In diesem Kapitel:

- *CORBA Architektur, Basismechanismen, Dienste*
- *Beispiele mit dem Portable Object Adapter (POA)*
  - Einfaches Beispiel
  - POA mit persistentem Server
  - POA mit transientem Server
  - POA mit dem Tie / Delegations Modell
  - ImplBase (Vererbung) transienter Server
- *Verteilung auf Client und Server*
- *Neue Features*
  - POA - Der Portable Object Adapter
  - Portable Interceptors
  - Das GIOP - General Inter ORB Protokoll
  - CORBA Spezifikation versus J2SE
  - RMI over IIOP und POA

CORBA - mit dem Portable Object Adapter (POA)

1.1. CORBA - Architektur, Basismechanismen, Dienste

1.1.1. Einführung

Die fortschreitende Vernetzung und der verstärkte Einsatz von objektorientierten Technologien gehören mit zu den bedeutenden Veränderungen der Informationstechnologie der letzten Jahre. Gleichzeitig wandelt sich die Art der Informationsverarbeitung und das Anforderungsprofil an IT-Lösungen[1] . Die Abkehr vom zentralistischen Ansatz der Datenverarbeitung hin zu einem dezentralen Ansatz verlangt nach neuen Konzepten, Architekturen und Technologien.

Die Art der Informationsverarbeitung entwickelte sich vom manuellen Ansatz über die Batch-Verarbeitung, die Online-Verarbeitung und den Personalcomputer zur kooperativen Informationsverarbeitung. Die Komplexität der dabei zugrunde liegenden Verarbeitungsmodelle stieg während dieser Evolution stark an. Zur Bewältigung der gestiegenen Komplexität wurden neue Konzepte und Technologien für Analyse, Design und Implementierung von Softwaresystemen entwickelt. Die Evolution der Implementierungs-Sprachen führte von der Assember-Programmierung über die prozeduralen Programmiersprachen zu den objektorientierten Programmiersprachen. Der bei jedem dieser Evolutionsschritte gewonnene Abstraktionsgrad ist einer der wesentlichen Aspekte, durch den die Komplexitätssteigerung der zugrunde liegenden Modelle handhabbar wird.

Der Kerngedanke des objektorientierten Ansatzes besteht in der Kapselung von Daten (Attributen) und Operationen zu einer Einheit, die ein nach aussen sichtbares Verhalten (Interface) hat, ihren internen Aufbau im Sinne des "Information Hiding" aber verbirgt. Das Interface besteht aus dem Objektnamen und der Menge der aufrufbaren Operationen. Eine Operation besteht aus zwei Teilen, einer Signatur und der Implementierung. Eine Signatur setzt sich aus dem Operationsnamen, dem Parameternamen und Parametertypen sowie dem Rückgabewert und den möglichen Exceptions zusammen. Die durch das Interface beschriebenen Operationen sind die von einem Objekt angebotenen Dienste. Ein Client kann diese Dienste in Anspruch nehmen und dabei das im Interface beschriebene Verhalten des

[1] Die Veränderungen beeinflussen und bedingen sich dabei gegenseitig in synergetischer Weise

© J.M.Joller

Objektes annehmen.[2] Das Verhalten des Objekts ist durch die öffentlich zugesicherten Eigenschaften des Objekts gegeben. Wird eine Methoden des Objekts mit Parametern aufgerufen die der Signatur entsprechen, besteht das Resultat aus einem Rückgabewerte des vereinbarten Typs oder einer Exception. Objekte, die der gleichen Klassifikation bezüglich ihres Verhaltens genügen, werden in Klassen zusammengefasst. Die Klasse eines Objektes ist im umgekehrten Falle die Beschreibung des charakteristischen Verhaltens des Objektes.

Die Abkehr von der zentralistischen Datenverarbeitung hin zu kooperativen Strukturen verteilter Systeme ist ein Trend in der Informationstechnologie. Die notwendigen Dienste werden nicht mehr von einer zentralen Stelle bereitgestellt, sondern von verschiedenen Systemen. Eine solche Struktur aus Diensterbringen und Dienstnehmern auf nicht notwendigerweise unterschiedlichen Systemen wird als Client/Server-Architektur bezeichnet. Die Vorteile einer solchen Architektur liegen in unterschiedlichen Teilaspekten der Informationsverarbeitung.

- **Lastverteilung**: Die Last der Anfragen muss nicht mehr von einem System getragen werden, sondern verteilt sich auf mehrere Systeme. Hierdurch kann die Antwortzeit auf eine Anfrage verkürzt werden, die maximale Anzahl der verarbeitbaren Anfragen steigt.

- **Skalierbarkeit**: Auf wachsende Anforderungen kann flexibel reagiert werden, indem die Anzahl der Systeme, die einen bestimmten Dienst erbringen, vergrössert wird.

- **Ausfallsicherheit**: Im Fall der zentralen Datenverarbeitung führt der Ausfall des zentralen Systems zum Ausfall des Gesammtsystems. Die Verteilung der Dienste auf mehrere Systeme kann eingesetzt werden, um eine erhöhte Ausfallsicherheit zu erreichen.

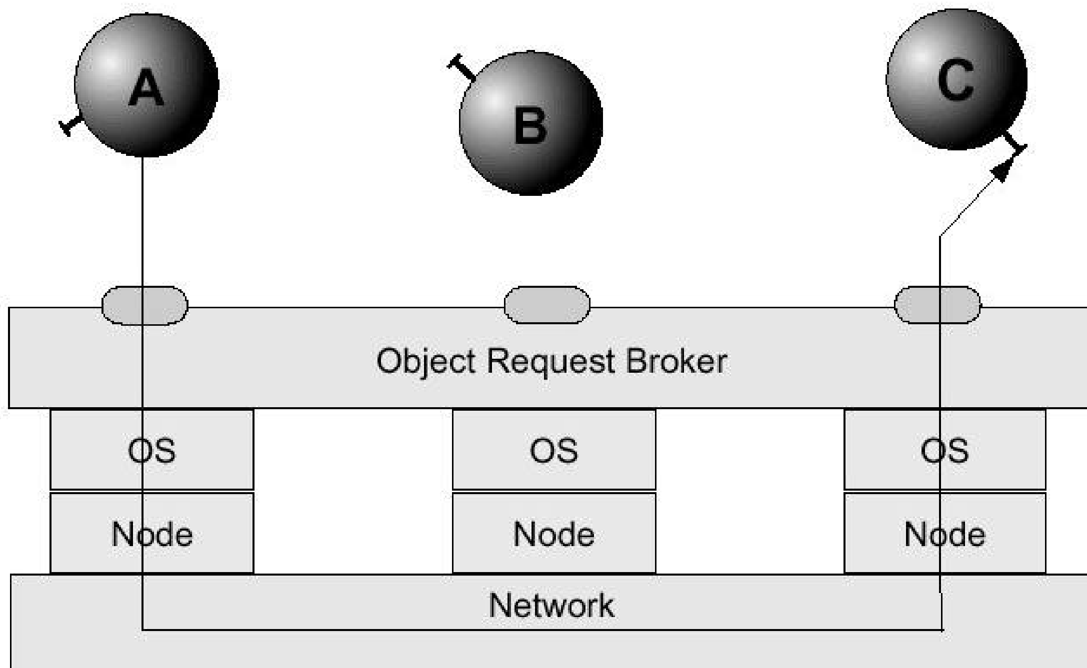


Abbildung 1Der Object Request Broker als Middleware für ein System verteilter Objekte.

[2] Für Beschreibung des Verhaltes werden in manchen objektorientierten Programmiersprachen noch Preconditions, Postconditions und Invarianten angegeben. [Gries 1981]

© J.M.Joller

Es existieren unterschiedliche Ansätze die zwei Paradigmen Objektorientierung und Client/Server zu verschmelzen. Ein gemeinsames Ziel vieler Architekturen für verteilte objektorientierte Systeme ist die Realisierung von Verteilungstransparenz [3] . Der Begriff Verteilungstranzparenz untergliedert sich im wesentlichen in folgenden Teilaspekte:

- *Ortstransparenz*: Ein Client besitzt keine Informationen über die physikalischen Ort eines von ihm benutzten Server-Objekts.
- *Aufruftransparenz*: Ein Client ist nicht von dem Pfad zu einem Server-Objekt abhängig. Das Art der Kommunikation und die verwendeten Netzwerkprotokolle sind für den Client transparent.
- *Relokationsransparenz*: Die Migration eines Server-Objekts zwischen Lokationen ist für den Client transparent. Dies umschliesst die sowohl die Tatsache der Migration als auch die betroffenen physikalischen Orte.
- *Repräsentationstransparenz* : Die interne Repräsentation eines Objektes ist für Clients nicht sichtbar. Verändert sich die interne Repräsentation eines Objekts ist dies für die das Objekt benutzende Clients transparent.
- *Maschinentyp-, Betriebssystem-, Programmiersprachentransparenz*: Clients sind von Betriebssystem, Maschinentyp und Implementierungssprache der Objekte unabhängig.

Diese Dienste werden von einer Instanz bereitgestellt, die allgemein als Object Request Broker (ORB) bezeichnet wird. Aus Sicht der Applikationen stellt er eine Plattform für verteilte Objekte dar. Ein solcher ORB kann auf einem nicht objektorientierten Betriebssystem laufen. Der ORB abstrahiert dabei einige Betriebssystemspezifika des zugrundeliegenden Betriebssystems. In diesem Sinne kapselt er das verwendete Betriebssystem ein und bietet den ihn benutzenden Objekten eine einheitliche Menge von Diensten plattformübergreifend an. Bei der Implementierung von Server-Objekten wird aber auf Systemressourcen über die Betriebssystemschnittstelle zugegriffen. Objekte sind somit nicht implizit plattformunabhängig. [Pope 1997]

[3] Die Bedeutung des deutschen Wortes Transparenz und des englischen Wortes Transparency unterscheiden sich. Während Transparenz im deutschen Sprachraum ein Durchscheinen meint, ist unter dem englischen Begriffs Trannsparency ein `nicht sichtbar werden lassen' im Sinne von Durchsichtigkeit zu verstehen.

© J.M.Joller

1.1.2. Die Object Management Group (OMG)

Die OMG ist ein Konsortium von heute (2001) mehr als 800 Mitgliedern, deren Ziele die Spezifikation einer Architektur für verteilte, objektorientierte Systemen ist. Die Mitglieder der OMG stammen aus allen Bereichen der Informationstechnologie. Neben vielen Anbieter und Entwicklern von Systemen befinden sich auch viele Forschungsinstitutionen und reine Anwender objektorientierter Technologien unter den Mitgliedern.

Die objektorientierte Broker Architektur der OMG ist die Common Object Request Broker Architektur (CORBA). Ein Ziel der OMG ist Unterstützung und Verbreitung des Einsatzes objektorientierter Technologien gemäss der von ihr spezifizierten Architektur (CORBA). Die OMG tritt nach aussen als Marketing-Organisation auf um die Verbreitung der von ihr spezifizierte Architektur zu fördern. Andererseits sieht sich die OMG auch in der Rolle eines Technologiekonsortiums, das Technologien für verteilte objektorientierte Systeme spezifiziert.

Die OMG ist kein Gremium zur Standardisierung und kann somit nicht mit Institutionen wie dem "American National Standards Institute" (ANSI) oder der "International Standards Organisation" (ISO) verglichen werden. Die OMG versteht sich als Organisation, die den Mitgliedern die Diskussion kontroverser Ansichten, den Austausch von Ideen, die Suche nach einem Konsens und einer Übereinkunft bezüglich der Verwendung bestimmter Architekturen und Technologien erlaubt. Der Arbeitsansatz der OMG ist stark konsensorientiert. Werden neue Technologien in den unterschiedlichen "Task Forces" diskutiert, hat jedes Mitglied [4] die Möglichkeit Vorschläge einzubringen. Ein solcher Vorschlag wird von der OMG nur dann akzeptiert, wenn eine funktionsfähige Implementierung vorliegt. [5][Pope 1997]
Aus dieser kurzen Beschreibung der Arbeitsweise der OMG ist ersichtlich, dass eine Einigung häufig nur auf kleinstem gemeinsamen Nenner möglich ist und die Reaktionszeit der OMG auf aktuelle Anforderungen des Marktes eher träge ist. Die starke Dynamik, die die ständig fortschreitende Adaption von Technologie und der Wandel der zugunde liegenden Architektur mit sich bringen, lässt Stimmen laut werden, die eben diese Dynamik als Anlass zur Kritik an der OMG respektive von CORBA nehmen.

1.1.3. Die Object Management Architecture (OMA)

Die OMA stellt die Grundlage aller weiteren Spezifikationen der OMG dar. Die wesentlichen Aufgaben der OMA bestehen in der Definininition einer einheitlichen Terminologie, der Partitionierung des komplexen Problemraums und die Definition eines Objektmodells. Missverständnisse aufgrund unterschiedlicher Bezeichner und deren Bedeutung werden hierdurch weitgehend vermieden. Die OMA beschreibt die grundlegenden Begriffe und deren Bedeutung auf einem sehr hohen Abstraktionsniveau. Das abstrakte Modell der OMA wird durch CORBA konkretisiert. In diesem Zusammenhang kann die OMA als Typ verstanden werden und CORBA als eine Instanz dieses Typs. Die OMA ist die Referenzarchitektur der OMG. [6][Orfali et al. 1997]

---

[4] Auch Nichtmitglieder können Technologievorschläge einbringen.
[5] Auf eine Anfrage der OMG (Request for Proposal RFP) werden Vorschläge eingereicht. Existieren mehrere konkurrierende Vorschläge werden diese häufig miteinander verschmolzen. Ein eingereichter Vorschlag wird nur akzeptiert, falls eine funktionsfähige Implementierung vorliegt. Eine Zusammenführung mehrerer Vorschläge zu einem neuen wird dagegen auch akzeptiert, wenn keine Referenzimplementierung vorliegt.
[6] Eine klare, präzise Definition des Objektmodells der OMG findet sich in den sogenannten CORE Models. Der Spezifikation von CORBA 2.0 liegt das Objektmodell CORE-92 zugrunde. [OMG 1995].

© J.M.Joller

Wesentliche Aspekte der transparenten Verteilung von Objekten sind Ortstransparenz, Plattformtransparenz und Sprachheterogenität. Die ortstransparente Verteilung von Objekten erlaubt die Interaktion von Objekten über Systemgrenzen hinweg, ohne dass dies für die Objekte ersichtlich wäre. Durch Plattformtransparenz und Programmiersprachheterogenität wird von den verwendeten Implementierungssprachen und den zugrundeliegenden Betriebssystemen abstrahiert. Eine Infrastruktur, die die Entwicklung und den Betrieb von verteilten, objektorientierten Anwendungen erlaubt, wird als objektorientierte Middleware bezeichnet. Eine solche Infrastruktur kann nicht auf die unterste Ebene des Methodenaufrufs beschränkt bleiben. Lösungen für Probleme auf höherer Ebene müssen integraler Bestandteil einer solchen Infrastruktur sein.

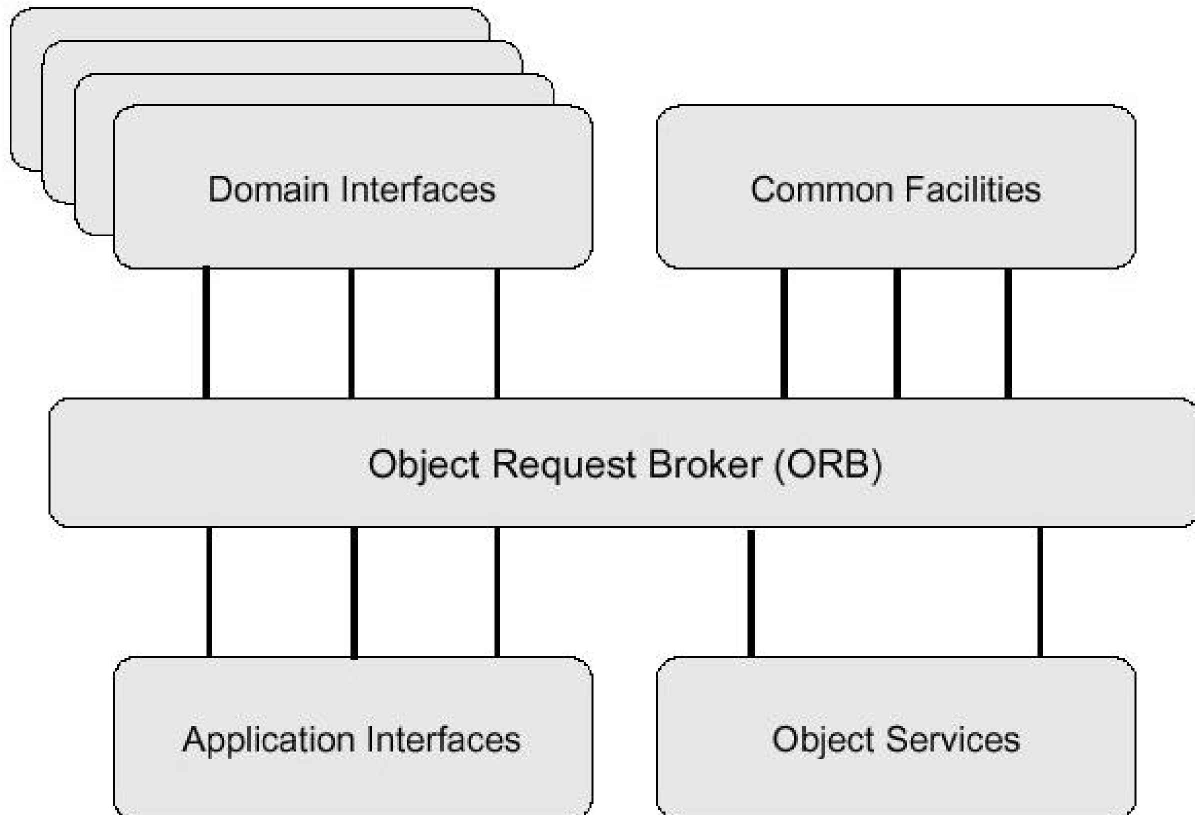


Abbildung 2Die Object Management Archiecture der OMG.

Zentraler Bestandteil der Object Management Architecture ist der *Object Request Broker* (ORB). Wesentliche Aufgaben des ORB bestehen darin, Methodenaufrufe (Request) von Client an Server-Objekte weiterzuvermitteln, sowie die Resultate und Fehlerbedingungen (Response) dieser Requests an den aufrufenden Client zurückzuliefern. Dieser ORB ist das Kernstück der gesamten verteilten Objektarchitektur. Die Veröffentlichung seiner Spezifikation erfolgte sehr früh und ist unter dem Begriff Common Object Request Broker Architecture (CORBA) bekannt. [Puder und Römer 1997], [OMG 1992]

Der ORB stellt die Verteilungskomponente dar. Die Entwicklung und Implementierung grosser, objektorientierter Softwareprojekte verlangt jedoch nach weiteren Diensten, die Lösungen für ein breites Spektrum immer wiederkehrender Problemstellungen bieten. (z.B. die transaktionsorientierte Abarbeitung von Methodenaufrufen, das persistente Abspeichern von Objekten, die Verfügbarkeit von Sicherheitsmechanismen und der Zugriff auf verteilte

© J.M.Joller

Objekte über einen Namensdienst).Diese fundamentalen Dienste sind im Rahmen der Common Object Services Specification (COSS) definiert.[OMG 1997b]
Die Praxis verlangt weitergehende Dienste als dies durch die COSS Services gegeben wäre. Infrastrukturelle Dienste werden durch die sogenannten Common Facilities spezifiziert. Als ein Beispiel für die domänenunabhängigen, infrastrukturellen Dienste der Common Facilities sei die Administration vernetzter CORBA-Systeme genannt. Dieses Beispiel erlangt besondere Bedeutung, berücksichtigt man, dass die OMG kein Produkt sondern Spezifikationen zur Verfügung stellt und die Produkte von unterschiedlichen Firmen und Institutionen stammen können. Die sich daraus ergebenden Unterschiede in den jeweiligen Implementierungen der Hersteller sind ein erwünschter Effekt. Eine Implementierung, die Lösungen für spezielle Problemstellungen bietet, kann für den allgemeinen Einsatz zu spezialisiert sein. Die OMG erlaubt es Anbietern, unter der Voraussetzung, dass die Konformität zur Spezifikation erhalten bleibt, proprietäre Konzepte in ihren Implementierungen zu integrieren.

Frameworks, die Dienste für bestimmte Anwendungsbereiche definieren, werden in den Domain Interfaces spezifiziert. Beispiele für solche Domain Interfaces sind spezielle Frameworks für die Bereiche Gesundheitswesen, Transport, Telekommunikation oder Finanzdienstleistungen. Die Domain Interfaces bieten Lösungen für immer wiederkehrende Problemstellungen aus den jeweiligen Bereichen, sind aber zu spezifisch um sie den Common Facilities zuzuordnen.

