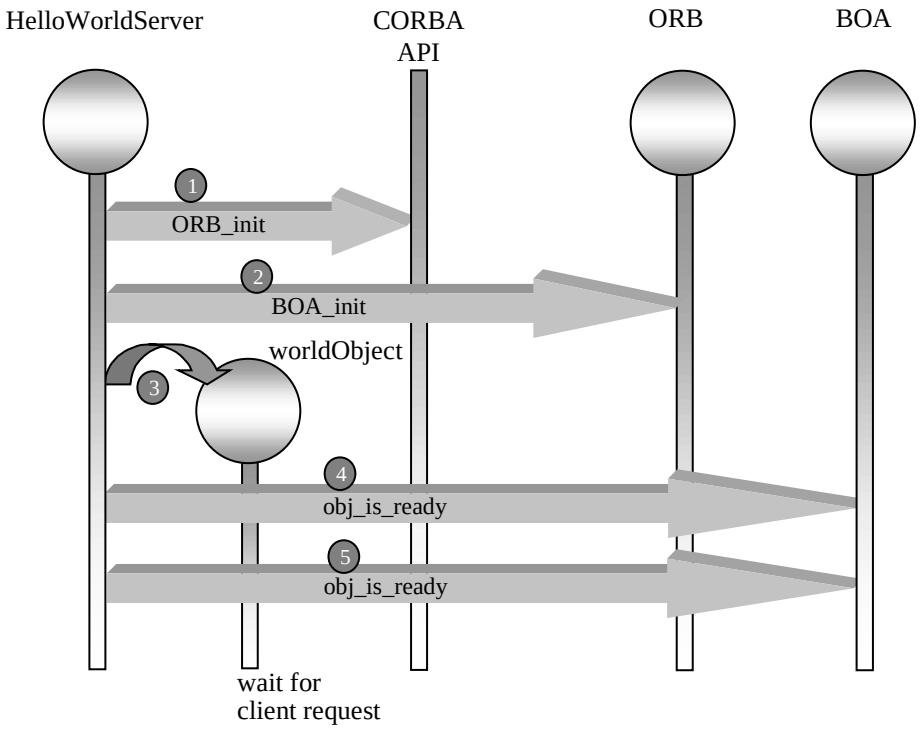


Abbildung 9-3: Objekt – Interaktions – Diagramm für den HelloWorldServer

Nachdem die Vorgehensweise – durch das Objekt-Interaktions-Diagramm – verdeutlicht wurde, ist die Implementierung des HelloWorldServers kein großes Problem.

```
import HelloWorld.*;
// HelloWorldServer.java: The main program
public class HelloWorldServer{
    static public void main (String[] args)
    {
        try
        {
            // Initialize the ORB
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();

            // Initialize the BOA
            org.omg.CORBA.BOA boa = orb.BOA_init();

            // Create the HelloWorld Object
            MyInterfaceImpl worldObject = new
MyInterfaceImpl("MyWorld");

            // Export to the ORB
            boa.obj_is_ready(worldObject);

            // Ready to service requests - endless loop
            boa.impl_is_ready();
        }
        catch(org.omg.CORBA.SystemException e)
        {
```

```
            System.err.println(e);
        }
    }
}
```

Die ORB-Klasse ist Teil der VisiBroker Laufzeit-Umgebung. Die ORB.*init*-Methode ist eine Java Klassenmethode und muß somit nicht auf ein spezielles Objekt angewendet werden. Als Rückgabewert erhält man ein Objekt vom Typ ORB. Das bedeutet, daß man eine Referenz auf den VisiBroker ORB erhält und somit den Basic Object Adapter initialisieren kann. Die BOA_*init*-Methode liefert eine Objektreferenz auf den Basic Object Adapter zurück. Diese Referenz kann dazu benutzt werden, das neu erzeugte MyInterfaceImpl-Objekt beim Basic Object Adapter zu registrieren. Zuletzt muß noch die Bereitschaft des neuen Objekts über die Methode *impl_is_ready* dem BOA gemeldet werden.
Damit ist der Server „ready for business“. Nach dem Start geht der Server in eine Endlosschleife - ohne daß man sie programmieren mußte - und bearbeitet eintreffende Requests. Diese Funktionalität wird automatisch von CORBA und dem BOA erbracht.