1.1.4. Die CORBA-Architektur

Die Common Object Request Broker Architektur, stellt das Kernstück der Architektur dar. Zunächst soll kurz beschrieben werden, wie das Ziel, Aufruf- und Ortstransparenz zu erreichen, realisiert worden ist. Die Grundidee besteht darin, Abstraktionsschichten zwischen Anwendung und zugrundeliegender Plattform einzuziehen. Der ORB übernimmt die Vermittlung zwischen Client-Applikation und Zielobjekt. Er transportiert die Requests der Clients zu den Server-Objekten und übermittelt im Gegenzug eventuell vorliegende Resultate oder Ausnahmebedingungen der Server-Objekte an die Clients. Ein Client muss den Ort des Server-Objektes nicht kennen, die Lokalisierung ist integraler Bestandteil eines Requests und wird durch den ORB vorgenommen.
Zu den Aufgaben der ORB Architektur gehört aber nicht nur der Transport von Aufrufen und deren Rückgabewerten, sondern auch das Führen eines Repositories der verfügbaren Server-Objekte. Ein Server-Objekt kann seine Verfügbarkeit explizit beim Broker anmelden, so dass dieser in die Lage versetzt wird die Anfragen durch Clients an das entsprechende Zielobjekt weiterzuleiten.
Client und Server sind nicht direkt mit dem Broker verbunden, sondern durch zwei weitere Indirektionsschichten getrennt. Auf Client-Seite ist dies ein Clientproxy [7], der die gleiche Schnittstelle implementiert wie das zugehörige entfernte Server-Objekt. Der Client nutzt das Server-Objekt nicht direkt, sondern den Clientproxy. Die Aufgabe des Clientproxy ist es, den Aufruf in ein plattformunabhängiges Format zu überführen, dem Broker diesen Aufruf zu übermitteln und auf das Eintreffen der Rückmeldung zu warten. Ein vorliegendes Resultat nimmt den umgekehrten Weg durch den Clientproxy. Die Konvertierung aus dem plattformunabhängigen in das spezifische Format wird durch den Clientproxy vorgenommen. Für den Client ist dieser Vorgang transparent. Die Clientproxies werden als Stubs bezeichnet. Auf Seite des Server-Objekts übernimmt ein Serverproxy eine ähnliche Funktion, indem

[7] Proxy := Stellvertreterkomponente

© J.M.Joller

dieser dem Server-Objekt gegenüber als Client auftritt. Die Serverproxies werden kurz Skeletons genannt. [Puder und Römer 1997]

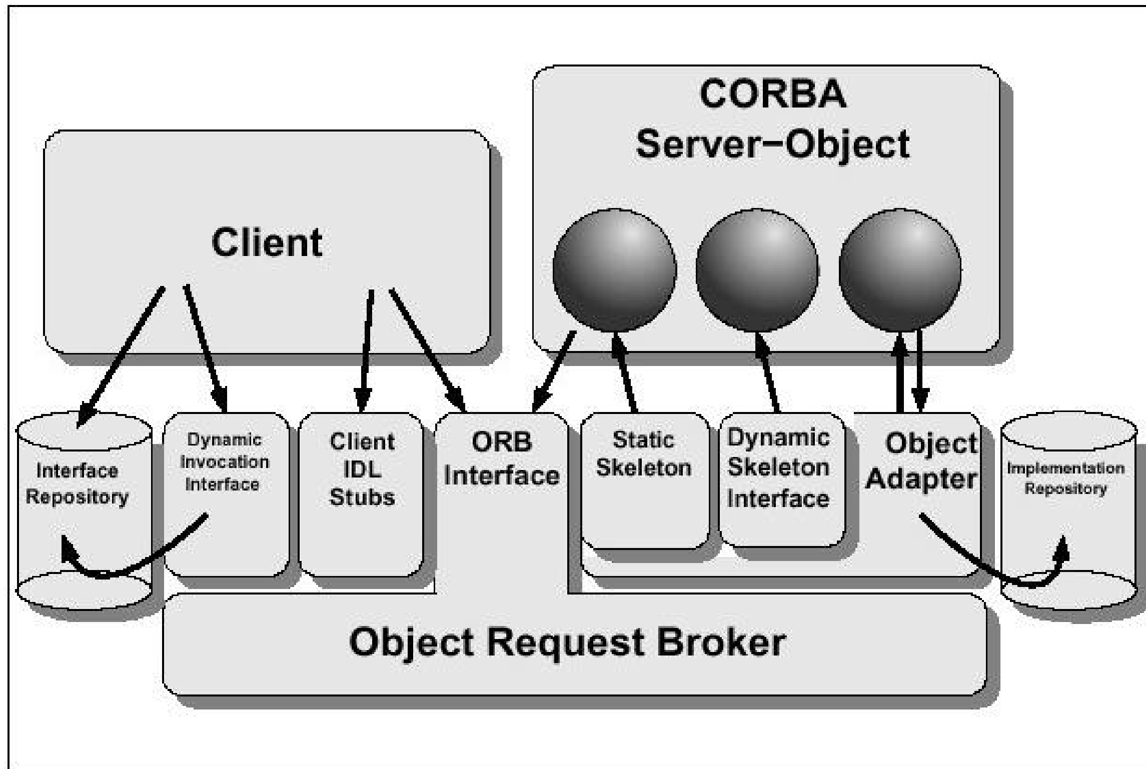


Abbildung 3Die CORBA-Architektur

Aus Gründen der Skalierbarkeit und Interoperabilität ist es wesentlich, dass unterschiedliche Broker Implementierungen miteinander kommunizieren können. Die OMG hat im Rahmen der CORBA 2.0 Spezifikation das Internet Inter-ORB Protocol (IIOP) definiert, das die Kooperation zwischen unterschiedlichen ORB-Implementierungen ermöglicht. [OMG 1998]

1.1.5. Die CORBA 2.0 Spezifikation

Der CORBA-Standard realisiert das durch die OMA beschriebene Modell der verteilten Objekte in heterogenen Umgebungen. Die CORBA-Namensgebung bezeichent die Client- bzw. Serverproxies als Stubs und Skeletons. In der bisherigen Ausführung, insbesondere der Betrachtung der OMA, wurde des öfteren der Begriff der Plattformtransparenz benutzt. Diese Transparenz kann sich nicht auf die Betriebssystemebene beschränken, sondern muss auch die Verwendung unterschiedlicher Programmiersprachen bei der Implementierung ermöglichen. Dies spiegelt sich im übergreifenden Objektmodell der OMG wieder. Jedes CORBA-Objekt wird als Instanz eines Interface-Typs gesehen, wobei diese Interface-Typen im klassischen Objektmodell einer Klasse entsprechen. [OMG 1997a]

Das aus der objektorienterten Welt bekannte Konzept der Spezialisierung/Generalisierung ist auch für diese Interface-Typen definiert, beschränkt sich aber auf die Spezialisierung von Interfaces. Die Vererbung von Implementierungen ist nicht definiert. Bei der Implementierung einer Unterklasse müssen somit auch alle Methoden der Elternklassen erneut implementiert werden.

© J.M.Joller

1.1.6. Die Interface Definition Language (IDL)

Auf Basis einer IDL-Spezifikation für ein Interface ist es so möglich Client und Service in unterschiedlich Sprachen zu realisieren. Dies wird erreicht, durch die Einführung einer deklarativen Sprache zur Beschreibung von Interfaces und der Definition einer Abbildungsvorschrift (Language Mapping) aus dieser Beschreibungssprache in existierende Programmiersprachen. Die Beschreibungssprache, die CORBA Interface Definition Language (CORBA-IDL), ist keine vollständige Programmiersprache. Es handelt sich um eine rein deklarative Sprache mit C++ ähnlicher Syntax, die nur Elemente zur Schnittstellendeklaration enthält. Aus einer Schnittstellenbeschreibung generiert ein IDL-Compiler das Server-Skeleton und den Client-Stub in einer Programmiersprache, für die ein Language Mapping durch die OMG definiert ist. [OMG 1994]

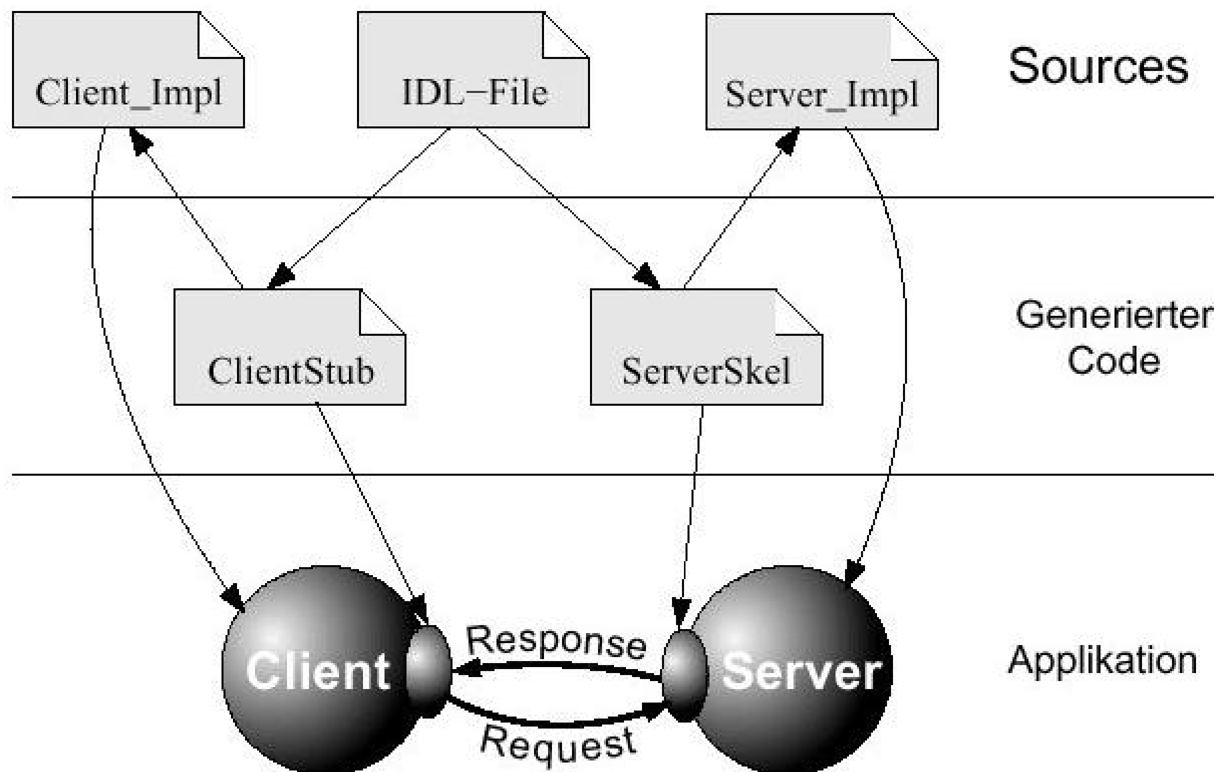


Abbildung 4Abhängigkeiten zwischen IDL-Beschreibung, Client-Stub, Server-Skeleton und den Implementierungen von Client und Server-Objekt.

© J.M.Joller

1.1.7. CORBA-Objektreferenzen

Aus Sicht eines Clients sind Server-Objekte abstrakt. Der Client verfügt nur über eine Objektreferenz des Server-Objektes und Informationen über das vom Server-Objekt implementierte Interface. Die einem Client vorliegenden Informationen zu den Interfaces sind auf den deklarativen Aspekt der Signaturen von Operationen und die Operationsnamen eingeschränkt. Für die Erzeugung eines Requests benötigt ein Client eine Objektreferenz eines Server-Objektes, den Namen der Operation und die Signatur der aufzurufenden Operation. Die Objektreferenzen sind implementierungsspezifisch und deshalb nur innerhalb einer ORB-Implementierung eindeutig. *Um Objektreferenzen über ORB-Grenzen hinweg austauschen zu können, definierte die OMG die Interoperable Objekt Referenzen (IOR). Diese sind global eindeutig.*

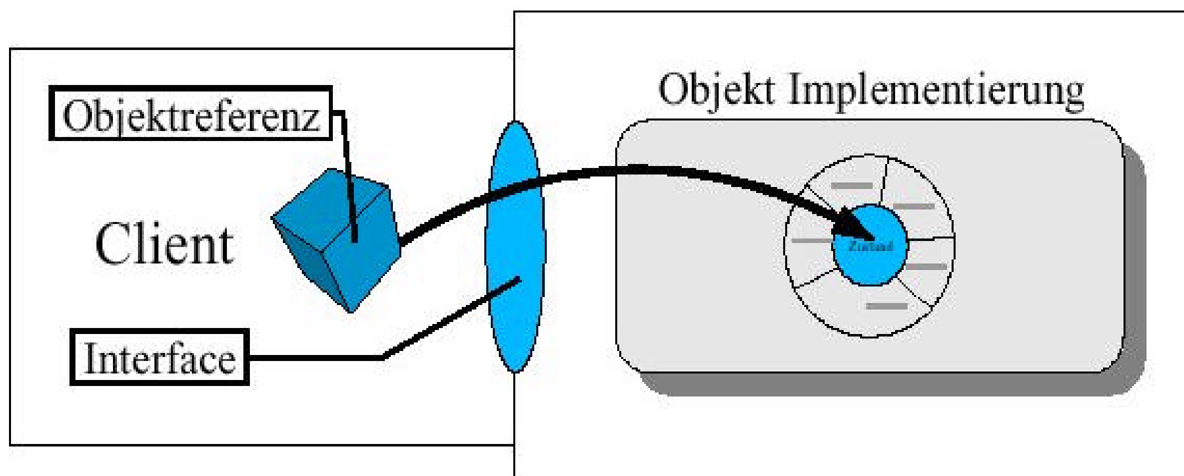


Abbildung 5Ein CORBA-Client benötigt eine Objektreferenz um Requests an ein Objekt zu generieren.

1.1.8. Statisches und dynamisches CORBA-Objektmodell

Im bisher beschriebenen CORBA-Modell müssen die Clients bereits zur Compile-Zeit Kenntnisse über das Interface der von ihnen benutzten CORBA-Objekte haben. Für die meisten Anwendungsbereiche ist dieses Modell ausreichend. Es gibt Anwendungsfälle bei denen die Notwendigkeit besteht, zur Laufzeit Objekte referenzieren zu müssen, deren Signatur zur Compile-Zeit nicht bekannt war[8]. CORBA bietet mit dem Dynamic Invocation Interface (DII) zusammen mit dem Interface Repository (IR) ein Konzept, dieses zu realisieren. Das Interface Repository enthält Beschreibungen aller im System registrierten CORBA-Objekte, während das Dynamic Invocation Interface, mit Hilfe der durch das Interface Repository bereitgestellten Typinformationen, das Erzeugen und Absenden von Requests an beliebige Server-Objekte ermöglicht. Für die Server-Objekte ist dieser Vorgang transparent. Aus Sicht der Server-Objekte ist nicht unterscheidbar, ob ein Request auf Seite des Clients per Static Invocation Interface (SII) oder per Dynamic Invocation Interface (DII) erzeugt wurde.
Auf Server-Seite steht das Pedant zum DII durch das Dynamic Skeleton Interface (DSI) zur Verfügung. Ein Server wird durch das DSI in die Lage versetzt, zur Laufzeit

[8] Soche Anwendungsfälle sind z.B. Debugger und Bridges. Asynchrone Requests sind unter CORBA 2.0 nur durch das DII realisierbar.

© J.M.Joller

Objektimplementierungen bereitzustellen, die zur Compile-Zeit nicht bekannt waren. Ein Client, der ein Server-Objekt benutzt, erlangt keinerlei Kenntnis darüber, ob der Server die Methoden per SSI oder DSI bereitstellt. Aus Sicht des Clients ist die Art und Weise der Diensterbringung des Servers transparent.
Bei der Implementierung von Clients oder Server-Objekten kann sowohl das statische als auch das dynamische Aufrufmodell zur Anwendung kommen. Den Vorteilen des Dynamic Invocation Interfaces und des Dynamic Skeleton Interfaces stehen eine ganze Reihe von Nachteilen gegenüber. Die Entscheidung für oder gegen den Einsatz von DII oder DSI wird in der Praxis meist zugunsten der statischen Interfaces ausfallen. Nur wenn die besonderen Fähigkeiten der dynamischen Interfaces benötigt werden kommen diese zum Einsatz. Die Vorteile der statischen Interfaces gegenüber den dynamischen Interfaces sind unter anderem:
Programmieraufwand geringer: Der Methodenaufruf ist einfach strukturiert. Zur Aktivierung werden nur eine Objektreferenz, der Name der Methode und Operationsparameter, entsprechend der Signatur der Operation, benötigt.

- Typ-Prüfung zur Compile-Zeit: Die Prüfung auf Typverträglichkeit wird zur Compile-Zeit vorgenommen.
- Höhere Performance: Benchmarks für existierende CORBA-Implementierungen haben gezeigt, dass der statische Methodenaufruf bezüglich der benötigten Ressourcen günstiger ist als ein entsprechender dynamischer Aufruf.

Der Quell-Code ist selbstdokumentierend: Die Bedeutung einer Code-Sequenz, die einen statischen Methodenaufruf darstellt ist, im Gegensatz zur Code-Sequenz eines dynamischen Aufrufs, leicht zu verstehen.

© J.M.Joller

1.1.9. CORBA Request-Typen und deren Semantik

Ein standard CORBA-Request folgt dem Request/Response-Paradigma . Ein Request, der von einem Client erzeugt wird, leitet der ORB an das Server-Objekt. Die Implementierung des Server-Objektes führt die durch den Request beschriebene Operation mit den übergebenen Parametern aus. Das Resultat oder eventuell vorliegende Exceptions werden durch den ORB an den aufrufenden Clients zurückgeliefert. Der Client wird für die Zeit zwischen Absenden des Requests und Empfang der Response blockiert. Die Semantik eines solchen Requests ist vom Typ at-most-once .

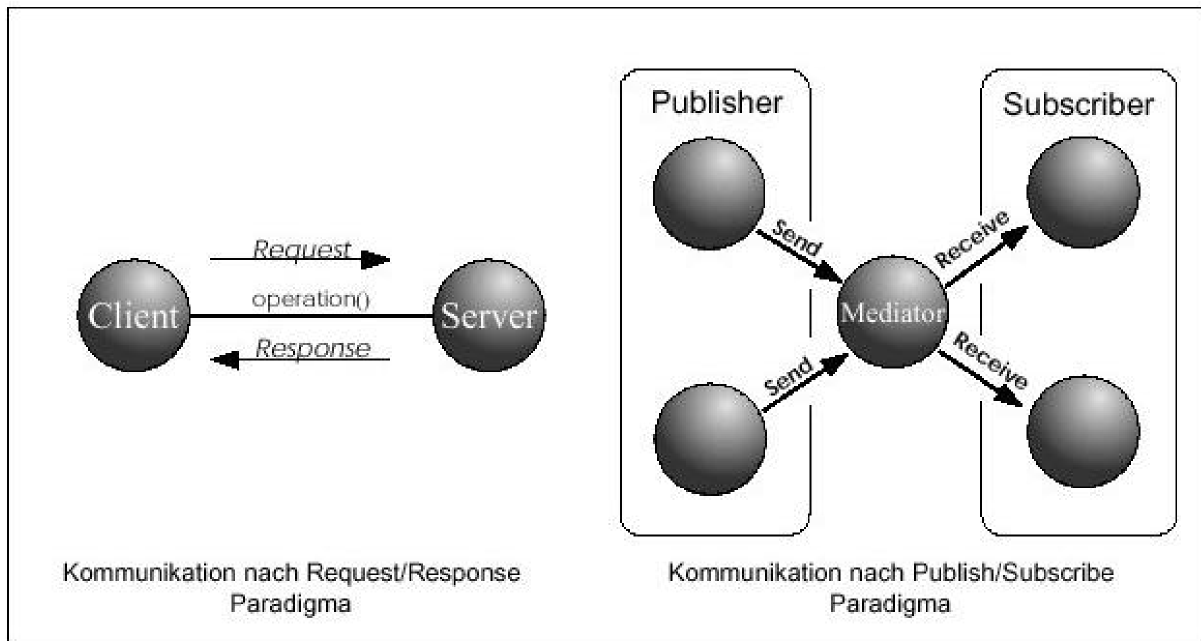


Abbildung 6Kommunikation nach dem Request/Response und Publish/Subscribe Paradigma

Oneway-Requests sind ein weiterer CORBA-Request-Typ. Ein solcher Request blockiert den Aufrufer nicht. Die Semantik eines oneway-Requests ist vom Typ best-effort . Ein Client, der eine oneway Operation eines Server-Objektes aufruft, bekommt keine Rückantwort. Oneway-Operationen werden bereits bei der IDL-Deklaration des Interfaces vereinbart.

Wird das Dynamic Invocation Interface zur Erzeugung des Requests benutzt steht ein weiterer Typ zur Verfügung: die verzögert-synchronen Requests . Der Client wird nach Absenden des Requests nicht blockiert. Der Status des Requests kann durch den Client vom ORB erfragt werden. Liegt die Rückantwort auf den Request vor, kann der Client diese vom ORB erfragen. Die Semantik eines solchen Aufrufs ist vom at-most-once Typ. Die Praxisrelevanz des auch als pseudo-asynchronen bezeichneten Aufrufs ist jedoch gering. In der Mehrzahl der Fälle werden standard CORBA-Requests vom Request/Response-Typ mit at-most-once Semantik eingesetzt werden.

© J.M.Joller

1.1.10. Der Basic Object Adapter (BOA)

Als Mittler zwischen dem ORB und den Server-Implementierungen dienen die sogenannten Object Adapter (OA). Sie implementieren eine Schnittstelle zu Methoden des ORB, die von Server-Implementierungen genutzt werden. Die Objektadapter sind eine Schicht zwischen dem ORB und den IDL Skeletons.

Zu den Aufgaben eines Object Adapters zählt die Aktivierung von Objekten. Ein CORBA-Objekt existiert innerhalb der Ablaufumgebung eines Prozesses. Die Aktivierung eines Objekts durch einen Objektadapter kann auf mehrere Arten geschehen: Ein neuer Prozess kann generiert werden, Ein Thread innerhalb eines existierenden Prozesses kann erzeugt werden oder ein existierender Thread oder Prozess kann benutzt werden.

Da spezielle Aufgabenstellungen spezielle Lösungen fordern, hat die OMG nicht einen Objektadapter definiert, sondern nur dessen Funktion allgemein beschrieben. Der CORBA 2.0 Standard fordert von allen CORBA konformen Implementierungen den sogenannten Basic Object Adapter (BOA). Dieser generische Adapter hat folgende Aufgaben:

- Generieren und interpretieren von Objektreferenzen
- Aktivierung und Deaktivierung von Objektimplementierungen
- Aufruf von Operationen der Objektimplementierungen durch das IDL Skeleton
- Authentisierung der aufrufenden Clients

CORBA unterscheidet strikt zwischen Servern und Objekten . Ein Server ist eine Ablaufumgebung für Objektinstanzen, ein Prozess oder Thread. Ein Objekt ist eine Instanz die ein bestimmtes Interface implementiert. Innerhalb der Ablaufumgebung eines Servers können mehrere Objekte existieren. Die Aktivierung eines Objektes erfolgt innerhalb eines Servers. Die Spezifikation des BOA in der CORBA 2.0 Spezifikation definiert vier Aktivierungsmodi:

- *Shared Server* : Alle Objekte eine Shared Servers teilen sich den gleichen Prozess. Existiert bereits ein Prozess, so reicht der BOA eingehende Requests an diesen Server weiter. Existiert kein Prozess, wird dieser generiert und der Request an den erzeugten Server übergeben. Zwischen Server und Objekten besteht eine 1:n Beziehung. Dies impliziert eine Serialisierung der Requests für alle Objekte des Server.