9.5 Das andere Ende – der Client

Der Client besteht aus einer Klasse – HelloWorldClient. In dieser Klasse muß ebenso wie im Server eine Hauptfunktion vorhanden sein, welche folgende Aufgaben hat:

1. den Object Request Broker (ORB) initialisieren
2. das remote MyInterface-Objekt lokalisieren
3. die getTheMessage-Methode aufzurufen
4. das Ergebnis ausgeben

Der Ablauf soll wieder durch ein Objekt-Interaktions-Diagramm verdeutlicht werden.

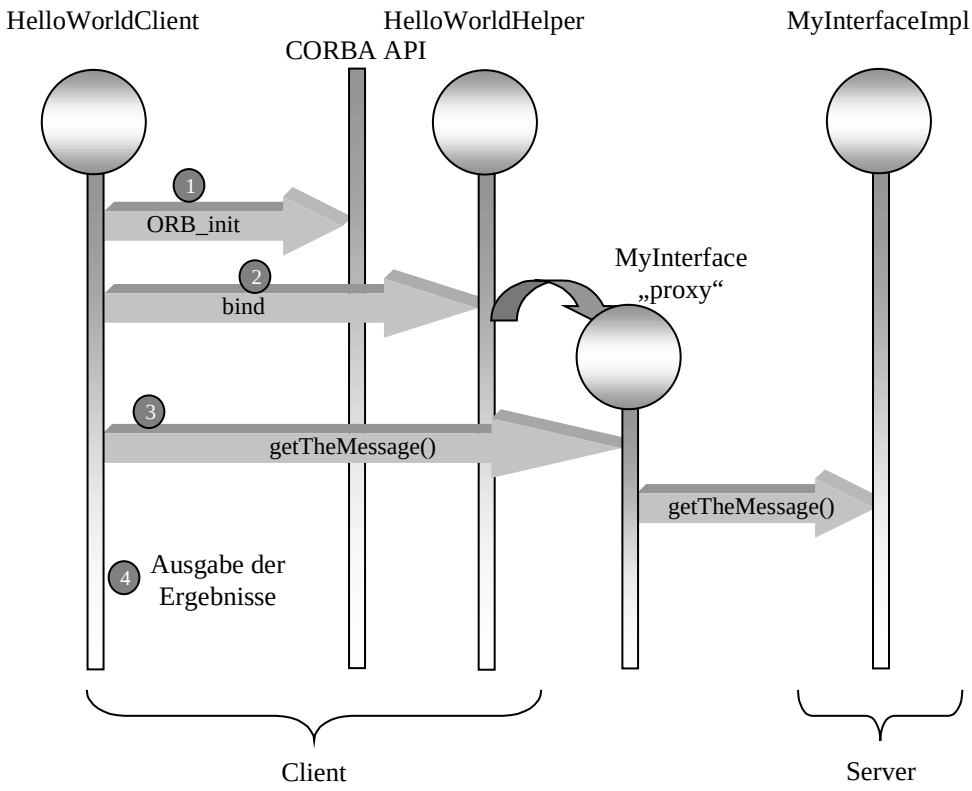


Abbildung 9-4: Objekt – Interaktions – Diagramm für den HelloWorldClient

Der Code des HelloWorldClient

```
// HelloWorldClient.java: The main program
public class HelloWorldClient {
    public static void main (String[] args)
    {
        try
        {
            // Initialize the ORB
            System.out.println("Initializing the ORB");
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();

            // Bind to the HelloWorld Object
            System.out.println("Binding to the HelloWorld Object");
            HelloWorld.MyInterface sayIt =

HelloWorld.MyInterfaceHelper.bind(orb,"MyWorld");

            // Invoke the getTheMessage() method
            System.out.println("Begin of Invocation");
            System.out.println(sayIt.getTheMessage());
            System.out.println("End of Invocation");
        }
        catch(org.omg.CORBA.SystemException e)
        {
            System.err.println("SystemException: " + e);
```

```
        }
    }
}
```

Wie beim HelloWorldServer muß als erstes der ORB initialisiert werden. Dazu wird wiederum die ORB.*init*-Methode aufgerufen, welche als Rückgabewert ein ORB-Objekt liefert. Damit erhält man wieder eine Referenz auf einen VisiBroker ORB.
Um jetzt die *getTheMessage()*-Methode aufzurufen, benötigt man zuerst eine Referenz auf ein MyInterface-Objekt. Das ist der Punkt, an dem die MyInterfaceHelper-Klasse mit der *bind*-Methode ins Spiel kommt. Durch die *bind*-Methode erhält man eine Referenz auf ein Proxy-Objekt, über welches man die Methoden aufrufen kann.
Für den Client unterscheidet sich ein statischer Methodenaufruf für ein entferntes Objekt nicht von einem Aufruf eines lokalen Objekts.

Glossar

Aktivierungsmodus (Activation Mode)
Der **Aktivierungsmodus** beschreibt die Art, wie ein **CORBA-Server** gestartet wird. Es wird unterschieden zwischen folgenden **CORBA**-Modi: **Shared Server**, **Unshared Server**, **Server-per-method**, **Persistent Server**.

Application Interfaces
Teil der **OMA**. Dies sind konkrete Produkte, wie sie vom End-User benutzt werden.

Attribut
Eine bekannte Eigenschaft einer **Klasse**, die einen Datenwert beschreibt, den jedes Objekt der **Klasse** besitzt. Die Gesamtheit aller **Attribut**e bezeichnet man als Objektzustand.

Basic Object Adapter (BOA)
Der **Basic Object Adapter** ist die Schnittstelle des **ORB**s für Objektimplementierungen.

Client
Programm, das **Operation**en eines entfernten Objektes aufruft. Allgemeiner: Programm, welches **Dienst**e eines **Servers** nutzt.

Common Facilities (Horizontal Common Facilities)
Teil der **OMA**. Dies ist eine Sammlung von Funktionalitäten, die inhaltlich, im Vergleich zu den **Object Services**, näher an der Applikationsschicht liegen. Sie haben einen horizontalen Charakter und können deshalb von den verschiedensten Applikationen verwendet werden.

CORBA
Common Object Request Broker Architecture. Standardisierung eines **ORB**s durch die **OMG**.

Distributed Component Object Model (DCOM)
Microsofts **Middleware** für verteilte Anwendungen. **DCOM** entstand aus COM, das zuvor innerhalb eines Rechners Microsofts Objektmodell umgesetzt hat. Microsoft hat mittlerweile die Weiterentwicklung von **DCOM** an die **Open Group** abgetreten.

Deferred Synchronous Mode
Dienstanfrage, bei der ein **Client** nicht auf die Beendigung einer von ihm aufgerufenen **Methode** wartet, bevor er seinen Programmablauf fortsetzt.

Dienst
Ein **Dienst** beschreibt eine **Operation**, die ein **Server** ausführen kann.

Dienstanfrage (Request)
Nachricht von einem **Client** an einen **Server**, einen bestimmten **Dienst** zu leisten. Der zu einer **Dienstanfrage** gehörende **Dienst** ist in **CORBA** eine **Operation**.

Domain Interfaces (Vertical Market Interfaces) Teil der **OMA**. Im Gegensatz zu den **Common Facilities** haben diese Schnittstellen vertikalen Charakter, was bedeutet, daß diese branchen-spezifischen **Dienst**e jeweils nur Teile der horizontalen Schichten benutzen.

Dynamic Service Activation Eine Aufgabe des **ORB**s ist die dynamische Aktivierung einer Objektimplementation, das heißt lokalisieren, instanziieren und zerstören eines Serverobjekts.

Exception Konzept einer Fehlerbehandlung durch den Programmierer mit dem Vorteil, daß nicht nur Fehlercodes, sondern auch selbst definierbare Typen als Hinweis zurückgegeben werden können. **Exception**s werden in C++ und Java vollständig unterstützt.