- *Unshared Server* : Jedes aktive Objekt existiert innerhalb eines separaten Prozesses. Existiert für einen dem BOA vorliegenden Request keine aktive Server-Instanz, erzeugt der BOA eine neue Instanz. Zwischen Server und Objekten besteht eine 1:1 Beziehung. Auf der Ebene der Requests entspricht dies eine Serialisierung auf Objektebene.

- *Server-Per-Method :* Für jeden Methodenaufruf wird ein neuer Prozess generiert.

- *Persistent Server :* Für einen Server dieses Typs erfolgt keine explizite Aktivierung. Solche Server werden typischerweise beim Startvorgang des Systems aktiviert. Der BOA reicht vorliegende Requests an den existierenden Server weiter.

Der BOA benutzt eine Software-Komponente, die als Implementation Repository (IMR) bezeichnet wird. Der BOA greift auf Dienste des zugrundeliegenden Betriebssystems zu, um

© J.M.Joller

ein Server-Objekt zu aktivieren. Dafür werden betriebssystemspezifische Informationen benötigt: Ort des Programm-Codes und Aktivierungsmodus der Server-Implementierung. Der Basic Object Adapter greift auf im IMR verwalteten Informationen zu. Die Struktur dieser Informationen ist durch die OMG nicht definiert worden, da sie vom verwendeten Betriebssystem abhängen.

1.1.11. Das Interface Repository

Für die Erzeugung und Aktivierung von Requests wurden in den vorangegangenen Abschnitten zwei Methoden definiert: Einerseits die statische Variante (SII) durch Stubs, andererseits die dynamische Methode mit Hilfe des DII. In beiden Fällen muss der ORB die Strukturen der Requests kennen, um sie bearbeiten zu können. Der ORB ist nicht nur für die Überprüfung der Typkonsistenz bei der Verwendung des DII verantwortlich, sondern auch für das Marshalling [9] und Unmarshalling der Operationsparameter. Die hierfür benötigten Informationen über die Interfaces, die in Form von IDL-Definitionen vorliegen, werden von einer Komponente des CORBA-Systems verwaltet. Diese Komponente wird als Interface Repository (IR) bezeichnet und ist für das zentrale Management der IDL-Definitionen (Signaturen) zuständig. Die Aufgabenstellung unterteilt sich in:

- persistentes Vorhalten der Daten
- Die öffentliche Bereitstellung dieser Informationen und deren Verteilung
- Das Management der Daten

Das Interface Repository, beziehungsweise die von ihm verwalteten Informationen, sind für die Inter-ORB-Kommunikation wesentlich. Also der Kommunikation zwischen Object Request Brokern von, nicht notwendigerweise unterschiedlichen, CORBA-Implementierungen.

[9] Die Umwandlung eines Datums in eine andere Repräsentation wird Marshalling genannt.

© J.M.Joller

1.1.12. Der Portable Object Adapter (POA)

Eine konzeptionelle Schwäche des BOA der CORBA 2.0 Spezifikation besteht in der nur unzureichenden Unterstützung für Server-Implementierungen. Diese Unterspezifikation des BOA führte in der Vergangenheit zu proprietären Erweiterungen des Funktionsumfangs durch die Hersteller von CORBA konformen Implementierungen. Die CORBA 2.2 Spezifikation bietet mit dem Portable Object Adapter (POA) einen um wesentliche Funktionalitäten erweiteren Objektadapter, der die im BOA vorhandenen konzeptionellen und funktionalen Lücken schliesst. Das Design des POA baut nicht auf dem BOA auf. Die vorhandenen BOA Implementierungen, inklusive der proprietären Erweiterungen der Hersteller, existiert parallel zum POA. Der POA ersetzt den BOA vollständig. Das Design des POA hat folgende Zielsetzungen:

- Programmierern soll die Implementierung von Objekten ermöglicht werden, die zwischen unterschiedlichen CORBA-Implementierungen portabel sind.
- Objekte mit persistenter Identität sollen realisierbar sein. Der Lebenszyklus eines Objektes erstreckt sich über den Lebenszyklus der Server-Implementierung.
- Die transparente Aktivierung von Objekten soll unterstützt werden.
- Die Implementierung eines Objektes soll mehrere Objektinstanzen gleichzeitig unterstützen können.
- Innerhalb eines Servers können mehrere POA-Instanzen existieren
- Die Objektimplementierungen selbst sollen für viele Verhaltensaspekte des durch sie implementierten Objekts verantwortlich sein.

Das POA-Modell ist eine Konkretisierung des abstrakten Objektmodells der OMA. Die meisten Komponenten des CORBA-Modells finden sich auch im POA-Modell wieder. Die Spezifikationen der Komponenten des POA-Modells sind jedoch schärfer da diese Komponenten spezialisiert sind. Diese Komponenten des POA-Modells sind:

- *Client* : Ein Client ist ein Ausführungsumgebung, innerhalb der Requests an Objekte erzeugt werden können. Hierzu muss innerhalb des Client eine Objektreferenz für das Objekt vorhanden sein.

- *Server* : Ein Server ist eine Ausführungsumgebung innerhalb der die Implementierung eines Objektes existiert.

- *Objekt :* Ein Objekt ist ein CORBA-Objekt im abstrakten Sinne, also eine Entität, die eine Identität, ein Interface und eine Implementierung besitzt. Aus der Sicht des Client ist die Identität des Objektes durch die Objektreferenz gekapselt

- *Servant* : Ein Servant ist eine Entität, die die Implementierung von Methoden eines Objekts bereitstellt. Ein Request der von einem Client über seine Objektreferenz gemacht wird, führt zum Aufruf des entsprechenden Servant. Der ORB vermittelt den Request in Zusammenarbeit mit einem POA. Ein Servant kann die Methoden verschiedener Objekte realisieren.

- *Objekt-ID* : Eine Objekt-ID wird vom POA und Servant zur Identifikation eines einzelnen abstrakten CORBA-Objekts benutzt. Für einen Client sind die Objekt-IDs nicht sichtbar in der Objektreferenz gekapselt.

© J.M.Joller

- *Objekteferenz* : Eine Objektreferenz entspricht weitestgehend der Objektreferenz des CORBA-Objektmodells. In der Objektreferenz sind die Identität eines POA und die Objekt-ID gekapselt.

- *POA* : Ein POA ist eine identifizierbare Entität innerhalb des Server-Kontext. Jeder POA verwaltet einen Namensraum für Objekt-IDs. Das Verhalten aller Objektimplementierungen die durch einen bestimmten POA verwaltet werden wir durch eine Policy beschrieben. Eine Policy ist ein dem POA zugeordnetes Objekt, das bestimmte, verhaltensbestimmende Attribute enhält.

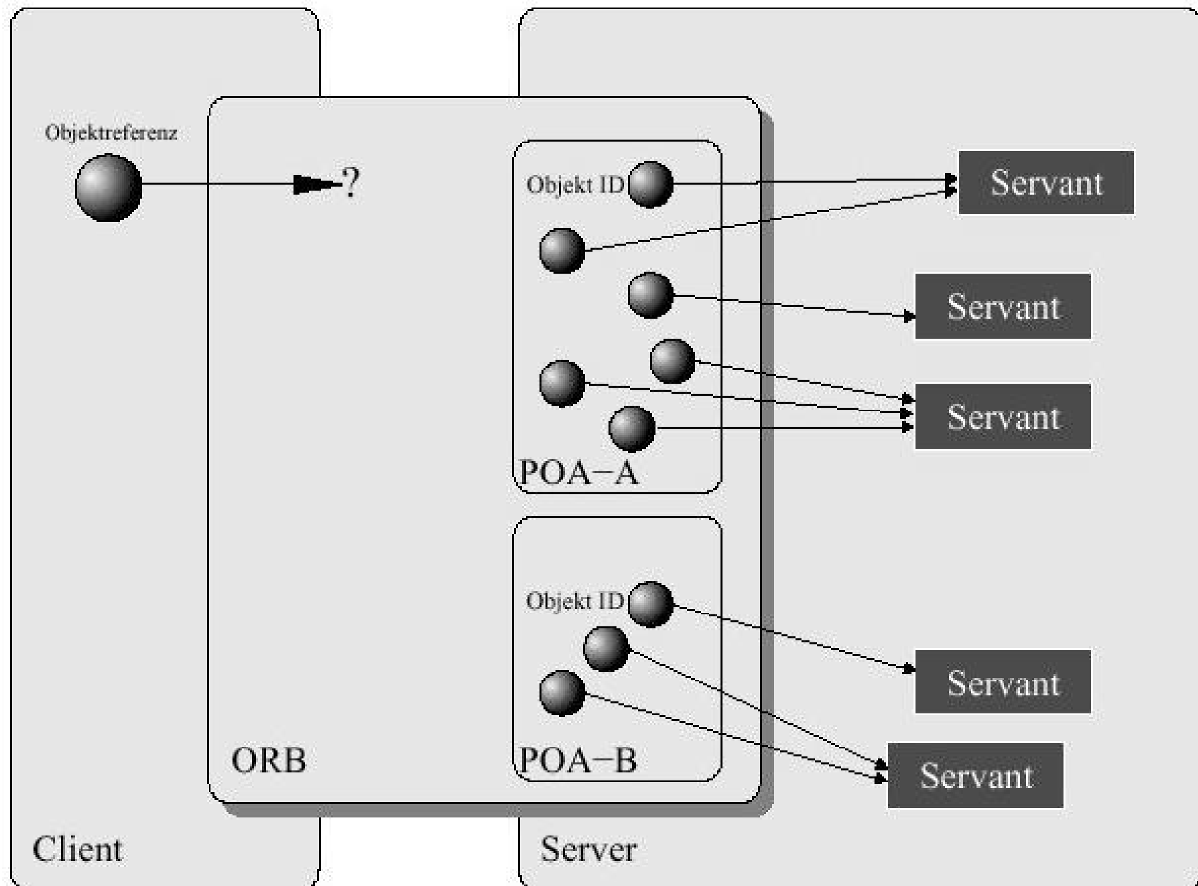


Abbildung 7Abstraktes POA Modell

Die der Architektur zugrundeliegende Idee besteht darin, innerhalb jedes Server mehrere POAs in einer hierarchischen Struktur zu erlauben. Initial existieren in einem Server nur der Root-POA , dessen Objektreferenz der Server beim ORB erfragt. Neue POA-Instanzen können in die bereits existierende POA-Hierarchie eingefügt werden.

Jeder POA verwaltet eine Active Object Map (AOM). Diese enthält die Zuordung aller durch diesen POA aktivierten Objekte zu Servants. Eine Aktivierung eines Objektes durch einen POA erfolgt durch Aufruf des entsprechenden Servant. Der POA durchsucht für alle eingehenden Request s seine AOM um den verantwortlichen Servant zu ermitteln und diesem den Request zu übergeben. Wird kein entsprechender Eintrag in der AOM gefunden, wird der Request entweder an einen generischen Servant übermittelt oder ein Servant Manager wird aufgerufen. Die Aufgabe des Servant Managers besteht dann in der Lokalisierung und Aktivierung eines passenden Servant.

© J.M.Joller

Die Einführung des POA in der CORBA 2.2 Spezifikation schiesst die funktionale- und konzeptionelle Lücke, die der BOA für die Realisierung von portablen Server-Implementierungen hat. Erweiterungen im funktionalen Bereich und die Möglichkeit Policies für das Verhalten von Objektimplementierungen anzugeben erleichtern die portable Implementierung von Servern. [OMG 1998], [Puder und Römer 1997]

1.1.13. Die CORBA-Dienste (COSS)

Die Spezifikation der Common Object Services (COSS) [OMG 1997b] beschreibt generische Dienste die im Rahmen der CORBA-Architektur zur Verfügung stehen. Sie stellen Komponenten für den Aufbau komplexerer Systeme und Komponenten dar. Die derzeitige Spezifikation umfasst folgende Dienste:

1.1.13.1. Naming Service

Eine Anfrage an den Namensdienst erfolgt per standard CORBA-Request, bei dem ein symbolischer Name durch den Namendienst in eine CORBA-Objektreferenz aufgelöst wird. Der Namensraum des Name Service ist hierarchisch organisiert. Vorhandene Namensdienste wie DCE CDS, ISO X500, Sun NIS+ oder LDAP können als Grundlage für die Implementierung des Name Service genutzt werden.

1.1.13.2. Relationship Service

Relationen zwischen unterschiedlichen Objekten können prinzipiell objektintern oder objektextern geführt werden. Der Relationship Service erlaubt es Relationen zwischen beliebigen Objekten einzuführen. Die Tatsache, dass solche Relationen bestehen ist für die betroffenen Objekte transparent.

1.1.13.3. Life Cycle Service

Dieser Dienst definiert ein Interface um Objekte zu generieren, kopieren, migrieren und zu löschen. In diesem Zusammenhang muss die Referenzielleintegrität sichergestellt werden. Dies wird durch die Zusammenarbeit des Life Cycle Service mit dem Relationship Service sichergestellt.

1.1.13.4. Query Service

Die Dienste des Query Service ermöglicht es aus einer Grundmenge von Objekten eine Teilmenge zu bilden. Die Objekte dieser Teilmenge genügen einer gegebenen Bedingung hinsichtlich bestimmter Attribute. Dabei wird das Prinzip der Kapselung des internen Zustands eines Objektes nicht verletzt, vielmehr wird durch den Query Service eine einheitliche Abfrageschnittstelle definiert. Das Design des Query Service ist so angelegt, dass Abfragemechanismen von existierenden Datenbanksystemen durch ihn gekapselt werden können.

1.1.13.5. Collection Service

Der Collection Service erlaubt es Objekte zu sogenannten Collections zu gruppieren und Operationen auf diesen Objektgruppen auszuführen. Beispiele für solche Collections sind Queues, Stacks, Lists, Arrays, Trees und Mengen.

1.1.13.6. Event Service

Die asynchrone Verteilung von Events von Event-Erzeugern zu Event-Konsumenten ist die Aufgabe des Event Service. Die dem Publish/Subscribe-Paradigma folgende Event-Verteilung

© J.M.Joller

beruht auf standard CORBA-Requests, die dem Request/Response-Paradigma folgen. Die notwendige Abbildung wird von einem Mediator (Eventchannel) vorgenommen.

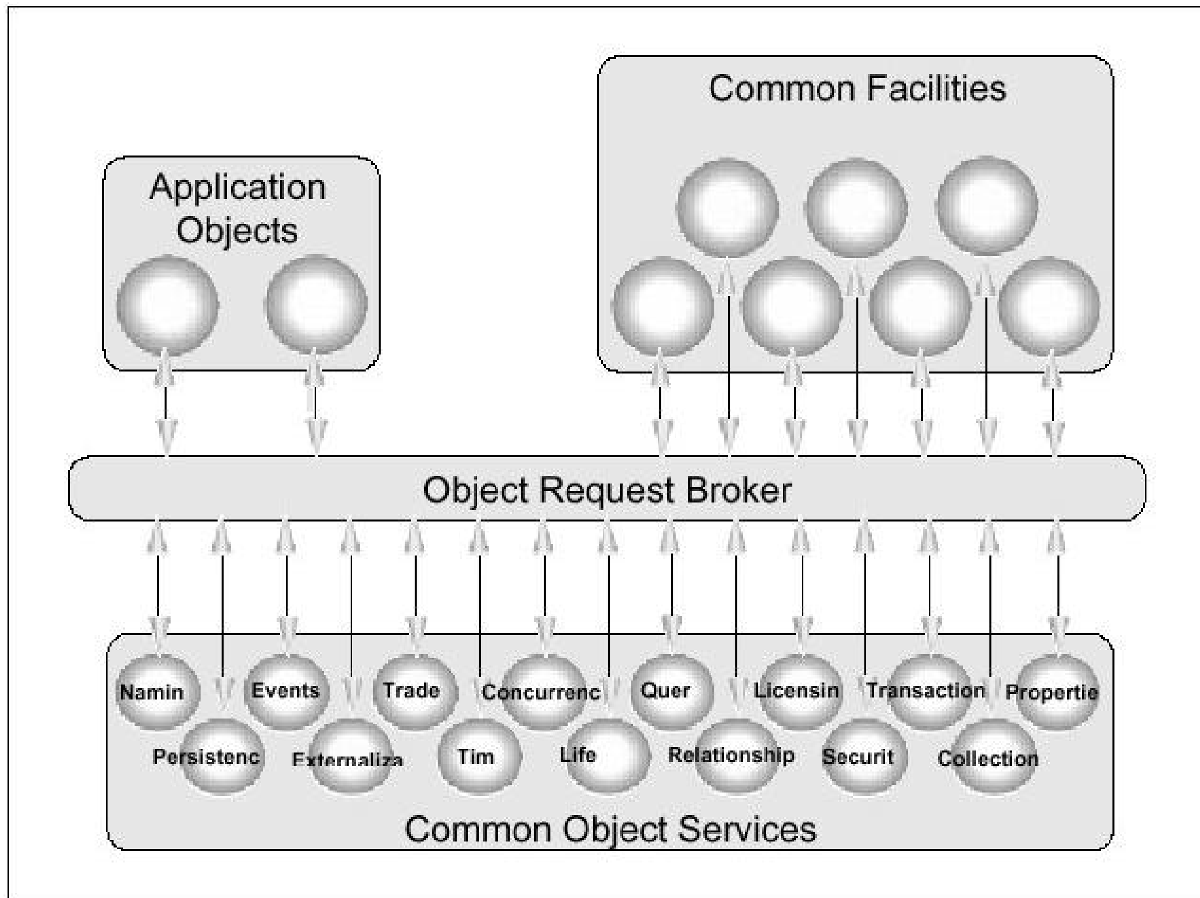


Abbildung 8Die Common Object Services des Object Management Architecture

1.1.13.7. Trader Service

Der Trader ist ein Repository für Objekteigenschaften. Eine Eigenschaft ist dabei ein Name, Wert Paar. Ein "lookup" Request an einen Trader Service beschreibt durch Angabe einer Anzahl dieser Tupel die Eigenschaft des gesuchten Objekts. Der Trader liefert als Rückgabewert Objektreferenzen auf Objekte die den spezifizierten Eigenschaften genügen. Die Ausdehnung der Suche kann bei der Anfrage beim Trader Service begrenzt werden.

1.1.13.8. Transaktions Service

Dieser Service integriert das Transaktionskonzept in die CORBA-Architektur. Der Object Transaktions Service (OTS) unterstützt zwei Transaktionsmodelle. Das einfache Transaktionsmodell (flat transactions) und das Modell geschachtelter Transaktionen (nested transactions). Das Modell der geschachtelten Transaktionen erlaubt es innerhalb einer Transaktion weitere, sogenannte Subtransaktionen, zu starten. Eine solche Transaktion wird dann erfolgreich abgeschlossen, falls alle Subtransaktionen erfolgreich abgeschlossen wurden.

1.1.13.9. Concurrency Control Service

Die Serialisierung von Zugriffen auf gemeinsam genutzte Ressourcen kann durch den Concurrency Control Service (CCS) modelliert werden. Der Begriff Ressource ist dabei nicht auf Objekte beschränkt. Die Koordinierung der Zugriffe auf Ressourcen erfolgt durch Locks, die durch den CCS verwaltet und koordiniert werden.

© J.M.Joller

1.1.13.10. Persistence Service

Der Persistent Object Service (POS) erlaubt es den Zustand eines Objekts persistent zu machen und zu einem beliebigen Zeitpunkt zu rekonstruieren. Als Speicher können unter anderem Dateisystem und Datenbanken, in relationaler und objektorientierter Form, Verwendung finden. Hauptsächlich soll durch den POS ein einheitliches Interface für unterschiedliche Speicherungsarten definiert werden.

1.1.13.11. Externalization Service

Die von diesem Dienst gebotene Funktionalität besteht in der Möglichkeit Objekte als ganzes zu Externalisieren und zu einem beliebigen Zeitpunkt, an einem beliebigen Ort zu Internalisieren. Prinzipiell entspricht dies einem Kopieren des Objektes mit einem Zwischenschritt. Die externe Form des Objektes kann auf unterschiedlichen Medien abgelegt oder transportiert werden. Der Externalization Service ergänzt den POS und den Life Cycle Service.

1.1.13.12. Time Service

Die synchronisation der Zeit auf unterschiedlichen Systemen ist für ein System verteilter Objekte von Bedeutung. Das durch den Object Time Service definierte Interface erlaubt die Abfrage der akutellen Zeit und die Abschätzung des relativen Fehlers der gelieferten aktuellen Zeit. Bei der Implementierung des Object Time Service kann auf existierende Zeitsynchronisationsmechanismen zurückgegriffen werden.

1.1.13.13. Security Service

Der Sicherheitstaspekt in verteilten Systemen untergliedert sich in die Bereiche Authentifizierung, Authorisation, Auditing, Kryptifizierung und Non-Repudiation [10], die durch die Dienste des Security Service abgedeckt werden. Das durch den Security Service definierte Interface erweitert das CORBA-Objektmodell um Sicherheitsoperationen. Da die Sicherheitsfunktionalität integraler Bestandteil des ORB ist können Applikationen ohne Modifikation auf einem solchen ORB laufen. Erweiterte Anforderungen im Sicherheitsbereich müssen jedoch schon in der Design- und Implementierungs-Phase von Objekten berücksichtigt werden und schliessen die direkte Nutzung der Dienste des Security Service ein.