Implementation Repository (IR) In einem **Implementation Repository** werden Informationen abgelegt, die nötig sind, um die Implementierungen einer mit Hilfe der **IDL** definierten Schnittstelle zu starten. Dies kann der Pfad und Name eines **Server**s und dessen **Aktivierungsmodus** sein.

Instanz Eine **Instanz** ist ein Exemplar einer Klasse.

Interface **CORBA** bezeichnet die Gesamtheit aller exportierten **Attribut**e und **Methoden** eines Objekts als **Interface**. Definiert werden diese **Interface**s mit Hilfe der **Interface Definition Language** (**IDL**). Elemente eines Objektes, die nicht im **Interface** spezifiziert wurden, sind für **Client**s nicht zugreifbar.

Interface Definition Language (IDL) Die **Interface Definition Language** ist eine rein beschreibende Sprache. Sie ist in ihrer Syntax stark an C++ angelehnt. Sie ermöglicht die Unabhängikeit von **CORBA** bezüglich Plattform, Betriebssystem und Programmiersprache, da sie die Schnittstelle von der Implementierung trennt.

Internet Inter-ORB Protocol (IIOP) Ein im **CORBA**-Standard festgeschriebenes Protokoll, das die Interoperabilität zwischen **ORB**s unterschiedlicher Hersteller garantiert.

Klasse Definition aller **Attribut**e und **Methode**n eines Objektes. Die konkreten Objekte nennt man **Instanz**en einer **Klasse**.

Marshalling Sobald eine Kommunikation zwischen Objekten über Rechnergrenzen hinweg stattfindet, müssen die Parameter des Methodenaufrufs über das Netz geschickt

	werden. **Marshalling** bezeichnet dabei den Vorgang des Ein- oder Auspackens der Parameter in einem Netzpaket.
Methode	Realisierung bzw. Implementierung einer **Operation**.
Middleware	Eine Verteilungsplattform, die die Interoperabilität zwischen Komponenten einer verteilten Anwendung über Rechnergrenzen hinweg ermöglicht. Bekannte **Middleware**-Konzepte sind **CORBA**, Remote Method Invocation (RMI), Remote Procedure Call (RPC), etc.
Object Management Architecture (OMA)	Die **Object Management Architecture** ist eine von der **OMG** eingeführte Referenzarchitektur für verteilte objektorientierte Systeme.
Object Management Group (OMG)	Die **Object Management Group** ist ein internationales Konsortium von über 700 Mitgliedern aus Industrie, Wirtschaft und Forschungseinrichtungen. Ziel der **OMG** ist die Spezifikation einer **Middleware**-Plattform für offene verteilte Umgebungen.
Object Request Broker (ORB)	Der **Object Request Broker** ist die zentrale Komponente des **CORBA**-Standards, der für die Kommunikation zwischen Objekten verantwortlich ist.
Object Services	Teil der **OMA**. Dies sind **Dienst**e, welche die Nutzung von Objekten in einer verteilten Umgebung unterstützen, aber nicht durch **CORBA** abgedeckt werden.
One-way-request	Sprachkonstrukt der **IDL**. Ermöglicht Funktionsaufrufe, die kein Ergebnis erwarten Es erfolgt daher keine Synchronisation mit dem **Server**.
Open Group	Ein internationales Konsortium von Anwendern und Herstellern, das als Dachorganisation der Open Software Foundation (OSF) und X/Open agiert.
Operation	Ein aufrufbarer **Dienst**. Wird in objektorientierten Programmiersprache wie C++ und Java **Methode** genannt.
Persistent Server	Ist dieser **Aktivierungsmodus** aktiv, dann wird der **Server** nicht vom **BOA**, sondern von außerhalb (Systemadministrator) gestartet. Der Server meldet seine Bereitschaft dem **BOA**. Dieser behandelt alle **Requests** für diesen Server wie einen **Shared Server**.
Response	Antwort eines **Servers** auf den **Request** eines **Clients**.

Server Entferntes Objekt, daß eine definierte Menge von **Operation**en beantwortet. Allgemeiner: Programm, welches **Dienst**e für **Client**s anbietet.