1.1.14. Zusammenfassung

Das Ziel beim Entwurf von CORBA ist eine Architektur für ein System verteilter Objekte. Die Schwerpunkte des Design liegen in den Bereichen: Ortstransparenz, Aufruftransparenz, Plattformtransparenz und Sprachheterogenität. Dieses Ziel wird aus Sicht der Applikationen erreicht. Die Implementierung von Objekten setzt für die Nutzung anderer CORBA-Objekte lediglich die Kenntnis deren öffentlicher Schnittstelle und eine Objektreferenz voraus. Plattformtransparenz bedeutet in diesem Zusammenhang aber nicht die Portabilität von Objektimplementierungen. Bei der Implementierung der CORBA-Objekte wird von den durch das zugrundeliegende Betriebssystem gebotenen Dienste Gebrauch gemacht. CORBA abstrahiert auf Implementierungsebene nicht von dem zugrundeliegenden Betriebssystemen. Gerade aus diesem Grund ist der Einsatz von JAVA als Implemtierungssprache für CORBA-Objekte ein aktuelles Thema. JAVA kapselt die Betriebssystemspezifika durch die virtuelle Maschine der JAVA-Laufzeitumgebung. Bei der Implementierung neuer Applikationen bietet der gemeinsame Einsatz von CORBA und JAVA viele Vorteile. Gleichzeitig ist durch die CORBA-Architektur eine Migration bestehender Applikationen in die objekorientierte Welt

[10] engl non-repudiation := die Unleugbarkeit

© J.M.Joller

CORBAs möglich. Hierzu wird eine Schnittstellenbeschreibung in IDL erstellt und aus dem hieraus generierten Server-Skeleton auf die Funktionen der vorhandenen Software-Komponenten zugegriffen [11]

Die Entwicklung von CORBA ist sicher noch nicht abgeschlossen. In vielen Bereichen sind Veränderungen und Erweiterungen zu erwarten. Die Veränderungen werden in unterschiedlichen Bereichen der CORBA-Architektur unterschiedlich stark sein. Das CORE Objektmodell der OMA wird sich nur in geringem Masse verändern. Der Bereich der CORBA-Dienste ist relativ dynamisch.Vorhandene Spezifikationen werden überarbeitet und erweitert, die Veröffentlichung neuer Dienste durch die OMG steht bevor. Spezifikationen für Probleme auf höherer Ebene werden folgen. Diese Spezifikationen werden im Bereich der komponentenbasierten Softwareentwicklung liegen. Hier sind ebenfalls synergetische Effekte durch die Enterprise JAVA Beans (EJB) zu erwarten [Orfali 1998].

[11] Diese Technik ist unter dem Begriff Wrapping bekannt.

© J.M.Joller

1.1.15. Literatur

- **[Baker 1997]**
  Sean Baker: CORBA Distributed Objects, Using Orbix: Addison-Wesley 1997
- **[Ben-Natan 1997]**
  Ron Ben-Natan: CORBA, A Guide To Common Object Request Broker Architecture: McGraw-Hill 1997
- **[Gries 1981]**
  David Gries: The Science of Programming. New York: Springer 1981
- **[Mowbray und Malveau 1997]**
  Thomas J. Mowbray, Raphael C. Malveau: CORBA Design Patterns: John Wiley & Sons 1997
- **[Orfali et al. 1997]**
  Robert Orfali, Dan Harkey, Jeri Edwards: Instant CORBA: John Wiley & Sons 1997
- **[Orfali und Harkley 1998]**
  Robert Orfali, Dan Harkley: Client/Server Programming with JAVA and CORBA: John Wiley & Sons 1998
- **[OMG und Soley 1992]**
  Object Management Group (OMG), Richard Mark Soley: Object Management Architecture Guide. OMG TC Document No. 92.11.1, Rev. 2.0, Second Edition. New York: John Wiley & Sons.
- **[OMG 1994]**
  IDL C++ Language Mapping Specification: OMG RFP Submission by Exportsoft Corporation, by Digital, Hewlett-Packard, IONA Technologies, IBM, Novell, and SunSoft. OMG Document No. 94-8-2. New York: John Wiley & Sons
- **[OMG 1997a]**
  The Common Object Request Broker: Architecture Spezification, Revision 2.0, Juli 1997, Object Management Group, formal document 97-07-04, http://www.omg.org
- **[OMG 1997b]**
  CORBAservices: Common Object Services Specivication. November 1997, Object Management group, formal document 97-12-02, http://www.omg.org
- **[OMG 1998]**
  The Common Object Request Broker: Architecture Spezification, Revision 2.2, February 1998, Object Management Group, formal document 98-02-33, http://www.omg.org
- **[Puder und Römer 1997]**
  Arno Puder, Kay Römer: MICO is CORBA. A CORBA 2.0 compliant implementation, dpunkt.verlag 1997
- **[Pope 1997]**
  Alan Pope: The CORBA Reference Guide, Understanding the Commen Object Request Broker Architecture: Addison-Wesley 1997
- **[Redlich 1996]**
  Jens-Peter Redlich: Corba 2.0, Praktische Einfhrung für C++ und Java: Addison-Wesley 1996
- **[Stal 1997]**
  Michael Stal: World Wide CORBA, verteilte Objekte im Netz, in: JAVAspektrum, 1997, Ausgabe 6, Seiten 28-34
- **[Schmidt und Vinoski 1997]**
  Douglas C. Schmidt, Steve Vinoski: Object Interconnections, The OMG Event Services, in: SIGS C++ Report, 1997, Ausgabe 2

© J.M.Joller

1.2. Beispiele mit dem Portable Object Adapter (POA)

Nun wollen wir einfache CORBA Applikationen mit der IDL (Interface Definition Language) und den Java IDL Compiler beschreiben und erzeugen. Seit JDK1.4 kann serverseitig durch den idlj Compiler das *Portable Inheritance Model*, oder POA Modell generiert werden. POA steht für Portable Object Adapter. In den OMG Dokumenten wird dieser Adapter in der CORBA Spezifikation beschrieben[12]. Wir befassen uns weniger mit der abstrakten Spezifikation als mit konkreten Beispielen.

CORBA unterstützt mindestens zwei unterschiedliche serverseitige Mappings für die Implementation von IDL Interfaces:

1) **das Vererbungsmodell**
   Mit dem Vererbungsmodell implementieren Sie das IDL Interface mit Implementationsklassen, welche auch die Compiler-generierten Skeletons erweitern.

   Zu diesem Implementationsverfahren gehören:
   - der OMG Standard *POA*.
     Zu einem gegebenen Interface `MeinInterface`, welches in `MeinInterface.idl` beschrieben wird, kann durch den idlj Compiler eine Java Klasse generiert werden, `MeinInterfacePOA.java`. Sie müssen das Interface MeinInterface implementieren und zugleich `MeinInterfacePOA` erweitern. `MeinInterfacePOA` beschreibt ein Stream-basiertes Skeleton, welches `org.omg.PortableServer.Servant` erweitert, welche als Basis Klasse für alle POA Servant Implementation dient.
   - *ImplBase*: zu einem Interface MeinInterface, welches in MeinInterface.idl beschrieben ist, wird MeinInterfaceImplBase.java generiert. Sie müssen dann das Interface MeinInterface in einer Klasse implementieren, welche auch MeinIntefaceImplBase erweitert.

   ImplBase war bis JDK1.3 Standard innerhalb von Java. Da diese Methode aber nicht OMG konform ist, wurde sie durch POA abgelöst. Sie sollten also wenn immer möglich nur noch die POA Methode benutzen, ausser Sie benötigen Kompatibilität zu JDK1.3 und älter.

2) **das Delegationsmodell**
   In diesem Modell implementieren Sie ein in IDL beschriebenes Interface mittels zweier Klassen:
   - einer IDL-generierten Tie Klasse, welche aus einer Compiler-generierten Skeleton abgeleitet wird, aber alle Calls an eine Implementationsklasse delegiert.
   - einer Klasse, welche das IDL-generierte Operations / Methoden Interface implementiert (beispielsweise MeineApplikationOperations). In dieser Klasse werden alle in der IDL Beschreibung aufgeführten Methoden definiert.

   Das Delegationsmodell wird auch als *Tie* oder *Tie Delegations* Modell bezeichnet. Da man bei diesem Modell auf die Compiler-generierten Skeletons (POA oder ImplBase) zurück greift, beschreiben wir das Tie Modell im Zusammenhang mit den andern Modellen.

[12] CORBA-Spec99-10-07.pdf, Seiten 301 - 360

© J.M.Joller

Nun wollen wir als erstes ein POA Vererbungsmodell implementieren.

Der Reihe nach müssen wir
- das IDL Interface für unsere Anwendung definieren
- einen Server definieren, welcher ein Servant Objekt anbietet, über einen Namenservice und die Standard Server- seitige Implementation (POA).
- einen Client definieren, welcher den Objektnamen des Servant-Objekts kennt, eine Referenz aus dem Namensraum / dem Nameservice bestimmt und die Methoden dieses Objekts einsetzt.

Zusätzlich werden Sie die entsprechenden Befehle / Prozeduren kennen lernen, mit denen die Applikation generiert und gestartet wird.

1.2.1. Definition des Interfaces

Als erstes muss man bei der Implementation einer CORBA Applikation die Objekte und deren Interfaces (Methoden, Parameter) spezifizieren. Dies geschieht wie in CORBA üblich mit der IDL, der Interface Definition Language. Die Syntax von IDL lehnt sich an jene von C++ an. Sie können Module, Interfaces, Datenstrukturen und vieles mehr definieren. IDL kann auf verschiedene Sprachen abgebildet werden, beispielsweise auf Java. Die Details dieser Abbildung finden Sie ebenfalls in der CORBA Spezifikation, Kapitel 3 (und einigen anderen PDFs).

Der folgende Code beschreibt in OMG IDL ein CORBA Objekt, dessen wieGehts() Methode eine Zeichenkette zurück liefert und deren shutdown() Methode den ORB, den Objekt Request Broker (den Server), herunterfährt.

Hallo.idl

```
module HalloApp
{
  interface Hallo
  {
  string wieGehts();
  oneway void shutdown();
  };
};
```

Bemerkung 1

Sie müssen darauf achten, dass der Name des Interfaces und der Name des Moduls unterschiedlich sind (hier `HalloApp` und `Hallo`), sonst kann es je nach Tool / Compiler zu Problemen führen.

© J.M.Joller

1.2.1.1. Übersetzen des Interfaces

Da wir die generierten Klassen in den Server und Client Programmen benötigen, bietet es sich an, die IDL Beschreibung schon mal zu übersetzen.

Auf dem Server / der CD steht Ihnen dazu eine Batch Prozedur zur Verfügung:

```
@echo off
Rem
Rem ------------------------------------------------------------------------
set JAVA_HOME=d:\jdk1.4
Rem ------------------------------------------------------------------------
Rem
%JAVA_HOME%\bin\idlj -v -fall Hallo.idl
@echo Die generierten Dateien stehen im Unterverzeichnis HalloApp
%JAVA_HOME%\bin\javac HalloApp\*.java
pause
```

Wichtig ist das Flag `-fall`, da nur in diesem Fall die Client- und Server-seitigen Stubs / Skeletons generiert werden (CORBA braucht beide, nicht wie RMI, bei dem man auf die Skeletons verzichten kann).

Die Java Version muss 1.4 oder neuer sein!

1.2.1.2. Generierte Dateien

Der `idlj` Compiler generiert eine ganze Menge Dateien. Die Anzahl Dateien hängt von der konkreten Implementation und Version ab.

Nochmal: *wichtig ist, dass Sie die Option* **-fa**ll *nicht vergessen.*

Und hier die Liste der generierten Dateien:

- `HalloPOA.java`
  Die abstrakte Klasse ist eine Stream-basierte Server Skeleton Klasse, welche CORBA Funktionalitäten für den Server bereit stellt. Die Klasse erweitert die Klasse `org.omg.PortableServer.Servant` und implementiert das `InvokeHandler` Interface und das `HalloOperations` Interface.
  Die Server Klasse `HalloImpl` erweitert `HalloPOA`.

- `_HalloStub.java`
  Diese Klasse repräsentiert den Client Stub, welcher CORBA Funktionalität für den Client zur Verfügung stellt. Die Klasse erweitert `org.omg.CORBA.portable.ObjectImpl` und implementiert das `Hallo.java` Interface.

- `Hallo.java`
  Dieses Interface enthält die Java Version unseres IDL Interfaces. Das `Hello.java` Interface erweitert `org.omg.CORBA.Object` und stellt Standard CORBA Objekt Funktionalität zur Verfügung. Das Interface erweitert auch das `HelloOperations` Interface und `org.omg.CORBA.portable.IDLEntity`.

- `HalloHelper.java`

© J.M.Joller

Diese Klasse stellt zusätzliche Methoden, wie beispielsweise die `narrow()` Methode zur Verfügung, welche benutzt wird, um CORBA Objektreferenzen zu casten. Die Helper Klasse ist verantwortlich für das Lesen und Schreiben der Datentypen in die CORBA Streams und das Extrahieren und Einfügen der Datentypen von Anys (siehe unten). Die Holder Klasse delegiert das Lesen und Schreiben an die Helper Klasse.

- `HalloHolder.java`
  Diese final Klasse enthält `Hallo`. Immer wenn Sie `out` oder `inout` Parameter verwenden, muss die Holder Klasse aktiv werden. Sie stellt auch Dienste für `org.omg.CORBA.portable.OutputStream` /InputStream zur Verfügung, speziell Dienste, welche CORBA anbietet, die sich aber in Java schlecht realisieren lassenDie Holder delegiert das Lesen und Schreiben an die Helper Klasse und implementiert `org.omg.CORBA.portable.Streamable`.

- `HalloOperations.java`
  Das Interface beschreibt die Methoden `wieGehts()` und `shutdown()`. Dieses Interface wird von Stubs und Skeletons benutzt.

1.2.1.2.1. Inhalt der generierten Dateien - ein Beispiel

Sie können diesen Abschnitt überspringen, falls Sie sich mehr für die Anwendungen interessieren.

Als Ausgangspunkt dient folgende Interface Definition:
Hallo.idl

```
module HalloApp
{
  interface Hallo
  {
  string wieGehts();
  oneway void shutdown();
  };
```

Hallo.java - das Signatur-Interface
Das `Hallo` (Signatur-)Interface in dieser Datei erweitert `org.omg.portable.IDLEntity`, `org.omg.CORBA.Object` und das Methoden-Interface `HalloOperations`.

Aus Client-Sicht implementiert eine Objektreferenz für ein CORBA Hallo Objekt dieses Interface.

Der Stub (Client-seitiges Proxy-Objekt) implementiert das Hallo Interface: er generiert Programmcode für die Methoden, ruft die Methoden auf und sorgt für das Mashalling und UnMashalling der Argumente.

```
package HalloApp;
/**
* HalloApp/Hallo.java
* Generated by the IDL-to-Java compiler (portable), version "3.1"
* from Hallo.idl
* Dienstag, 17. Juli 2001 19.40 Uhr CEST
*/
public interface Hallo extends HalloOperations, org.omg.CORBA.Object,
org.omg.CORBA.portable.IDLEntity {} // interface Hallo
```


© J.M.Joller

HalloOperations.java - das Methoden-Interface
Das Java *Operations*- Interface wird serverseitig benutzt. Der Server stellt Implementationen für die in diesem Interface angegebenen Methoden zur Verfügung. In unserem Fall sind dies zwei Methoden. Stubs und Skeleton teilen sich diese Datei.

In der Regel wird der Programmierer, der den Server schreiben muss, `HalloPOA` erweitern und die Methoden ausprogrammieren.

```
package HalloApp;


/**
* HalloApp/HalloOperations.java
* Generated by the IDL-to-Java compiler (portable), version "3.1"
* from Hallo.idl
* Dienstag, 17. Juli 2001 19.40 Uhr CEST
*/

public interface HalloOperations
{
  String wieGehts ();
  void shutdown ();
} // interface HalloOperations
```

HalloHelper.java - die Helper Klasse
Die `HalloHelper` Klasse stellt zusätzliche Funktionen zur Verfügung, beispielsweise eine `narrow()` Methode, welche zum Casten der CORBA Objektreferenzen verwendet wird.

Die Helper Klasse liest und schreibt Datentypen aus/ in CORBA Streams und `Anys.` Die Holder Klasse (siehe unten) delegiert das Schreiben und Lesen an die Helper Klasse.

```
package HalloApp;


/**
* HalloApp/HalloHelper.java
* Generated by the IDL-to-Java compiler (portable), version "3.1"
* from Hallo.idl
* Dienstag, 17. Juli 2001 19.40 Uhr CEST
*/

abstract public class HalloHelper
{
  private static String  _id = "IDL:HalloApp/Hallo:1.0";

  public static void insert (org.omg.CORBA.Any a, HalloApp.Hallo that)
  {
    org.omg.CORBA.portable.OutputStream out = a.create_output_stream ();
    a.type (type ());
    write (out, that);
    a.read_value (out.create_input_stream (), type ());
  }

  public static HalloApp.Hallo extract (org.omg.CORBA.Any a)
  {
    return read (a.create_input_stream ());
  }
```


© J.M.Joller

```
  private static org.omg.CORBA.TypeCode __typeCode = null;
  synchronized public static org.omg.CORBA.TypeCode type ()
  {
    if (__typeCode == null)
    {
      __typeCode = org.omg.CORBA.ORB.init ().create_interface_tc
(HalloApp.HalloHelper.id (), "Hallo");
    }
    return __typeCode;
  }

  public static String id ()
  {
    return _id;
  }

  public static HalloApp.Hallo read (org.omg.CORBA.portable.InputStream
istream)
  {
    return narrow (istream.read_Object (_HalloStub.class));
  }

  public static void write (org.omg.CORBA.portable.OutputStream ostream,
HalloApp.Hallo value)
  {
    ostream.write_Object ((org.omg.CORBA.Object) value);
  }

  public static HalloApp.Hallo narrow (org.omg.CORBA.Object obj)
  {
    if (obj == null)
      return null;
    else if (obj instanceof HalloApp.Hallo)
      return (HalloApp.Hallo)obj;
    else if (!obj._is_a (id ()))
      throw new org.omg.CORBA.BAD_PARAM ();
    else
    {
      org.omg.CORBA.portable.Delegate delegate =
((org.omg.CORBA.portable.ObjectImpl)obj)._get_delegate ();
      HalloApp._HalloStub stub = new HalloApp._HalloStub ();
      stub._set_delegate(delegate);
      return stub;
    }
  }

}
```


© J.M.Joller

HalloHolder.java - die Holder Klasse
Die Java Klasse `HalloHolder` enthält eine öffentliche Instanz von `Hallo`. Immer wenn ein `out` oder `inout` Parameter in IDL verwendet wird, kommt die Holder Klasse zum Einsatz. Diese stellt Methoden für

`org.omg.CORBA.portable.OutputStream` und `org.omg.CORBA.portable.InputStream`

Parameter zur Verfügung, welche in CORBA möglich, in Java aber nur schlecht abbildbar sind. Die eigentlichen Lese- / Schreib- Operationen werden durch die Helper Klasse erledigt. Die Klasse implementiert `org.omg.CORBA.portable.Streamable`.

```
package HalloApp;

/**
* HalloApp/HalloHolder.java
* Generated by the IDL-to-Java compiler (portable), version "3.1"
* from Hallo.idl
* Dienstag, 17. Juli 2001 19.40 Uhr CEST
*/

public final class HalloHolder implements org.omg.CORBA.portable.Streamable
{
  public HalloApp.Hallo value = null;

  public HalloHolder ()
  {  }

  public HalloHolder (HalloApp.Hallo initialValue)
  {
    value = initialValue;
  }

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

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

  public org.omg.CORBA.TypeCode _type ()
  {
    return HalloApp.HalloHelper.type ();
  }

}
```


© J.M.Joller

_HalloStub.java - der Client Stub
Wie der Name sagt, ist dies der Client Stub. Die Klasse erweitert `org.omg.CORBA.portable.ObjectImpl` und implementiert das `Hallo.java` Interface.

```
package HalloApp;


/**
* HalloApp/_HalloStub.java
* Generated by the IDL-to-Java compiler (portable), version "3.1"
* from Hallo.idl
* Dienstag, 17. Juli 2001 19.40 Uhr CEST
*/

public class _HalloStub extends org.omg.CORBA.portable.ObjectImpl
implements HalloApp.Hallo
{

  public String wieGehts ()
  {
    org.omg.CORBA.portable.InputStream $in = null;
    try {
       org.omg.CORBA.portable.OutputStream $out = _request ("wieGehts",
true);
       $in = _invoke ($out);
       String $result = $in.read_string ();
       return $result;
    } catch (org.omg.CORBA.portable.ApplicationException $ex) {
       $in = $ex.getInputStream ();
       String _id = $ex.getId ();
       throw new org.omg.CORBA.MARSHAL (_id);
    } catch (org.omg.CORBA.portable.RemarshalException $rm) {
       return wieGehts ();
    } finally {
        _releaseReply ($in);
    }
  } // wieGehts

  public void shutdown ()
  {
    org.omg.CORBA.portable.InputStream $in = null;
    try {
       org.omg.CORBA.portable.OutputStream $out = _request ("shutdown",
false);
       $in = _invoke ($out);
    } catch (org.omg.CORBA.portable.ApplicationException $ex) {
       $in = $ex.getInputStream ();
       String _id = $ex.getId ();
       throw new org.omg.CORBA.MARSHAL (_id);
    } catch (org.omg.CORBA.portable.RemarshalException $rm) {
       shutdown ();
    } finally {
        _releaseReply ($in);
    }
  } // shutdown

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


© J.M.Joller

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

  private void readObject (java.io.ObjectInputStream s) throws
java.io.IOException
  {
     String str = s.readUTF ();
     String[] args = null;
     java.util.Properties props = null;
     org.omg.CORBA.Object obj = org.omg.CORBA.ORB.init (args,
props).string_to_object (str);
     org.omg.CORBA.portable.Delegate delegate =
((org.omg.CORBA.portable.ObjectImpl) obj)._get_delegate ();
     _set_delegate (delegate);
  }

  private void writeObject (java.io.ObjectOutputStream s) throws
java.io.IOException
  {
     String[] args = null;
     java.util.Properties props = null;
     String str = org.omg.CORBA.ORB.init (args, props).object_to_string
(this);
     s.writeUTF (str);
  }
} // class _HalloStub
```


© J.M.Joller

HalloPOA.java - das Server Skeleton
Die Klasse `HalloPOA` ist das Skeleton, also das serverseitige Proxy-Objekt. Die Klasse erweitert `org.omg.PortableServer.Servant`, und implementiert das `InvokeHandler` und das `HalloOperations` Interface. Die Server Klasse erweitert `HalloPOA`:
`class HalloImpl extends HalloPOA {`

```
package HalloApp;
/**
* HalloApp/HalloPOA.java
* Generated by the IDL-to-Java compiler (portable), version "3.1"
* from Hallo.idl
* Dienstag, 17. Juli 2001 19.40 Uhr CEST
*/

public abstract class HalloPOA extends org.omg.PortableServer.Servant
 implements HalloApp.HalloOperations, org.omg.CORBA.portable.InvokeHandler
{
  // Constructors
  private static java.util.Hashtable _methods = new java.util.Hashtable ();
  static
  {
    _methods.put ("wieGehts", new java.lang.Integer (0));
    _methods.put ("shutdown", new java.lang.Integer (1));
  }

  public org.omg.CORBA.portable.OutputStream _invoke (String $method,
                                org.omg.CORBA.portable.InputStream in,
                                org.omg.CORBA.portable.ResponseHandler $rh)
  {
    org.omg.CORBA.portable.OutputStream out = null;
    java.lang.Integer __method = (java.lang.Integer)_methods.get ($method);
    if (__method == null)
      throw new org.omg.CORBA.BAD_OPERATION (0,
org.omg.CORBA.CompletionStatus.COMPLETED_MAYBE);

    switch (__method.intValue ())
    {
       case 0:  // HalloApp/Hallo/wieGehts
       {
         String $result = null;
         $result = this.wieGehts ();
         out = $rh.createReply();
         out.write_string ($result);
         break;
       }

       case 1:  // HalloApp/Hallo/shutdown
       {
         this.shutdown ();
         out = $rh.createReply();
         break;
       }

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

    return out;
```


© J.M.Joller

```
  } // _invoke

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

  public String[] _all_interfaces (org.omg.PortableServer.POA poa, byte[]
objectId)
  {
    return (String[])__ids.clone ();
  }

  public Hallo _this()
  {
    return HalloHelper.narrow(
    super._this_object());
  }

  public Hallo _this(org.omg.CORBA.ORB orb)
  {
    return HalloHelper.narrow(
    super._this_object(orb));
  }


} // class HalloPOA
```


© J.M.Joller

1.2.2. Implementation des Servers und des Servants

Unser Beispielserver besteht aus zwei Klassen: dem eigentlichen *Server* und der *Servant* Klasse, als der Klasse, die unser Interface implementiert und deren Dienste durch den ORB vermittelt werden.

Das Servant Objekt, `HalloImpl`, ist die Implementation des `Hallo` IDL Interfaces. Jede `Hallo` Instanz wird durch eine `HalloImpl` Instanz implementiert. Das Servant Objekt ist eine Unterklasse von `HalloPOA`, welche vom IDL Compiler generiert wurde. Das Servant Objekt besitzt zwei Methoden:

- `wieGehts()` und
- `shutdown()`

Diese Methoden sehen genauso aus, wie die üblichen Java Methoden. Das Marshalling der Argumente und der Ergebnisse wird durch das Skeleton erledigt.

Der Server selber besitzt lediglich eine `main()` Methode, welche:

- eine ORB Instanz kreiert und initialisiert.
- eine Referenz auf den `RootPOA` bestimmt und den `POAManager` aktiviert.
- eine Servant Instanz kreiert (die Implementation des remote Objekts, welches seine Dienste anbieten soll) und dieses beim ORB anmeldet.
- eine CORBA Objektreferenz auf den Namenskontext bestimmt. In diesem Namenskontext wird das neue CORBA Objekt registriert.
- den Root Naming Context bestimmt.
- das Servant Objekt im Namenskontext unter dem Namen "POA Grüsse" registriert.
- auf Anfragen von Clients wartet.

Server.java

```
// Server.java
package poavererbung;

/**
 * Title:         CORBA - Portable Object Adapter Vererbungsmodell
 * Description:
 * Copyright:     Copyright (c) J.M.Joller
 * Company:       Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

import HalloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

import java.util.Properties;
```


© J.M.Joller

```
class HalloImpl extends HalloPOA {
  private ORB orb;

  public void setORB(ORB orb_val) {
    orb = orb_val;
  }

  // implementieren der IDL Methoden
  public String wieGehts() {
    return "Mir geht's gut! Wie geht's Dir?\n";
  }

  // in IDL wird eine shutdown() Methode definiert
  public void shutdown() {
    orb.shutdown(false);
  }
}


public class Server {

  public static void main(String args[]) {
    try{
      // Kreieren und Initialisieren des ORB
      System.out.println("[POA-Server.main()]Kreieren und Initialisieren
des
                          ORB");
      ORB orb = ORB.init(args, null);

    // Bestimmen einer Referenz auf die rootpoa & aktivieren des POAManager
    System.out.println("[POA-Server.main()]Bestimmen einer Referenz auf die
rootpoa & aktivieren des POAManager");
      POA rootpoa = (POA)orb.resolve_initial_references("RootPOA");
      rootpoa.the_POAManager().activate();

      // kreieren des Servants und registrieren beim ORB
      System.out.println("[POA-Server.main()]Kreieren des Servants und
                                                    registrieren beim
ORB");
      HalloImpl halloImpl = new HalloImpl();
      halloImpl.setORB(orb);

      // Bestimmen einer Objektreferenz zum Servant
      System.out.println("[POA-Server.main()]Bestimmen einer Objektreferenz
                                                             zum Servant");
      org.omg.CORBA.Object ref = rootpoa.servant_to_reference(halloImpl);
      Hallo href = HalloHelper.narrow(ref);

      // Bestimme den root naming context
      // NameService nutzt den transient name service
      System.out.println("[POA-Server.main()]Bestimme den root naming
                                                               context");
      org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
      // Einsatz des NamingContextExt, welcher Teil der Interoperable
      // Naming Service (INS) Spezifikation ist.
      System.out.println("[POA-Server.main()]Einsatz des
NamingContextExt");
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

      // binden der Objekt Referenz im Naming / Namensraum
```


© J.M.Joller

```
      System.out.println("[POA-Server.main()]Binden der Objekt Referenz im
                                                         Namensraum");
      String name = "POA Grüsse";
      NameComponent path[] = ncRef.to_name( name );
      ncRef.rebind(path, href);

      System.out.println("[POA-Server.main()]Der Server ist bereit und
wartet
                                                       auf Anfragen
...");
      // Warten auf Methodenaufrufe durch den Client
      orb.run();
    }
      catch (Exception e) {
        System.err.println("[POA-Server.Exception] " + e);
        e.printStackTrace(System.out);
      }
      System.out.println("[POA-Server.main()]Der Server wird herunter
                                                       gefahren ...");
  }
}
```

1.2.2.1. Den Server verstehen

In unserem Beispiel besteht der Server aus zwei Klassen, dem *Servant* und dem *Server*.

Die *Servant* Klasse ist `HalloImpl`, die Implementation des `Hallo` IDL Interfaces. Die Hallo Objekte werden durch die Klasse HalloImpl implementiert. Die Servant Klasse ist eine Unterklasse von `HelloPOA` (daher die Bezeichnung : Vererbungsmodell). `HalloPOA` wurde durch den IDL Compiler generiert. Die Servant Klasse muss pro IDL Operation eine Methode besitzen. In unserem Beispiel also insgesamt zwei: `wieGehts()` und `shutdown()`. Diese Methoden sind reine Java Methoden. Skeleton und Stub sorgen für das Mashalling und UnMashalling.

Die *Server* Klasse besitzt eine `main()` Methode, welche:

- eine ORB Instanz kreiert und initialisiert.
- eine Referenz auf das Root POA bestimmt und den `POAManager` aktiviert.
- eine Servant Instanz kreiert (also eine Instanz eines CORBA `Hallo` Objekts) und dies dem ORB mitteilt.
- eine CORBA Objektreferenz für den Namenskontext bestimmt, in dem das neue CORBA Objekt registriert wird.
- das Root des Namenskontextes bestimmt.
- das neue Objekt im Namenskontext unter dem Namen "POA Grüsse" registriert.
- auf Aufrufe der Methoden des neuen Objekts durch Clients wartet.

© J.M.Joller

1.2.2.2. Server.java verstehen

Nach dem generellen Aufbau eines Servers wollen wir nun die konkrete Implementation unseres Servers anschauen.

1.2.2.2.1. Vorbereitende Arbeiten

Die grundlegende Struktur eines CORBA Programms, hier des Servers, ist die selbe wie bei allen Java Programmen.

1. Zuerst werden die nötigen Klassen und Packages importiert, welche die Server Klasse ausmachen.
2. dann wird die `main()` Methode ausprogrammiert
3. und schliesslich die Ausnahmen abgefangen.

Importieren der benötigten Packages

Als erstes importieren wir die generierten Klassen aus dem IDL Teil:

```
// Das Package enthält unser Proxyobjekt
import HalloApp.*;

// Unser HalloServer wird Naming Service benutzen, um die Objekte zu
// verwalten (Cos:Common Object Name Services; siehe JNDI Skript)
import org.omg.CosNaming.*;

// Im Namensraum können Ausnahmen geworfen werden
import org.omg.CosNaming.NamingContextPackage.*;

// CORBA Basisklassen
import org.omg.CORBA.*;

// POA Klassen
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

// Properties zum Initialisieren des ORBs
import java.util.Properties;
```

Definition der Servant Klasse

In diesem einfachen Beispiel definieren wir die Servant Klasse gleich im `Server.java` , aber natürlich als separate Klasse (einfach nicht in einer separaten Datei).

```
class HalloImpl extends HalloPOA { // Vererbungsmodell
  // Die in IDL definierten Operatione werden hier als Methoden
  // implementiert :
  // wieGehts() und shutdown()
}
```

Die Servant Klasse ist eine Unterklasse von HalloPOA, erbt also alle generellen CORBA Funktionalitäten, welche vom IDL Compiler generiert wurden.

© J.M.Joller

Nun kreieren wir als erstes eine private Variable, orb, welche wir in der Methode `setORB(ORB...)` verwenden. Die `setORB()` Methode ist private und wird eingesetzt, um den ORB eindeutig zu kennzeichnen. Dies wird in der `shutdown()` Methode benötigt (damit wir den korrekten ORB stoppen).

```
  private ORB orb;

  public void setORB(ORB orb_val) {
    orb = orb_val;
  }
```

Nun deklarieren und implementieren wir die `wieGehts()` Methode:

```
public String wieGehts() {
    System.out.println("[HelloImpl.wieGehts()]Aufruf der Servant Methode");
    return "Mir geht's gut! Wie geht's Dir?\n";
  }
```

Und schliesslich implementieren wir die `shutdown()` Methode. Die `shutdown()` Methode ruft `org.omg.CORBA.ORB.shutdown(boolean)` auf. Ein Aufruf von `shutdown(false)` bewirkt, dass der ORB sofort, ohne Warten, heruntergefahren wird.

```
  public void shutdown() {
    orb.shutdown(false);
  }
```

Deklarieren der Server Klasse
Als nächstes deklarieren wir die Server Klasse:

```
public class HelloServer
{
  // Die main() Methode muss hier implementiert werden
}
```

Definieren der `main()` Methode
Die generelle Struktur der `main()` Methode ergibt sich aus den Anforderungen von Java:

```
public static void main(String args[])
  {
    // try-catch Block
  }
```


© J.M.Joller

Behandlung von CORBA Ausnahmen
Alle CORBA Programme können zur Laufzeit CORBA Ausnahmen werfen. Daher umschliessen wir den gesamten Programmcode der `main()` Methode mit einem `try... catch()` Block. CORBA Ausnahmen können durch Mashalling-, UnMarshalling-, Kommunikations-Probleme und vieles mehr geworfen werden.

```
try{
        // hier steht der Rest des Servers
} catch(Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
      }
```

Kreieren und Initialisieren eines ORB Objekts
Jeder CORBA Server und Client benötigt ein lokales ORB Objekt. Jeder Server instanziert einen ORB und registriert seine Servant Objekte, damit der ORB den Server finden kann, falls ein Client ein Servant Objekt verlangt (die Dienste eines Servant Objekts, um genauer zu sein).

Die ORB Variable wird innerhalb des try ... catch Blockes initialisiert:

```
ORB orb = ORB.init(args, null); // init(args, props)
```

Innerhalb der `init()` Methode setzen wir Laufzeiteigenschaften des ORBs (`props`) und Argumente für die `main()` Methode (`args`), welche wir auf der Kommandozeile angeben können.

Bestimmen einer Referenz auf den Root POA und Aktivierung des POA Managers:
Um auf die Dienste des ORBs, beispielsweise den Namensdienst, zugreifen zu können, müssen wir zuerst eine initiale Referenz auf den ORB erhalten. Dies geschieht mit der Methode `resolve_initial_references()`.

Da die POAs hierarchisch strukturiert sind, bestimmen wir eine Referenz auf das Root (zum POA finden Sie im Theorieteil mehr).

```
POA rootpoa = (POA)orb.resolve_initial_references("RootPOA");
```

Schliesslich aktivieren wir den POA Manager.

```
rootpoa.the_POAManager().activate();
```

Die `activate()` Methode ändert den Zustand des POA Managers auf 'aktiv'. Damit steht der POA Manager bereit Anfragen zu bearbeiten. Die Aufgabe des POA Managers ist es, den Zustand der POAs, die im zugeordnet sind, zu kapseln. Zu jedem POA Objekt gehört ein POA Manager. Ein POA Manager kann ein oder mehrere POA Objekte verwalten.

© J.M.Joller

Verwalten des Servant Objekts
Die Aufgabe des Servers ist es, ein oder mehrere Servant Objekte zu instanzieren.
Der Servant, das Servant Objekt, erbt / implementiert die Interfaces, welche aus der IDL Beschreibung generiert wurden und führt schliesslich die Arbeit aus, welche im IDL Interface versprochen wird.
Unser `Server` verwendet `HalloImpl`.

Instanzieren des Servant Objekts
Im `try...catch` Block, nach der Aktivierung des POA Managers, wird das Servant Objekt instanziert:
```
HalloImpl halloImpl = new HalloImpl();
```

Mit der `setORB(ORB...)` Methode wird es möglich sein, gezielt unseren Server herunterzufahren. Dies wurde im IDL Interface in der Operation `shutdown()` verlangt.
```
halloImpl.setORB(orb);
```

Das Herunterfahren könnte auch auf andere Art und Weise implementiert werden. Beispielsweise könnte ein Flag gesetzt werden, welche periodisch abgefragt wird und das Herunterfahren verlangt, falls es einen bestimmten Wert annimmt.

Nun bestimmen wir eine CORBA Objektreferenz auf unser Servant Objekt und casten es zu einem `Hallo` Objekt.
```
org.omg.CORBA.Object ref = rootpoa.servant_to_reference(halloImpl);
Hallo href = HalloHelper.narrow(ref);
```

Mit COS Naming arbeiten
Unser Server arbeitet mit dem Common Object Service (COS) Namensdienst, um das Servant Objekt Kunden anzubieten. Damit der Server dies tun kann, muss er zuerst eine Referenz auf den COS Namensdienst erhalten.

Wir könnten auch andere Techniken verwenden, um die Interfaces und Methoden des Servant Objekts zu publizieren. Eine oft verwendete Technik ist die Darstellung des Servant Objekts als Zeichenkette in einer Datei (Stringifying). Diese Technik finden Sie in den Kursunterlagen zur CORBA Praxis.

JDK1.4 liefert zwei Namensdienste mit:
- `tnameserv` : einen transienten Namensdienst und
- `orbd`: ein Daemon Prozess mit einem Bootstrap Service, einem transienten Namensdienst, einem persistenten Namensdienst und einem Server Manager.
  In unserem Beispiel verwenden wir `orbd`

Bestimmen des initialen Namenskontextes
Im `try catch` Block bestimmen wir eine Referenz auf den Namensraum:
```
org.omg.CORBA.Object objRef =orb.resolve_initial_references("NameService");
```

Die Zeichenkette "NameService" ist für alle CORBA ORBs definiert. Dabei wird der transiente Namensdienst verwendet. Falls Sie die Zeichenkette "PNameService" eingeben, wird (versucht) einen persistenten Namensdienst zu finden und einzusetzen. In unserem Beispiel verwenden wir den transienten Dienst des `orbd`.

© J.M.Joller

Umwandlung der Objektreferenz
CORBA Objektreferenzen sind generische CORBA Objekte. Um sie in unserem Server Programm einsetzen zu können, müssen wir das Objekt in den korrekten Datentyp / Objekttyp umwandeln. Dies geschieht mit der `narrow()` Methode.

```
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
```

Die dabei verwendete Helper Klasse ist ähnlich aufgebaut wie die HalloHelper Klasse, die wir mit den IDL Compiler generiert haben.

Das Objekt `ncRef` ist nun ein `org.omg.CosNaming.NamingContextExt` und kann eingesetzt werden, um auf Namensdienste zuzugreifen und den Server (und die Servant Objekte) anzumelden.
Das `NamingContextExt Objekt` ist eine Erweiterung im JDK 1.4 und Teil der Interoperable Naming Service Spezifikation.

Registrierung des Servant beim Namensdienst
Zur Registrierung unseres Servant Objekts benötigen wir einen Namen und einen Zugriffspfad. Diese definieren wir in den folgenden Zeilen:

```
String name = "POA Grüsse";
NameComponent path[] = ncRef.to_name( name );
```

Mit diesen Angaben können wir nun das Servant Objekt in den Namensraum eintragen:

```
ncRef.rebind(path, href);
```

Damit hat ein Client die Möglichkeit mit einem Aufruf der `resolve("POA Grüsse")` Methode im Initialkontext eine Objektreferenz auf unser Servant Objekt zu finden und zu erhalten.

Warten auf Aufrufe
Nach all diesen Arbeiten ist unser Server bereit Anfragen von Clients entgegen zu nehmen.

```
orb.run();
```

Nach diesem Aufruf steht der Server dem ORB für Anfragen zur Verfügung. Immer nach einem Aufruf wartet der Server auf weitere Anfragen. deswegen haben wir im Client die Möglichkeit eines Shutdowns eingebaut.

1.2.2.3. Übersetzen des Servers

In unserem Fall wurde der Server im JBuilder, mit der passenden JDK Version, übersetzt. Sonst steht auf dem Server / der CD eine Batch Prozedur zur Verfügung:

```
@echo off
Rem
Rem ---------------------------------------------------------------------
set JAVA_HOME=d:\jdk1.4
Rem ---------------------------------------------------------------------
Rem
%JAVA_HOME%\bin\javac -verbose -classpath HalloApp\;.;..
poavererbung\*.java
pause
```

Auch hier ist entscheidend, dass die JDK Version mindestens 1.4 sein muss, da erst ab dieser Version der POS zur Verfügung steht!

© J.M.Joller

1.2.3. Implementation der Client Applikation - Transienter Client

Unser Client Beispiel ist denkbar einfach:
- es kreiert und initialisiert einen ORB
- es bestimmt eine Referenz auf das Root des Naming Contextes
- es sucht das Servant Objekt "POA Grüsse" im Naming Context und bestimmt eine Referenz auf das CORBA Objekt und
- ruft die Methoden wieGehts() und shutdown() des Servant Objektes auf.

Client.java

```
package poavererbung;

/**
 * Title:        CORBA - Portable Object Adapter Vererbungsmodell
 * Description:
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

import HalloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class Client
{
  static Hallo halloImpl;

  public static void main(String args[])
    {
      try{
        // kreiere und initialisiere den ORB
        System.out.println("[POA-Client.main()]Kreiere und initialisiere
den

ORB");
        ORB orb = ORB.init(args, null);

        // bestimme den root naming context
        System.out.println("[POA-Client.main()]bestimme den root naming

context");
        org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
        // Benutze den NamingContextExt anstelle von NamingContext.
        // Dies ist Teil des Interoperable Naming Service.
        System.out.println("[POA-Client.main()]Benutze den

NamingContextExt");
        NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

        // Auflösen der Objekt Referez im Namensraum / Naming
        System.out.println("[POA-Client.main()]Aufloesen der Objekt Referez
                                                               im
Namensraum");
        String name = "POA Grüsse";
```


© J.M.Joller

```
        halloImpl = HalloHelper.narrow(ncRef.resolve_str(name));
        System.out.println("[POA-Client.main()]Bestimmen einer Referenz auf
                                                               das Server
Objekt\n");
        System.out.println(halloImpl+"\n\n");
        System.out.println("[POA-Client.main()]Aufruf der Servant Methode

'wieGehts()'");
        System.out.println("[POA-Client.main()]Antwort des Servants :

"+halloImpl.wieGehts());
        halloImpl.shutdown();

        } catch (Exception e) {
          System.out.println("[POA-Client.main()]Fehler " + e) ;
          e.printStackTrace(System.out);
          }
    }
}
```

1.2.3.1. Übersetzen des Clients

In unserem Fall wurde der Server im JBuilder, mit der passenden JDK Version, übersetzt. Sonst steht auf dem Server / der CD eine Batch Prozedur zur Verfügung:

```
@echo off
Rem
Rem --------------------------------------------------------------------
set JAVA_HOME=d:\jdk1.4
Rem --------------------------------------------------------------------
Rem
%JAVA_HOME%\bin\javac -verbose -classpath HalloApp\;.;..
poavererbung\*.java
pause
```

Auch hier ist entscheidend, dass die JDK Version mindestens 1.4 sein muss, da erst ab dieser Version der POS zur Verfügung steht!

1.2.4. Generierung der Applikation

Als erstes muss die IDL Beschreibung übersetzt werden. Dabei wird Java Quellcode generiert. Diesen müssen wir übersetzen, bevor die Client / Server Applikation generiert werden kann, da beide, Client und Server, die generierten Klassen importieren. Sie finden alle Batch Prozeduren auf dem Server / der CD, wichtig ist lediglich die Reihenfolge einzuhalten und eventuell den Pfad auf JDK, JAVA_HOME passend zu setzen. Dies geschieht in der Batch Prozedur UmgebungSetzen.bat.

© J.M.Joller

1.2.5. Starten der Applikation

Obschon das Beispiel sehr einfach ist, kann man daran eigentlich alles einer typischen CORBA Applikation erkennen, speziell in diesem Fall, eines statischen Methodenaufrufes, statisch, weil die Details des Servant Objekts im voraus bekannt sind. Falls das Interface nicht bekannt ist, spricht man von einer dynamischen Nutzung des Servant Objekts. Sie finden mehr darüber im Abschnitt über die grundlegenden CORBA Konzepte.

1.2.5.1. Starten des Name Service

Das Beispiel benötigt auch einen Namensservice, mit dessen Hilfe CORBA Objekte benannt und wieder gefunden werden. Ab JDK1.4 haben Sie die Wahl:
- entweder Sie verwenden den tnameserv, einen transienten / temporären Namenservice oder
- orbd, einen ORB Daemon, welcher einen Bootstrap Dienst, einen transienten / temporären Namensdient, einen Persistenzdienst und einen Server Manager enthält.

'Temporär' besagt in diesem Zusammenhang, dass die Speicherungen nicht in einer Datei oder einer Datenbank gesichert werden.