Server-per-method Ist dieser **Aktivierungsmodus** aktiv, dann wird bei jedem **Request** ein neuer **Server** gestartet. Die Lebensdauer des **Server**s ist allerdings nur so lang wie die Dauer des **Methode**naufrufes.

Shared Server Ist dieser **Aktivierungsmodus** aktiv, dann wird der **Server** vom **BOA** ein einziges Mal gestartet. Alle ankommenden **requests**, welche seine **Dienst**e beanspruchen, werden dann von diesem **Server** bearbeitet. Behandelt der **Server** einen **Request**, dann müssen alle anderen **Request**s warten.

Signatur Die **Signatur** eines Objektes ist die vollständige formale Beschreibung aller **Attribut**e und **Methode**n mit deren Typisierungen. Die **Signatur** einer **Operation** ist die vollständige Typisierung aller Ein- und Ausgabeparameter.

Skeleton Vom **IDL**-Compiler generierter Programmcode, der das **Marshalling** bei einem **Methode**naufruf auf **Server-**Seite übernimmt. Der **Skeleton** verhält sich gegenüber dem **Server** wie der **Client.**

Stub Vom **IDL**-Compiler generierter Programmcode, der das **Marshalling** bei einem **Methode**naufruf auf **Client-**Seite übernimmt. Der **Stub** verhält sich gegenüber dem **Client** wie der **Server.**

Synchronous Mode **Dienstanfrage**, bei der ein **Client** auf die Beendigung einer von ihm aufgerufenen **Methode** wartet, bevor er seinen Programmablauf fortsetzt.

Universal Unique Identifier (UUID) **DCOM** versieht alle Schnittstellen mit einer globalen eindeutigen Identifikation, um Namens-konflikte zu vermeiden. Das Hilfsprogramm uuidgen erzeugt mit jedem Aufruf eine garantiert (weltweit) eindeutige Identifikation.

Unshared Server Ist dieser **Aktivierungsmodus** aktiv, dann wird für jedes Objekt ein eigener **Server**-Prozeß gestartet.

WEB-Fähigkeit Eigenschaft von Programmen, welche in WEB-Browsern übers Internet/Intranet ausführbar sind.

Literaturverzeichnis

[DST95] *HP Distributed Smalltalk, Unleashing the Power of Objects for Distributed Computing, Schulungsunterlagen der Firma Hewlett-Packard*, 1995

[Lod97] Lodderstedt T., Müller S., *Ein Brückenschlag: Die Standards OLE und CORBA in Interaktion,* UNIX open, UNIX open Verlagsgesellschaft mbH, Heft 2/97

[Neu97] Neumann L., *CORBA, Eine Einführung in Architektur, Funktionalität und Programmierung,* Schulungsunterlagen des Zentrums für Graphische Datenverarbeitung (ZGDV), 1997

[Orf97] Orfali R., Harkey D., *Client/Server Programming with JAVA and CORBA,* John Wiley & Sons, 1997

[Orf98] Orfali R., Harkey D., Edwards J., *Instant CORBA: Führung durch die CORBA-Welt,* Addison-Wesley (Deutschland) GmbH, 1998

[Ort97] Ortak M., *Die Kooperation von Objekten im Internet,* OBJEKTspektrum, Sigs Publications, Heft 5/97

[Pud97] Puder A., *Objektiv betrachtet, Verteilte Objekte:DCOM versus CORBA,* ix, Heise-Verlag, Heft 8/97

[Red96] Redlich J.-P., *CORBA 2.0: Praktische Einführung für C++ und Java,* Addison-Wesley (Deutschland) GmbH, 1996

[Say97] Sayegh A., *CORBA: Standard, Spezifikationen, Entwicklung,* O'Reilly-Verlag, 1997

[Sch97] Schwarz M., *'Moment, ich verbinde ...',* c't, Heise-Verlag, Heft 3/97

[Sie96] Siegel J., *CORBA - Fundamentals and Programming,* John Wiley & Sons, 1996

[Stal97] Stal M., *World Wide CORBA - verteilte Objekte im Netz,* OBJEKTspektrum, Sigs Publications, Heft 6/97