In unserem Beispiel verwenden wir den orbd, den neuen ORB Daemon. Als Standard-Port verwenden wir den Port 1024. Dieser kann beim Starten des Daemons beliebig gesetzt werden, beispielsweise auf 1050, mit der ORBInitialPort Option:

`-ORBInitialPort 1050`

Und so wird der Namenservice gestartet, mit Hilfe der Batch Prozedur:

```
@echo off
Rem
Rem -----------------------------------------------------------------------
call UmgebungSetzen.bat
Rem -----------------------------------------------------------------------
Rem
echo Der Name Server wird an Port 1050 (ORBInitialPort) gestartet
echo Der Server kann nur mit CTRL/C gestoppt werden
%JAVA_HOME%\bin\orbd -ORBInitialPort 1050 -ORBInitialHost localhost
pause
```

Der Port muss angegeben werden, falls Sie den ORB Daemon einsetzen wollen, sonst stürzt der Server ab!

© J.M.Joller

1.2.5.2. Starten des Servers

Für das Starten des Servers steht folgende Batch Prozedur zur Verfügung:

```
@echo off
Rem
Rem ----------------------------------------------------------------------
call UmgebungSetzen.bat
Rem ----------------------------------------------------------------------
Rem
@echo Starten des CORBA Servers...
Rem
%JAVA_HOME%\bin\java -cp .;..;..\HalloApp; poavererbung.Server -
ORBInitialPort 1050
pause
```

1.2.5.3. Starten des Clients

Für das Starten des Clients steht folgende Batch Prozedur zur Verfügung:

```
@echo off
Rem
Rem ----------------------------------------------------------------------
call UmgebungSetzen.bat
Rem ----------------------------------------------------------------------
Rem
@echo Starten des CORBA Clients mit POA Vererbung...
%JAVA_HOME%\bin\java -cp .;..;..\HalloApp; poavererbung.Client "Es war
einmal ..." -ORBInitialPort 1050
cd ..
pause
```


© J.M.Joller

1.2.5.4. Beispielausgabe des Servers

Hier eine mögliche Ausgabe des Servers:

[POA-Server.main()]Kreieren und Initialisieren des ORB
[POA-Server.main()]Bestimmen einer Referenz auf die rootpoa & aktivieren des POAManager
[POA-Server.main()]Kreieren des Servants und registrieren beim ORB
[POA-Server.main()]Bestimmen einer Objektreferenz zum Servant
[POA-Server.main()]Bestimme den root naming context
[POA-Server.main()]Einsatz des NamingContextExt
[POA-Server.main()]Binden der Objekt Referenz im Namensraum
[POA-Server.main()]Der Server ist bereit und wartet auf Anfragen ...
[HelloImpl.wieGehts()]Aufruf der Servant Methode
[HelloImpl.shutdown()]Aufruf der Servant Methode
[POA-Server.main()]Der Server wird herunter gefahren ...

1.2.5.5. Beispielausgabe des Clients

Hier eine mögliche Ausgabe des Clients:

POA-Client.main()]Kreiere und initialisiere den ORB
[POA-Client.main()]bestimme den root naming context
[POA-Client.main()]Benutze den NamingContextExt
[POA-Client.main()]Aufloesen der Objekt Referez im Namensraum
[POA-Client.main()]Bestimmen einer Referenz auf das Server Objekt

HalloApp._HalloStub:IOR:000000000000001749444c3a48616c6c6f4170702f48616c6c6f3a3
12e30000000000001000000000000000680001020000000000a3132372e302e302e310004a3000
00021afabcb00000000202d11d8d200000001000000000000000000000004000000000300000
0000000010000000100000020000000000000100010000000205010001000100200001010900
00000100010100

[POA-Client.main()]Aufruf der Servant Methode 'wieGehts()'
[POA-Client.main()]Antwort des Servants : Mir geht's gut! Wie geht's Dir?

© J.M.Joller

1.3. Server-seitiges POA Modell für Persistenz

In diesem Abschnitt befassen wir uns mit dem gleichen Programm wie in der Einführung; allerdings verwenden wir Policies des POA, so dass der Server und der Namensdienst persistent werden. Die Änderungen werden wir speziell markieren, da sie eher gering sind!

1.3.1. Transiente und Persistente Objekte

Ein persistentes CORBA lebt genau so lange bis es explizit zerstört wird. Im Gegensatz dazu kann im persistenten Fall ein Client mit einer Referenz auf ein persistentes CORBA Objekt diese Referenz benutzen, selbst wenn der Objektserver nicht gestartet wurde, da der ORB Daemon, ORBD, den Server starten wird, falls eines der vom Server angebotenen Objekte aufgerufen wird.

Ab J2SE 1.4 umfasst JavaIDL einen Portable Objekt Adapter (POA), den Object Request Broker Daemon (ORBD) und das Server-Tool. Mit Hilfe dieser Werkzeuge und features kann ein persistenter Objekt-Server gebaut werden.

Im Gegensatz dazu besitzt ein transientes CORBA Objekt genau die selbe Lebensdauer wie der anbietende Server. Sobald der Server beendet wird, verschwindet das transiente Objekt, zudem werden alle Objektreferenzen darauf ungültig.

Transiente Objekte können für asynchrone Kommunikation sehr wertvoll sein. Ein Beispielablauf könnte folgendermassen aussehen:

- eine Applikation gilt als Brücke zwischen zwei anderen Applikationen
- die Brückenapplikation wird nur im Falle der Kommunikation benötigt, nicht früher und nachher auch nicht mehr (sie kann aber jederzeit wieder generiert werden).
- in disem Fall macht es kaum Sinn, den Server / das Serverobjekt dauernd laufen zu lassen und die zu kommunizierenden Daten permanent zu speichern.

Eine solche Anordnung erinnert an Callback Systeme.

in diesem Abschnitt wollen wir

- ein IDL Interface für eine persistente Anwendung definieren.
  Dieses sieht völlig gleich aus, wie im transienten Fall im einführenden Beispiel.
- einen persistenten Server bauen, welcher in der Lage ist, ein Objekt in einem Namensraum zu publizieren, mittels einer Server-seitigen Implementation (POA).
- einen Servant bauen, welcher die im IDL beschriebenen Operationen als Methoden implementiert.
- einen Anwendungsclient schreiben, welcher den Objektnamen kennt, eine Refernz darauf aus dem Namensdienst bestimmt und das Objekt einsetzt, indem er dessen Methoden aufruft.
- zeigen, wie diese Applikation übersetzt und gestartet werden kann.

© J.M.Joller

1.3.2. Definition des Interfaces

Als erstes müssen wir wieder die Objekte mittels Interfaces, in IDL beschrieben, spezifizieren.

Das folgende IDL Programm beschreibt ein Objekt, welches zwei Methoden besitzt, genua wie unser einführendes Beispiel. Auf der Stufe IDL sehen Sie keinerlei Unterschiede, ob Sie nun das persistente oder das transiente oder sonst ein Modell implementieren.

PersistentesHallo.idl

```
module Persistent
{
  interface Hallo
  {
  string wieGehts();
  oneway void shutdown();
  };
};
```

Auch hier müssen Sie wieder darauf achten, dass der Name des Interfaces und des Moduls unterschiedlich sind!

In JBuilder haben Sie, sofern Sie die Enterprise Edition installiert haben, die Möglichkeit, den IDL Wizard zu verwenden. Allerdings müssen Sie dafür zuerst die Enterprise Edition installieren und konfigurieren, unter Tools beispielsweise angeben, welchen ORB Sie einsetzen (Visibroker, OrbixWeb und eigene).

1.3.2.1. Übersetzen des Interfaces

Um die aus der IDL Beschreibung generierten Java Klassen praktisch in der Client, Servant und Server Implementation einsetzen zu können, müssen wir den IDLJ Compiler aufrufen.

Im JBuilder (zumindest der Enterprise Edition) können Sie einfach die IDL Beschreibung mit 'Build' oder 'Make' in Java übersetzen, sofern Ihre Enterprise Edition korrekt konfiguriert ist. Dabei werden die Dateien gemäss den Anforderungen des vorgegebenen ORBs generiert: falls Sie JDK 1.4 einsetzen wollen, müssen Sie unbedingt diesen ORB konfigurieren, oder mit Batch Dateien arbeiten.

Beachten Sie, dass falls Sie die IDL Beschreibung ins JBuilder projekt aufnehmen, die Hilfsklassen des bevorzugten ORBs, also nicht jene für JDK 1.4 generiert werden.

Der ORB kann mit einer Property Datei gesetzt werden. Falls Sie beispielsweise VisiBroker installieren, wird im lib Verzeichnis der JRE die Datei ORB.properties angelegt. Diese wird beim Starten mit `java` gelesen, muss also, falls vorhanden, Ihren ORB beschreiben!

1.3.2.2. Generierte Klassen

Falls Sie das BAT Skript einsetzen, werden die selben Dateien generiert wie im einführenden Beispiel. Falls Sie einen anderen ORB einsetzen, werden vermutlich weitere, weniger oder andere Dateien generiert. Die entsprechende ORB Dokumentation wird Ihnen bei der Interpretation weiterhelfen.

© J.M.Joller

1.3.3. Implementation des Servants

Unser persistenter Servant unterscheidet sich nicht vom transienten, den wir im einführenden Beispiel realisiert haben. Der Servant muss die definierten Interfaces implementieren.

PersistenterServant.java

```
//package persistenz;

import org.omg.CORBA.ORB;

/**
 * Title:        Serverseitige Persistenz in CORBA mit POA
 * Description:  Einfaches Beispiel für den Einsatz des POA mit Policy
 * @author J.M.Joller
 * @version 1.0
 */

public class PersistenterServant extends Persistent.HalloPOA {
    private ORB orb;

    public PersistenterServant( ORB orb ) {
        this.orb = orb;
        //System.out.println("[PersistenterServant]Konstruktor(orb)");
    }

    /**
     *  wieGehts() Methode : einfache Antwort auf eine Anfrage
     */
    public String wieGehts( ) {
        System.out.println("[PersistenterServant]wieGehts()");
        return "Mir geht's gut! Wie geht's Dir?...";
    }

    /**
     *  shutdown() Methode zum Herunterfahern des ORBs
     */
    public void shutdown( ) {
        System.out.println("[PersistenterServant]shutdown()");
        orb.shutdown( false );
    }
}
```

Bemerkung

Der Einfachheit halber haben wir die Shutdown Methode in das Servant Objekt geschoben. In der Regel wird dies nicht sinnvoll sein, da sonst jeder Client den ORB herunter fahren könnte.

Dieser Shutdown ist etwas trickreich:

- falls die `orb.shutdown()` Methode mit dem Parameter `true` ("wait for completion") innerhalb einer remote Methode aufgerufen wird , wird der ORB einen Deadlock produzieren, also sich aufhängen. Andere Threads könnten aber `orb.shutdown()` ohne Probleme aufrufen.
- falls Sie mehrere Clients haben und einer ruft `shutdown(false)` auf, dann werden alle Clients Probleme haben.
- korrekterweise sollte die `orb.shutdown(false)` Methode im Server implementiert werden.

© J.M.Joller

1.3.4. Implementation des Servers

Zuerst das Schema:

unser persistenter Server besitzt wie jedes Java Hauptprogramm eine `main()` Methode, welche

- eine ORB Instanz kreiert.
- ein Servant Objekt kreiert.
- eine Referenz auf das POA Root bestimmt.
- die Policy kreiert, welche aus dem Server einen persistenten Server macht.
- den POA kreiert, basierend auf der persistenten Policy.
- den `POAManager` aktiviert.
- dem persistenten POA den Servant zuordnet.
- eine Objektreferenz auf den Namenskontext bestimmt, um das neue CORBA Objekt zu registrieren.
- die Objektreferenz mit der `narrow()` Methode in einen Namenskontext umwandelt.
- das neue Objekt im Namenskontext unter dem Namen "Persistente Variante" registriert.
- auf Kundenanfragen wartet.

PersistenterServer.java

```
//package persistenz;

import java.util.Properties;
import org.omg.CORBA.Object;
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NameComponent;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
import org.omg.CORBA.Policy;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.*;
import org.omg.PortableServer.Servant;

/**
 * Title:        Serverseitige Persistenz in CORBA mit POA
 * Description:  Einfaches Beispiel für den Einsatz des POA
 *               unter Berücksichtigung der Persistenz des Servers
 *               und des Namensdienstes
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class PersistenterServer {

// optinaler Installer (das Server Tool startet diesen)
    public static void install(org.omg.CORBA.ORB orb) {
      System.out.println("Install wurde gestartet");
    }
```


© J.M.Joller

```
    public static void main( String args[] ) {
        System.out.println("[PersistenterServer]Start");
        Properties properties = System.getProperties();
        properties.put( "org.omg.CORBA.ORBInitialHost","localhost" );
        properties.put( "org.omg.CORBA.ORBInitialPort", "1050" );
        System.out.println("[PersistenterServer]Properties:");
        System.out.println("[PersistenterServer]\tORBInitialHost :
"+properties.getProperty("org.omg.CORBA.ORBInitialHost"));
        System.out.println("[PersistenterServer]\tORBInitialPort :
"+properties.getProperty("org.omg.CORBA.ORBInitialPort"));
        try {
            // 1) Instanzieren des ORBs
            System.out.println("[PersistenterServer]ORB.init()");
            ORB orb = ORB.init(args, properties);

            // 2) Instanzieren des Servants
            PersistenterServant servant = new PersistenterServant(orb);
System.out.println("[PersistenterServer]new PersistenterServant(orb)");

            // 3) Servant mit persistentem POA
            // ******************
            // 3-1) rootPOA
            POA rootPOA = (POA)orb.resolve_initial_references("RootPOA");

System.out.println("[PersistenterServer](POA)orb.resolve_initial_references
('RootPOA')");

            // 3-2) Persistent Policy
            System.out.println("[PersistenterServer]POA Policy");
            Policy[] persistentPolicy = new Policy[1];
            persistentPolicy[0] = rootPOA.create_lifespan_policy(
                LifespanPolicyValue.PERSISTENT);
            // 3-3) Kreiere POA mit der obigen Policy
            POA persistentPOA = rootPOA.create_POA("persistenterPOA", null,
                persistentPolicy );
            // 3-4) Aktiviere den POAManager des persistenten POA
            // Falls dieser Schritt fehlt, bleibt der persistente Server
            // stehen,
            // weil der POAManager im 'HOLD' Zustand stecken bleibt.
            persistentPOA.the_POAManager().activate( );
            // ********************** Ende 3)

            // 4) Assoziiere den Servant mit dem persistenten POA
            persistentPOA.activate_object( servant );

System.out.println("[PersistenterServer]POA.activate_object(servant)");

            // 5) Auflösen des RootNaming Kontextes und binden eines Namens
            // für das Servant Objekt
            /** HINWEIS:
             *  Falls der Server persistent ist, dann sollte auch ein
             *  persistenter Namensdienst eingesetzt werden.
             *  Im Falle von ORBD ist die Wahl von 'NameService' in der
             *  Methode resolve_initial_reference() eine gute Wahl.
             */
            System.out.println("[PersistenterServer]Naming Context");
            org.omg.CORBA.Object obj = orb.resolve_initial_references(
                "NameService" );
       NamingContextExt rootContext = NamingContextExtHelper.narrow( obj );

            NameComponent[] nc = rootContext.to_name(
```


© J.M.Joller

```
                "PersistenterServer" );
            rootContext.rebind( nc, persistentPOA.servant_to_reference(
                servant ) );

            // 6) Warten auf Kundenanfragen
System.out.println("[PersistenterServer]orb.run() - warten auf Clients");
            orb.run();
        } catch ( Exception e ) {
System.err.println( "Exception : Starten des persistenten Servers" + e );
        }
    }
}
```

1.3.4.1. Übersetzen des Servers

Das Übersetzen geschieht analog zum einführenden Beispiel, entweder im JBuilder, mit JDK1.4 oder höher, oder mit dem Batch Skript. Allerdings muss zuerst die Servant Klasse implementiert sein!

Das Server Tool ist im Moment noch recht fehlerhaft!
Beispielsweise kann es passieren, dass eine Server Klasse mit Package Namen nicht installiert (persistent gemacht) werden kann.

Auch ORBD ist noch etwas wackelig. Aber im Vergleich zum Server Tool recht stabil!

© J.M.Joller

1.3.5. Implementation der Clients Applikation

Der generelle Aufbau dieses Clients sieht folgendermassen aus:

- kreieren und initialisieren eines ORBs
- bestimmen des Links zum Servant Objekt, mittels einer CORBA URL und dem Interaoperablen Namensdienst / Service (INS). Die Vorgabe lautet: `localhost` und Port `1050`. Mit diesem Namensdienst wird der Servant unter dem Namen `PersistenterServer` gesucht.
- aufrufen der Methoden des Servant Objekts (`wieGehts()` und `shutdown()`) in einer Endlosschleife. Das heisst, dass der (bei ORBD registrierte) Server dauernd hoch- und runter gefahren wird.

PersistenterClient.java

```
//package persistenz;

import java.util.Properties;
import org.omg.CORBA.ORB;
import org.omg.CORBA.OBJ_ADAPTER;
import org.omg.CosNaming.NamingContext;
import org.omg.CosNaming.NamingContextHelper;
import org.omg.CosNaming.NameComponent;
import org.omg.PortableServer.POA;

import Persistent.*;

/**
 * Title:        Serverseitige Persistenz in CORBA mit POA
 * Description:  Einfaches Beispiel für den Einsatz des POA
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class PersistenterClient {

    public static void main(String args[]) {

        try {
            // 1) Instanzieren des ORBs
            ORB orb = ORB.init(args, null);

            // 2) Auflösen des Servant Objekts über eine URL
            // Der Namensdienst muss gestartet sein und
            // der Host 'localhost', der Port 1050 sein
            // 'PersistenteVariante' steht im Namensraum (durch den Server)
            org.omg.CORBA.Object obj = orb.string_to_object(
                "corbaname::localhost:1050#PersistenterServer");

            Hallo persistentHallo = HalloHelper.narrow( obj );

            // 3) Aufruf der wieGehts() Methode alle 60 Sekunden und
            //    Shutdown
            // Der Server wird automatisch gestartet, da er als persistent
            // definiert wurde.
            while( true ) {
System.out.println("[PersistenrerClient]Aufruf des persistenten Servers" );
```


© J.M.Joller

```
                String meldungVomServer = persistentHallo.wieGehts();
System.out.println("[PersistenrerClient]Message vom persistenten Server: "
+
                    meldungVomServer );
System.out.println( "[PersistenrerClient]Herunterfahren des persistenten
Servers.." );
                persistentHallo.shutdown( );
                Thread.sleep( 60000 );
            }
        } catch ( Exception e ) {
System.err.println( "[PersistenrerClient]Exception im persistenten
Client..." + e );
            e.printStackTrace( );
        }
    }
}
```

1.3.5.1. Übersetzen des Clients

Diese Applikation wurde (teilweise) im JBuilder entwickelt. Zudem steht Ihnen ein Batch Skript zum Übersetzen zur Verfügung, genau wie im einführenden Beispiel.

Auch hier ist wichtig zu beachten, dass entweder ein ORB installiert wird, welcher POA unterstützt, oder aber JDK1.4 oder neuer.

Beachten Sie die Hinweise beim Server und der IDL Beschreibung:

1. Server mit Packages lassen sich zum Teil mit dem Server Tool nicht korrekt in den ORB Daemon (`orbd`) eintragen.
2. IDL Beschreibungen, welche im JBuilder Projekt drin sind, werden von dem entsprechenden ORB Hilfsprogramm in Java übersetzt (beispielsweise `vidl2java`) und sind damit nicht kompatibel zu JDK1.4 und höher.

© J.M.Joller

1.3.6. Starten der Applikation

Bevor Sie Ihre Applikation starten müssen Sie unbedingt sicher sein, dass Sie keinen Standard ORB gesetzt haben, der sich von dem in der Applikation verwendeten unterscheidet. Sie finden die Angaben zum Standard ORB in der Datei
`%JAVA_HOME%\jre\lib\orb.properties`.

Falls Sie beispielsweise den Inprise Aplikations-Server installiert haben, zusammen mit JBuilder Enterprise und (automatisch) dem VisiBroker, finden Sie in diesem Verzeichnis in dieser Datei folgende Beschreibung:

```
%JAVA_HOME%\jre\lib\orb.properties
# Make VisiBroker for Java the default ORB
org.omg.CORBA.ORBClass=com.inprise.vbroker.orb.ORB
org.omg.CORBA.ORBSingletonClass=com.inprise.vbroker.orb.ORB
```

1. Starten des ORB Daemons:
   `orbd -ORBInitialPort 1050 -serverPollingTime 200`

   Die Port Angabe bezieht sich auf den Namensdienst;
   die Polling Zeit bezieht sich auf jenes Intervall, in dem der ORB Daemon nachfragt, ob der mittels Server Tool registrierte Server (siehe unten) noch aktiv ist. Standardwert ist 1000, also eine Prüfung alle Sekunden. Wie haben einen tieferen Wert gewählt, um sicherzustellen, dass unsere Client Applikation korrekt funktioniert (Endlosschleife). Falls der Daemon einen Fehler registriert, startet er den Server automatisch von Neuem.

   Damit Sie besser beobachten können, was der ORB Daemon so alles anstellt, starten wir ihn im Debug Modus: `-ORBDebug orbd`. Damit wird jeder Aufruf von und jede Antwort vom `orbd` sichtbar.

2. Starten des Servers:
   Um einen persistenten Server zu starten, muss beim Einsatz von ORBD der Server mit dem Server Tool (servertool.exe) registriert, entfernt und gestoppt werden.

   a) Beim **Starten des Server Tools** müssen Sie den ORB Port und den Host angeben. Dabei müssen diese Angaben mit jenen aus dem ORB Daemon übereinstimmen.

   In unserem Fall sieht dies folgendermassen aus:
   `servertool -ORBInitialPort 1050 -ORBInitialHost localhost`

   Dann erscheint ein einfacher DOS Prompt, Ihr Eingabefenster für die Server Tool Befehle. Falls Sie Hilfe benötigen können Sie mit `Help` eine Liste der Befehle ausgeben lassen und mit `Help <Befehl>` die Befehlssyntax.

   b) Um unseren Server persistent zur Verfügung zu haben, müssen wir ihn mit **dem Server Tool registrieren**

   Dazu steht der register Befehl im Server Tool zur Verfügung:

© J.M.Joller

Der Befehl benötigt folgende Angaben:
- den Servernamen
- einen fakultativen Applikationsnamen
- den Classpath zur Server Klasse (ohne Package!)
- fakultativ Angaben von Argumenten für den Server oder dessen JVM

Um nicht dauernd die Zeit mit Tippen zu vergeuden, steht Ihnen ein Batch Skript zur Verfügung:

```
echo off
Rem
Rem --------------------------------------------------------------------
-
call UmgebungSetzen.bat
Rem --------------------------------------------------------------------
-
Rem
@echo Starten des Server Tools...
Rem
Rem steht in der Umgebung: set JAVA_HOME=d:\jdk1.4
%JAVA_HOME%\bin\servertool -ORBInitialPort 1050 < servertool.txt
pause
```

In der Datei servertool.txt stehen die Server Tool Befehle. Sie müssen diese Ihrer Umgebung anpassen (`Classpath`):

servertool.txt

```
register -server PersistenterServer -applicationName PersistenteVariante
-classpath
"D:\UnterrichtsUnterlagen\ParalleleUndVerteilteSysteme\CORBAmitPOA\Beisp
iele\POAPersistent\Persistenz"
quit
```


© J.M.Joller

Der ORB Daemon meldet sich und gibt dem Server Tool eine ServerID zurück. Falls der Server bereits früher einmal registriert wurde, wird die `serverid=0` zurück gebenen (der Server ist bereits registriert und in eine 'Datenbank' eingetragen).

Die bestehenden Applikationen können Sie auch mit dem Server Tool auflisten lassen:

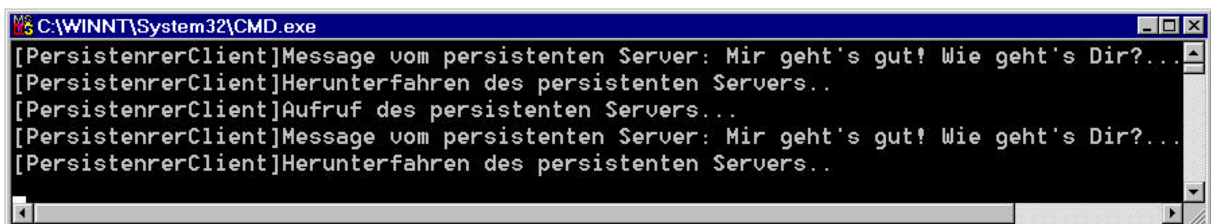
3. Nun müssen wir nur noch die Client Applikation starten:
   `java -classpath . PersistenterClient`

   oder mit dem Batch Skript:

   3StartenDesCORBAClients.bat

```
@echo off
Rem
Rem -------------------------------------------------------------------
-
call UmgebungSetzen.bat
Rem -------------------------------------------------------------------
-
Rem
@echo Starten des CORBA Clients mit POA Vererbung...
cd Persistenz
Rem set JAVA_HOME=d:\jdk1.4
%JAVA_HOME%\bin\java -cp .;..;Persistenz\Persistent; PersistenterClient
"Es war einmal ..." -ORBInitialPort 1050
cd ..
pause
```

   Beispielausgabe:



© J.M.Joller

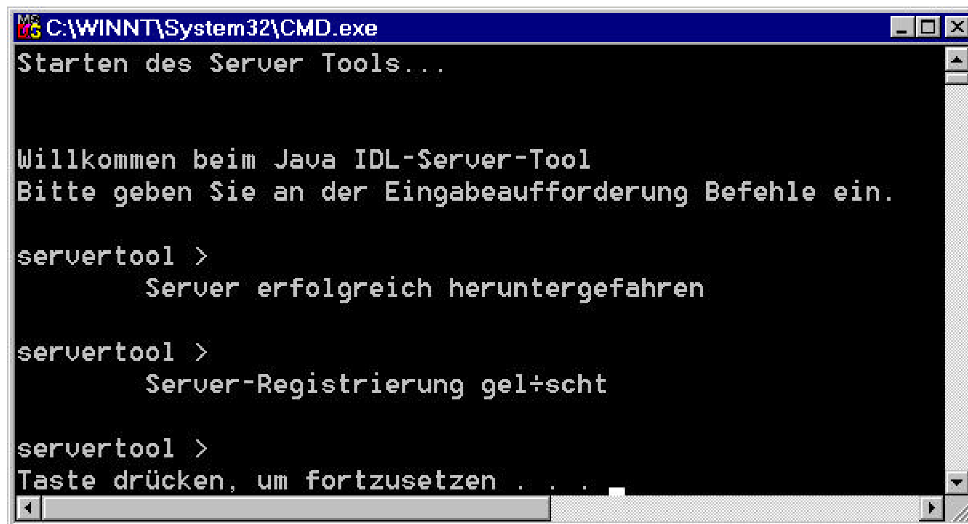
4. Stoppen des Servers und Entfernen des Eintrages in der Persistenz-DB
   In unserem einfachen Beispiel können wir den Client abstürzen lassen, also einfach den Server anhalten und aus der DB entfernen. Falls Sie den Eintrag nicht entfernen, können Sie bei einem erneuten Start, nach dem Start des ORB Daemons gleich den Client starten, da alle Informationen über den Server permanent in einer Datei abgeseichert wurden.

   Dafür steht Ihnen wieder ein Batch Skript zur Verfügung:

   4StoppenUndEntfernenDesServers.bat

```
echo off
Rem
Rem --------------------------------------------------------------------
-
call UmgebungSetzen.bat
Rem --------------------------------------------------------------------
-
Rem
@echo Starten des Server Tools...
Rem
Rem set JAVA_HOME=d:\jdk1.4
%JAVA_HOME%\bin\servertool -ORBInitialPort 1050 <
stoppenundentfernen.txt
pause
```

   Falls Sie den Server nur herunterfahren, wird der Client in der nächsten Schleife den Server automatisch erneut starten!



© J.M.Joller

1.4. Serverseitiges POA Modell mit transientem Server

Dieser Fall ist identisch mit dem einführenden Beispiel. Der wesentliche Unterschied zum persistenten Fall ist das Fehlen einer POA Policy und damit das einfachere Starten des Clients und des Servers: ohne Server Tool, da keinerlei persistente Einträge gemacht werden müssen.

1.5. POA Tie (Delegation) Modell mit transientem Server

Kurz zur Wiederholung:
- wir unterscheiden grundsätzlich zwischen
  1) Vererbungsmodellen und
  2) Delegationsmodellen (`Tie` oder `Tie`-Delegationsmodell)
  für CORBA Applikationen.
- innerhalb der Vererbungsmodelle unterscheidet man weiter
  1) OMG POA basierte Modelle
  2) `ImplBase` basierte Modelle (veraltet)

Das `Tie` Modell delegiert die Aufrufe in die Implementationsklassen. Sie benötigen zwei durch IDL generierte Klassen:
- die `Tie` Klasse (oder ImplBase Klassen)
- die `xxxOperations` Klasse

1.5.1. Definition des Interfaces

Unsere IDL Beschreibung ist völlig identisch mit jener aus den vorangehenden Beispielen. Auch in diesem Fall sollten Sie darauf achten die Namensgebung so zu wählen, dass Sie Modulnamen und Interface unterscheiden können.

DelegationsHallo.idl
```
module Delegation{
  interface Hallo
  {
  string wieGehts();
  oneway void shutdown();
  };
};
```


© J.M.Joller

1.5.1.1. Übersetzen des Interfaces

Die Generiereung der Java Klassen geschieht mit Hilfe des folgenden Batch Skripts:

```
@echo off
Rem
Rem ------------------------------------------------------------------
call UmgebungSetzen.bat
Rem ------------------------------------------------------------------
Rem
cd Delegation
@echo Generieren der Java Klassen
%JAVA_HOME%\bin\idlj -v -fall -td . DelegationsHallo.idl
@echo Generieren der Java Tie Klassen
%JAVA_HOME%\bin\idlj -v -fallTie -td . DelegationsHallo.idl
@echo Die generierten Dateien stehen im Unterverzeichnis

Delegation\Delegation
@echo Uebersetzen der generierten Java Dateien
%JAVA_HOME%\bin\javac Delegation\*.java
cd ..
pause
```

Beachten Sie, dass der IDL Compiler zweimal aufgerufen wird:
- das erste Mal zur Generierung der üblichen Hilfsklassen
- das zweite Mal zur Generierung der Tie Klassen

1.5.1.2. Generierte Klassen

Neben den üblichen Klassen, die generiert werden:
- `HalloPOA.java`
  eine abstrakte Klasse, welche als Server Skeleton dient und die Grundlage schafft für eine Server Implementation
- `_HalloStub.java`
  dem Client Stub, welcher die grundlegenden Funktionalitäten von CORBA für den Client zur Verfügung stellt
- `Hallo.java`
  der Java Version der IDL Interface Beschreibung
- `HalloHelper.java`
  einer Hilfsklasse zur Verwaltung der CORBA Objektreferenzen
- `HalloHolder.java`
  einer Klasse, welche eine öffentliche Instanz eines Hallo Objekts enthält (zur Behandlung der INOUT, IN und OUT Parameter bei IDL Operationen)
- `HalloOperations.java`
  einem Interface für die in IDL beschriebenen Operationen

© J.M.Joller

finden wir neu (wegen der zweiten Generierung mit der Tie Option)

- `HalloPOATie.java`
  welche den Delegationsmechanismus implementiert. Sie sehen im Programmcode , wie ein Parameter im Konstruktor (die Delegationsklasse bzw. das Delegationsobjekt) beim Methodenaufruf dazu aufgefordert wird, die Arbeit zu erledigen:

```
package DelegationsModell;
/**
* DelegationsModell/HalloPOATie.java
* Generated by the IDL-to-Java compiler (portable), version "3.1"
* from DelegationsHallo.idl
* Montag, 27. August 2001 15.36 Uhr CEST
*/

public class HalloPOATie extends HalloPOA
{

  // Constructors

  public HalloPOATie ( DelegationsModell.HalloOperations delegate ) {
      this._impl = delegate;
  }
  public HalloPOATie ( DelegationsModell.HalloOperations delegate ,
org.omg.PortableServer.POA poa ) {
      this._impl = delegate;
      this._poa      = poa;
  }
  public DelegationsModell.HalloOperations _delegate() {
      return this._impl;
  }
  public void _delegate (DelegationsModell.HalloOperations delegate ) {
      this._impl = delegate;
  }
  public org.omg.PortableServer.POA _default_POA() {
      if(_poa != null) {
          return _poa;
      }
      else {
          return super._default_POA();
      }
  }
  public String wieGehts ()
  {
    return _impl.wieGehts();
  } // wieGehts

  public void shutdown ()
  {
    _impl.shutdown();
  } // shutdown

  private DelegationsModell.HalloOperations _impl;
  private org.omg.PortableServer.POA _poa;

} // class HalloPOATie
```


© J.M.Joller

1.5.2. Implementation des Servants

Der Servant besitzt pro IDL Operation eine Methode, wie gehabt. Der Aufbau unterscheidet sich nicht von jenem in den vorhergehenden Beispielen, einzig die Methode `setORB()` wurde in diesem Beispiel wieder eingesetzt..

```
package poatie;

import POATieModell.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

import java.util.Properties;

/**
 * Title:        POA Tie Modell
 * Description:  Delegationsmodell, mit Package Angaben
 * Copyright:    Copyright (c) J.M.Joller
 * @author J.M.Joller
 * @version 1.0
 */

public class DelegationsServantImpl extends HalloPOA{
    private ORB orb;

    // angepasst für Delegationsmodell
    public void setORB( org.omg.CORBA.ORB orb_val) {
        System.out.println("[DelegationsServant]setORB()");
        this.orb = orb_val;
        //System.out.println("[PersistenterServant]Konstruktor(orb)");
    }

    /**
     *  wieGehts() Methode implementiert eine einfache Antwort auf eine
Anfrage
     */
    public String wieGehts( ) {
        System.out.println("[DelegationsServant]wieGehts()");
        return "Mir geht's gut! Wie geht's Dir?...";
    }

    /**
     *  shutdown() Methode zum Herunterfahern des ORBs
     */
    public void shutdown( ) {
        System.out.println("[DelegationsServant]shutdown()");
        orb.shutdown( false );
    }
}
```

1.5.2.1. Übersetzen des Servants

Dieses geschieht innerhalb des Jbuilders oder mit einem Batch Skript, in meinem Fall innerhalb des JBuilders.

© J.M.Joller

1.5.3. Implementation des Servers

In unserem Beispiel besteht der Server aus zwei Klassen, dem *Servant* und dem *Server*.

Die *Server* Klasse besitzt eine `main()` Methode, welche:

- eine ORB Instanz kreiert und initialisiert.
- eine Referenz auf das Root POA bestimmt und den `POAManager` aktiviert.
- ein Tie Objekt mit dem Servant als Delegate kreiert.
- eine CORBA Objektreferenz für den Namenskontext bestimmt, in dem das neue CORBA Objekt registriert wird.
- das Root des Namenskontextes bestimmt.
- das neue Objekt im Namenskontext unter dem Namen " `DelegationsVariante`" registriert.
- auf Aufrufe der Methoden des neuen Objekts durch Clients wartet.

DelegationsServer.java

```
package poatie;

import POATieModell.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

import java.util.Properties;


/**
 * Title:        POA Tie Modell
 * Description:  Delegationsmodell, mit Package Angaben
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class DelegationsServer{

  public static void main(String args[]){
    System.out.println("[DelegationsServer]Start");
    try{
      // kreieren und initialisieren des ORBs
      System.out.println("[DelegationsServer]ORB.init()");
      ORB orb = ORB.init(args, null);

      // Referenz auf rootpoa & aktivieren des POAManager
      System.out.println("[DelegationsServer]Referenz auf rootPOA");
      POA rootpoa = (POA)orb.resolve_initial_references("RootPOA");
      System.out.println("[DelegationsServer]aktivieren des POAManagers");
      rootpoa.the_POAManager().activate();

      // Servant kreieren und beim ORB registrieren
System.out.println("[DelegationsServer]Servant kreieren und registrieren");
      DelegationsServantImpl halloImpl = new DelegationsServantImpl();
      halloImpl.setORB(orb);
```


© J.M.Joller

```
      // kreieren des Tie; der Servant ist das Delegate Objekt
      System.out.println("[DelegationsServer]Delegation kreieren");
      HalloPOATie tie = new HalloPOATie(halloImpl, rootpoa);

      // objectRef für den tie bestimmen
      // und implizites Aktivieren des Objekts
      Hallo href = tie._this(orb);

      // bestimme root naming context
      System.out.println("[DelegationsServer]Namenskontext");
      org.omg.CORBA.Object objRef =
                          orb.resolve_initial_references("NameService");

      // nutze den NamingContextExt(Teil des Interoperable
      // Naming Service).
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

      // binden der Objekt Referenz im Namensraum
      System.out.println("[DelegationsServer]binden des Servants");
      String name = "DelegationsVariante";
      NameComponent path[] = ncRef.to_name( name );
      ncRef.rebind(path, href);

System.out.println("[DelegationsServer]Der Server ist bereit und wartet auf
Anfragen ...");

      // warten auf Client Anfragen
      orb.run();
      }

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

    System.out.println("[DelegationsServer]Der Server wird heruntergefahren
...");

  }
}
```

1.5.3.1. Übersetzen des Servers

Auch der Server wird direkt im JBuilder übersetzt. Sie können aber auch das Batch Skript einsetzen.

© J.M.Joller

1.5.4. Implementation der Client Applikation

Der generelle Aufbau dieses Clients sieht wie immer folgendermassen aus:

- kreieren und initialisieren eines ORBs
- bestimmen des Links zum Servant Objekt, mittels einer CORBA URL und dem Interaoperablen Namensdienst / Service (INS). Die Vorgabe lautet: `localhost` und Port `1050`. Mit diesem Namensdienst wird der Servant unter dem Namen `DelegationsVariante` gesucht.
- aufrufen der Methoden des Servant Objekts (`wieGehts()` und `shutdown`

DelegationsClient.java

```
package poatie;

import POATieModell.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;


/**
 * Title:        POA Tie Modell
 * Description:  Delegationsmodell, mit Package Angaben
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class DelegationsClient{

  public static void main(String args[]){
      System.out.println("[DelegationsClient]Start");

    try{
      // kreiere und initialisiere den ORB
      System.out.println("[DelegationsClient]ORB.init()");
      ORB orb = ORB.init(args, null);

      // bestimme den root naming context
      System.out.println("[DelegationsClient]Namingkontext Root
bestimmen");
      org.omg.CORBA.Object objRef =
        orb.resolve_initial_references("NameService");

      // verwende NamingContextExt statt NamingContext.
      // Dieser ist Teil des Interoperable Naming Service.
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

      // auflösen der Objekt Referenz im Namensraum
      System.out.println("[DelegationsClient]Servant suchen");
      String name = "DelegationsVariante";
      Hallo delegationsHalloImpl =
                             HalloHelper.narrow(ncRef.resolve_str(name));

System.out.println("[DelegatiosnClient]Bestimme Referenz auf den Servant: "
+ delegationsHalloImpl);
      System.out.println(delegationsHalloImpl.wieGehts());
```


© J.M.Joller

```
        System.out.println("[DelegationsClient]Server shutdown()");
        delegationsHalloImpl.shutdown();
        }

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

1.5.4.1. Übersetzen des Clients

Auch diesen Schritt habe ich direkt im JBuilder durchgeführt. Das Batch Skript habe ich nicht getestet. Aber zum Übersetzen aller Java dateien im Package kann ein einziges Skript eingesetzt werden:

```
@echo off
Rem
Rem ---------------------------------------------------------------------
call UmgebungSetzen.bat
Rem ---------------------------------------------------------------------
Rem
%JAVA_HOME%\bin\javac -verbose -classpath POATieModell\;.;.. poatie\*.java
pause
```

1.5.5. Starten der Applikation

Dieses Beispiel funktioniert ähnlich wie das einführende beispiel. Sie können also der Reihe nach, wie im Verzeichnis bei den Batch Skripts angegeben:

1. den Namensservice starten
2. den Server starten
3. und schliesslich den Client, der auch gleich den Server wieder herunterfährt.

Falls Sie den Server zu schnell starten, kann es vorkommen, dass der Namensdienst noch nicht voll gestartet wurde.

Namensdienst

© J.M.Joller

Server

```
C:\WINNT\System32\CMD.exe
Starten des Servers...
[DelegationsServer]Start
[DelegationsServer]ORB.init()
[DelegationsServer]Referenz auf rootPOA
[DelegationsServer]aktivieren des POAManagers
[DelegationsServer]Servant kreieren und registrieren
[DelegationsServant]setORB()
[DelegationsServer]Delegation kreieren
[DelegationsServer]Namenskontext
[DelegationsServer]binden des Servants
[DelegationsServer]Der Server ist bereit und wartet auf Anfragen ...
[DelegationsServant]wieGehts()
[DelegationsServant]shutdown()
[DelegationsServer]Der Server wird heruntergefahren ...
Taste drücken, um fortzusetzen . . .
```

Client

```
C:\WINNT\System32\CMD.exe
Starten des CORBA Clients mit POA Vererbung...
[DelegationsClient]Start
[DelegationsClient]ORB.init()
[DelegationsClient]Namingkontext Root bestimmen
[DelegationsClient]Servant suchen
[DelegatiosnClient]Bestimme Referenz auf den Servant: DelegationsModell._HalloStub:IOR:00
02049444c3a44656c656761746 96f6e734d6f64656c6c2f48616c6c6f3a312e3000000000001000000000000000
000000a3132372e302e302e310004430000021afabcb000000000204225b52500000000100000000000000000
0000300000000000000010000000100000020000000000000100010000000205010001000100200001010900000
Mir geht's gut! Wie geht's Dir?...
[DelegationsClient]Server shutdown()
Taste drücken, um fortzusetzen . . .
```


© J.M.Joller

1.6. ImplBase (Vererbung) basierter transienter Server

Das ImplBase Modell wurde durch das POA Modell ersetzt. Es steht aber immer noch zur Verfügung, speziell aus Kompatibilitätsgründen.

Für neue Entwicklungen sollten Sie aber dieses Modell nicht mehr einsetzen!

1.6.1. Definition des Interfaces

Unser Interface unterscheidet sich auch in diesem Fall nicht von den vorangehenden, einzig im Modulnamen:

ImplBaseHallo.idl

```
module ImplBaseModell{
  interface Hallo
  {
  string wieGehts();
  oneway void shutdown();
  };
};
```

Achten Sie darauf, dass der Modulnamen und das Interface nicht gleich heissen!

1.6.1.1. Übersetzen des Interfaces

Für diese Aufgabe steht Ihnen ein Batch Skript zur Verfügung:

```
@echo off
Rem
Rem ---------------------------------------------------------------------
call UmgebungSetzen.bat
Rem ---------------------------------------------------------------------
Rem
@echo Generieren der Java Klassen
%JAVA_HOME%\bin\idlj -v -fall -oldImplBase -td . ImplBaseHallo.idl
@echo Die generierten Dateien stehen im Unterverzeichnis ImplBaseModell
@echo Uebersetzen der generierten Java Dateien
%JAVA_HOME%\bin\javac ImplBaseModell\*.java
cd ..
pause
```


© J.M.Joller

1.6.1.2. Generierte Dateien

Die generierten Dateien unterscheiden sich zum Teil wesentlich von jenen der anderen Modelle. Aber die Grundfunktionen bleiben die selben:

- `_HalloImplBase`
  eine abstrakte Klasse, das Server Skeleton, mit der CORBA Funktionalität. Die Server Klasse erbt die in dieser Datei beschriebene Klasse.

- `_HalloStub`
  stellt dem Client die CORBA Funktionalität zur Verfügung.

- `Hallo.java`
  ist die Java Version des IDL Interfaces

- `HalloHelper.java`
  stellt zusätzlich benötigte Funktionalitäten (`narrow`...) zur Verfügung.

- `HalloHolder.java`
  enthält wie immer eine öffentliche Kopie eines Hello Objekts, um die ganzen IN, INOUT und OUT Parameter aus dem IDL korrekt handhaben zu können.

- `HalloOperations.java`
  enthält die Interfaces aus dem IDL bzw. die Methode.

© J.M.Joller

1.6.2. Implementation des Servants

Der Servant besitzt pro IDL Operation eine Methode, wie gehabt. Der Aufbau unterscheidet sich nicht von jenem in den vorhergehenden Beispielen, einzig die Methode `setORB()` wurde in diesem Beispiel wieder eingesetzt..

ImplBaseModell.java

```
package poaimplbase;

import ImplBaseModell.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

import java.util.Properties;

/**
 * Title:        Verebungbasiertes CORBA Modell
 * Description:  ImplBase basiertes CORBA Beispiel
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class ImplBaseServant extends _HalloImplBase {
    private ORB orb;

    // angepasst für Delegationsmodell
    public void setORB( org.omg.CORBA.ORB orb_val) {
        System.out.println("[ImplBaseServant]setORB()");
        this.orb = orb_val;
        //System.out.println("[ImplBaseServant]Konstruktor(orb)");
    }

    /**
     *  wieGehts() Methode implementiert eine einfache Antwort auf eine
Anfrage
     */
    public String wieGehts( ) {
        System.out.println("[ImplBaseServant]wieGehts()");
        return "Mir geht's gut! Wie geht's Dir?...";
    }

    /**
     *  shutdown() Methode zum Herunterfahern des ORBs
     */
    public void shutdown( ) {
        System.out.println("[ImplBaseServant]shutdown()");
        orb.shutdown( false );
    }
}
```


© J.M.Joller

1.6.3. Implementation des Servers

Die *Server* Klasse besitzt eine `main()` Methode, welche:

- eine ORB Instanz kreiert und initialisiert.
- eine Instanz des Servants kreiert und dies dem ORB mitteilt.
- eine CORBA Objektreferenz für den Namenskontext bestimmt, in dem das neue CORBA Objekt registriert wird.
- das Root des Namenskontextes bestimmt.
- das neue Objekt im Namenskontext unter dem Namen " ImplBaseVariante" registriert.
- auf Aufrufe der Methoden des neuen Objekts durch Clients wartet.

Beachte Sie einmal mehr, dass dieses Modell veraltet ist und durch das POA Modell ersetzt wurde. Damit wir die Java IDL CORBA Implementation OMG kompatibel!

Die Unterschiede im Programmcode zur verherigen Lösung sind denkbar gering. Im Hintergrund geschieht allerdings einiges völlig anders.

ImplBaseServer.java

```
package poaimplbase;

import ImplBaseModell.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

import java.util.Properties;

/**
 * Title:        Verebungbasiertes CORBA Modell
 * Description:  ImplBase basiertes CORBA Beispiel
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class ImplBaseServer{

  public static void main(String args[]){
    System.out.println("[ImplBaseServer]Start");
    try{
      // kreieren und initialisieren des ORBs
      System.out.println("[ImplBaseServer]ORB.init()");
      ORB orb = ORB.init(args, null);

      // kreieren des Servants
      System.out.println("[ImplBaseServer]Servant kreieren");
      ImplBaseServant implBaseHallo = new ImplBaseServant();
      implBaseHallo.setORB(orb);

      // bestimme root naming context
      System.out.println("[ImplBaseServer]Namenskontext");
      org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
```


© J.M.Joller

```
      NamingContext ncRef = NamingContextHelper.narrow(objRef);

      Hallo href = HalloHelper.narrow(implBaseHallo);

      // binden der Objekt Referenz im Namensraum
      System.out.println("[ImplBaseServer]binden des Servants");
      String name = "ImplBaseVariante";
      NameComponent nc = new NameComponent(name,"");
      NameComponent path[] = {nc};
      ncRef.rebind(path, href);

      System.out.println("[ImplBaseServer]Der Server ist bereit und wartet
auf Anfragen ...");

      // warten auf Client Anfragen
      orb.run();
      }

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

    System.out.println("[ImplBaseServer]Der Server wird heruntergefahren
...");

  }
}
```

1.6.3.1. Übersetzen des Servers

Das generieren der Class Dateien geschieht entweder im JBuilder oder mit dem Batch Skript (für alle Java Dateien).

© J.M.Joller

1.6.4. Implementation der Client Applikation

Unsere einfache Client Applikation funktioniert wie die anderen Beispiele:

- zuerst wird ein ORB kreiert
- dann wird eine Referenz auf den Namensraum bestimmt
- und schliesslich die Objektreferenz unter dem vorgegebenen Namen bestimmt.

ImplBaseClient.java

```
package poaimplbase;

import ImplBaseModell.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;


/**
 * Title:        Verebungbasiertes CORBA Modell
 * Description:  ImplBase basiertes CORBA Beispiel
*/

public class ImplBaseClient {
  static Hallo implBaseHallo;

  public static void main(String args[]){
      System.out.println("[ImplBaseClient]Start");

    try{
      // kreiere und initialisiere den ORB
      System.out.println("[ImplBaseClient]ORB.init()");
      ORB orb = ORB.init(args, null);

      // bestimme den root naming context
      System.out.println("[ImplBaseClient]Namingkontext Root bestimmen");
      org.omg.CORBA.Object objRef =
        orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(objRef);

      // bestimmen der Objekt Referenz im Namensraum
      System.out.println("[ImplBaseClient]suchen des Servants");
      String name = "ImplBaseVariante";
      NameComponent nc = new NameComponent(name,"");
      NameComponent path[] = {nc};

      Hallo implBaseHallo = HalloHelper.narrow(ncRef.resolve(path));

System.out.println("[ImplBaseClient]Bestimme Referenz auf den Servant: " +
implBaseHallo+"\n");
      System.out.println(implBaseHallo.wieGehts());
      System.out.println("[ImplBaseClient]Server shutdown()");
      implBaseHallo.shutdown();
      }
    catch (Exception e) {
      System.out.println("ERROR : " + e) ;
      e.printStackTrace(System.out);
    }
  }
}
```


© J.M.Joller

1.6.4.1. Übersetzen der Client Applikation

Dieser Schritt geschieht wieder entweder im JBuilder oder aber mit dem allgemeinen Java Übersetzungsskript, angepasst auf unsere Verhältnisse (Pfade).

```
@echo off
Rem
Rem ---------------------------------------------------------------------
call UmgebungSetzen.bat
Rem ---------------------------------------------------------------------
Rem
%JAVA_HOME%\bin\javac -verbose -classpath ImplBaseModell\;.;..

poaimplbase\*.java
pause
```

1.6.5. Starten der Applikation

Der Ablauf zum Starten dieser Applikation ist genau gleich, auch die Ausgabe, wie im vorherigen Beispiel:

1. starten des Namensdienstes
2. starten des Servers
3. starten des Clients

1.7. Verteilen der Klassen - Beispiele mit mehreren Rechnern

Da die meisten von Ihnen kaum die Möglichkeit haben, ein eigenes lokales Netzwerk aufzubauen, werden wir im Folgenden 'trocken' beschreiben, wie die Verteilung der Klassen zu geschehen hätte.

Das Vorgehen:

1. kreieren Sie wie in den Beispielen eine IDL Beschreibung
2. übersetzen Sie diese, gemäss einem der besprochenen Modelle (mit einer oder zwei IDL Stufen [POA transient / persistent, Tie, ImplBase]) auf der Client Maschine
3. kreieren Sie den Client auf der Client Maschine,
   importieren Sie Stubs und Skeletons, wie in den Beispielen.
4. kreieren Sie den Server auf der Server Maschine. Dieser benötigt die aus dem IDL generierten Dateien ebenfalls (ausser dem Stub).
5. Starten Sie den Objekt Request Broker Daemon (`orbd`) auf der Server Maschine
   `start orbd -ORBInitialPort 1050 -ORBInitialHost` *`servermachinename`*
6. Starten Sie den Server auf der Server Maschine
   `java HelloServer -ORBInitialPort 1050`

7. Nun starten Sie den Client auf der Client Maschine
   `java HelloClient -ORBInitialHost` *`nameserverhost`* `-ORBInitialPort 1050`

8. Falls die Applikation beendet ist, sollten Sie den Daemon stoppen.

© J.M.Joller

1.8. Neuere Features

Folgende neue Möglichkeiten stehen ab J2SE1.4 zur Verfügung:
- POA - der Portable Objekt Adapter
- der Portable Interceptor
- INS - der Interoperable Naming Service
- GIOP - das General InterORB Protokoll

Schauen wir uns kurz an, um was es dabei geht!

1.8.1. Der Portable Object Adapter

1.8.1.1. Was ist der POA?

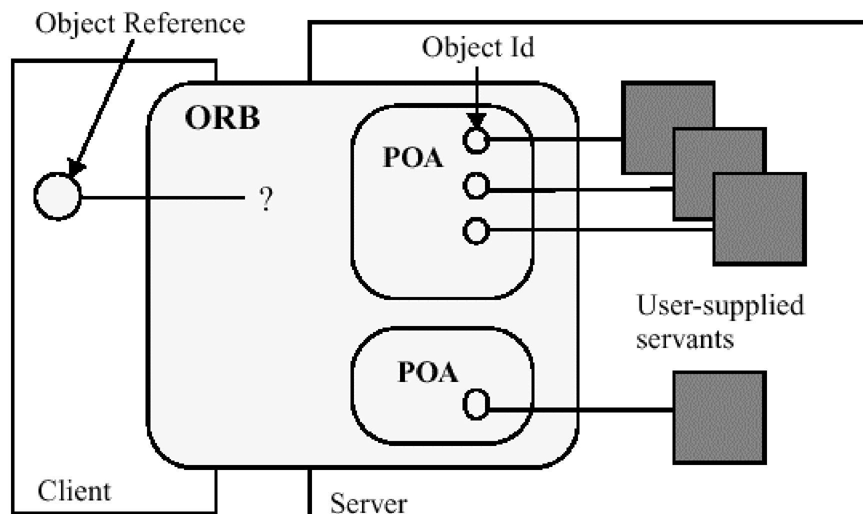
Ein Objekt Adapter ist ein Mechanismus, welcher eine Anfrage mittels einer Objektreferenz mit dem korrekten Objekt für diese Referenz verbindet. Ein POA ist ein spezieller Objektadapter, welcher in der CORBA Spezifikation definiert wurde. Mit POA sollen folgende Ziele erreicht werden:

- die Objektimplementationen sollen portabel zwischen unterschiedlichen ORBs sein.
- Persistenz soll unterstützt werden.
- die transparente Aktivierung von Objekten soll möglich sein.
- ein einzelner Servant soll in der Lage sein, mehrere Objektidentitäten gleichzeitig zu unterstützen.

POA wird in Kapitel 11 der CORBA 2.3.1 Specification
http://cgi.omg.org/cgi-bin/doc?formal/99-10-07 beschrieben

1.8.1.2. Das abstrakte POA Modell

Die CORBA Spezifikation beschreibt die Architektur des abstrakten Modells, welches durch POA impliziert wird, sowie die Wechselwirkung zwischen den unterschiedlichen Komponenten.



© J.M.Joller

Der ORB ist eine Abstraktion, welche sowohl Client-seitig, als auch Server-seitig sichtbar ist. Der POA ist ein Objekt, welches Server-seitig sichtbar ist. Die vom Benutzer implementierten Objekte werden beim POA registriert, wie sehen wir gleich.

Die Clients kreieren bzw. bestimmen Referenzen, mit deren Hilfe Anfragen getätigt werden können.

Der ORB. der POA und die Implementationsobjekte arbeiten eng zusammen, um beispielsweise festzustellen, welcher Servant benötigt wird und welche Methoden ausgeführt werden sollen.

Die obige Skizze versucht dieses Wechselspiel etwas zu verdeutlichen:
ein POA befasst sich mit einer Objekt-ID und einem aktiven Servant. Aktiver Servant besagt, dass das entsprechende Objekt im Adressraum, im Speicher der JVM ist und dem POA bekannt ist und einem oder mehreren Objekt-Ids. Diese Zuordnungen können auf unterschiedliche Art und Weise realisiert werden.

Falls der POA die RETAIN Policy unterstützt, wird der POA ein Verzeichnis aktiver Objekte unterhalten. In diesem werden den Objekten fixe Objekt-Ids zugeordnet.

Falls der POA die USE_DEFAULT_SERVANT Policy unterstützt, kann ein Standard Servant mit diesem POA registriert / verknüpft werden.

Falls der POA die USE_SERVANT_MANAGER Policy unterstützt, kann ein vom Benutzer erstellter und dem POA zur Verfügung gestellter Servant Manager mit dem POA verknüpft werden.

Falls die Active Object Map nicht eingesetzt wird, oder aber eine Anfrage für ein Servant Objekt an den POA gelangt, welches dieser nicht kennt, kann der POA entweder den Standard-Servant einsetzen, oder aber den Servant Manager beauftragen, den passenden Servant zu beschaffen / zu aktivieren.Falls die RETAIN Policy eingesetzt wird, wird das Servant Objekt, welches vom Servant Manager stammt, in die Active Object Map eingetragen. Sonst wird der Servant lediglich für die aktuelle Anfrage eingesetzt.

Jedes Objekt kann lediglich einfach im Active Object Map vorhandne sein.

© J.M.Joller

1.8.1.3. Einsatz des POA

Die folgenden Schritte werden typischerweise im POA Lebenslauf durchlaufen.

1. Bestimmen des root POA
2. definieren der POA Policies
3. kreieren des POA
4. aktivieren des POA Managers
5. Aktivierung des Servants (inklusive Aktivierung eines allfälligen Ties / Delegates)
6. kreieren der Objektreferenz

1.8.1.4. Bestimmen des root POA

Als erstes muss der erste POA bestimmt werden, root POA genannt. Dieser wird durch den ORB verwaltet und dient sozusagen dem Bootstrapping, mit einem initialen Namen "RootPOA".

Das RootPOA Objekt erhält man auf folgende Art und Weise:

```
ORB orb = ORB.init( args, null );
      POA rootPOA = (POA)orb.resolve_initial_references("RootPOA");
```

1.8.1.5. Definieren der POA Policies

Der Portable Objekt Adapter (POA) ist so entworfen worden, dass er von mehreren ORB Implementationen eingesetzt werden kann, mit minimalem Anpassungsaufwand zur Anpassung an die unterschiedlichen Produkte.

Der POA unterstützt auch persistente Objekte - aus Client Sicht wenigstens. Aus Clientsicht sind die Servant Objekte im persistenten Fall immer verfügbar und lebendig und speichern die in ihnen enthaltenen Daten auch dann, wenn der Server aus was für Gründen auch immer, wiederholt gestartet werden muss.

Der POA gestattet einem Objekt-Implementierer viele unterschiedliche Kontrollmechanismen: über die Objekt Identität, die Speicherung und den Lebenszyklus. Aber Sie können auch einen POA ohne Policy kreieren. In diesem Fall werden folgende Standardwerte verwendet:

- Thread Policy: ORB_CTRL_MODEL
- Lifespan Policy: TRANSIENT
- Object Id Uniqueness Policy: UNIQUE_ID
- Id Assignment Policy: SYSTEM_ID
- Servant Retention Policy: RETAIN
- Request Processing Policy: USE_ACTIVE_OBJECT_MAP_ONLY
- Implicit Activation Policy: IMPLICIT_ACTIVATION

Falls Sie Policies einsetzen wollen, sollten Sie auf das Originaldokument zugreifen!

© J.M.Joller

Im weiter unten besprochenen Beispiel für den Einsatz der POA im Rahmen des RMI über IIOP könnten folgendermassen Policies definiert werden:

```
Policy[] tpolicy = new Policy[3];
tpolicy[0]=rootPOA.create_lifespan_policy(LifespanPolicyValue.TRANSIENT);
tpolicy[1]=rootPOA.create_request_processing_policy(RequestProcessingPolicy
                                       Value.USE_ACTIVE_OBJECT_MAP_ONLY );
tpolicy[2]=rootPOA.create_servant_retention_policy(
                                      ServantRetentionPolicyValue.RETAIN);
```

Die einzelnen Policies werden wir gleich genauer anschauen. Im Detail wird POA und die Policies im 11. Kapitel der CORBA Spezifikation beschrieben:
Chapter 11, *Portable Object Adapter* of the CORBA/IIOP 2.3.1 Specification
http://cgi.omg.org/cgi-bin/doc?formal/99-10-07

1.8.1.5.1. Thread Policy

Die Thread Policy spezifiziert das Modell, mit dem der POA kreiert wird. Standardwert ist ORB_CTRL_MODEL.

Folgende Werte sind möglich:

- ORB_CTRL_MODEL - der ORB ist für die Zuordnung der Anfragen zu Thread verantwortlich; der ORB kontrolliert somit die Zuordnung der POAs zu Threads.

- SINGLE_THREAD_MODEL - es existiert lediglich ein Thread und falls mehrere Anfragen eintreffen, werden diese der Reihe nach bearbeitet, sequentiell.

1.8.1.5.2. Lifespan Policy

Diese Policy spezifiziert die Lebensdauer einer Objektimplementierung im POA. Standardwert ist TRANSIENT.

Folgende Werte sind möglich:

- TRANSIENT - das (Servant) Objekt kann lediglich solange leben, wie der POA. Zudem muss zuerst der POA kreiert werden
- PERSISTENT - das (Servant) Objekt im POA (welches dem POA zugeordnet ist) kann länger leben als der Prozess, welcher dieses Objekt im POA kreierte. Mit anderen Worten, das Objekt muss permanent gespeichert werden.

1.8.1.5.3. Object Id Uniqueness Policy

Diese Policy gibt an, ob jedes Servant Objekt, welches vom POA aktiviert wird, eine eindeutige Objekt ID besitzen muss. Standardeinstellung ist UNIQUE_ID.

Mögliche Werte:

- UNIQUE_ID - Servants, welche vom POA aktiviert wurden, können maximal eine Objekt ID besitzen
- MULTIPLE_ID - Ein POA aktivierter Servant unterstützt ein oder mehrere Objekt ID's.

© J.M.Joller

1.8.1.5.4. Id Assignment Policy

Diese Policy gibt an, ob die Objekt ID's im POA durch die Applikation oder den ORB vergeben werden. Standardwert ist SYSTEM_ID.

Mögliche Werte:

- USER_ID - die Objekte erhalten ihre ID lediglich durch die Applikationen.

- SYSTEM_ID - der POA vergibt die Objekt ID's. Falls die Objekte auch noch persistent sein sollen, muss die Objekt ID eindeutig sein.

1.8.1.5.5. Servant Retention Policy

Diese Policy gibt an, ob der POA eine Active Object Map unterhält, also eine Liste der aktiven Objekte. Standardwert ist RETAIN.

Mögliche Werte sind:

- RETAIN - der POA unterhält eine Active Object Map.
- NON_RETAIN - es existiert keine durch den POA unterhaltene Active Object Map.

1.8.1.5.6. Request Processing Policy

Diese Policy spezifiziert, wie Anfragen vom POA bearbeitet werden. Standardwert ist USE_ACTIVE_OBJECT_MAP_ONLY.

Mögliche Werte:

- USE_ACTIVE_OBJECT_MAP_ONLY - falls das Objekt nicht im Active Object Map gefunden wird, wird eine OBJECT_NOT_EXIST Exception vom Client geworfen. Zudem muss RETAIN gesetzt sein.
- USE_DEFAULT_SERVANT - falls die Objekt ID nicht im Active Object Map gefunden wird, oder die NON_RETAIN Policy gesetzt ist und ein Standard-Servant definiert wurde, wird die Anfrage vom Standard-Servant erledigt.
- USE_SERVANT_MANAGER - falls die Objekt ID in der Active Object Map nicht gefunden wird, oder die NON_RETAIN Policy gesetzt wurde und ein Servant Manager beim POA angemeldet wurde, wird versucht, den Servant zu finden. Sonst wird eine Exception geworfen.

1.8.1.5.7. Implicit Activation Policy

Diese Policy gibt an. ob implizites Aktivieren durch den POA unterstützt wird. Standardwert ist IMPLICIT_ACTIVATION.

Folgende Werte sind möglich:

- IMPLICIT_ACTIVATION - implizites Aktivieren des Servants ist möglich, sofern die SYSTEM_ID und die RETAIN Policies gesetzt wurden.
- NO_IMPLICIT_ACTIVATION - es ist kein implizites Aktivieren möglich.

© J.M.Joller

1.8.1.6. Kreieren des POA

Beim Kreieren eines neuen POAs kann eine spezifische Policy für den neuen POA angegeben werden. Zudem kann ein Adapter Activator und ein Servant Manager (callback Objekte, mit denen Objekte und verschachtelte POAs auf Verlangen aktiviert werden können) angegeben werden. Zudem können Sie mittels Objekt ID's den Objektraum unterteilen, da die Objekt ID's den POAs zugeordnet sind. Zudem kann der Entwickler mittels spezieller POAs die Kontrolle über Objektgruppen gezielt definieren.

Ein POA wird als Blatt eines POA Baumes kreiert. Die Wurzel (das root) ist dabei vorgegeben. Die Methode zum Kreieren eines POAs ist

```
org.omg.PortableServer.POAOperations.create_POA(...)

public POA create_POA(String adapter_name, POAManager a_POAManager,
       Policy[] policies) throws AdapterAlreadyExists, InvalidPolicy
```

Sie benötigen folgende Angaben, um einen neuen POA kreieren zu können:

- der Name des POA:
  jeder POA besitzt einen eindeutigen Namen innerhalb eines Baumes, zum Beispiel `childPOA`.
- POA Manager:
  Angabe des POAManagers, der dem neuen POA zugeordnet ist. Falls `null` als Parameter verwendet wird, wird ein neuer POA Manager kreiert.
- Policy List:
  Angabe eines Policy Objektes, welches dem POA zugeordnet ist und dessen Verhalten kontrolliert.

Hier ein Beispiel aus einem der vorangehenden Beispiele (Persistent Server):

```
// 3-3) Kreieren eines POAs mit einer Persistenzpolicy.

POA persistentPOA = rootPOA.create_POA("childPOA", null,
   persistentPolicy );
```

1.8.1.7. Aktivieren des POAManager

Jeder POA besitzt einen ihm zugeordneten POA Manager, welcher den Zustand des POAs überwacht und beispielsweise regelt, was mit Anfragen geschieht, falls der POA bereits besetzt ist (Warteschlange, ...). Der POA Manager kann auch einen POA deaktivieren. Einem POA Manager kann ein oder mehrere POAs zugeordnet sein.

Der POA Manager besitzt mehrere Zustände:

- HOLD / HOLDING:
  in diesem Zustand wartet der POA auf eintreffende Anfragen.
- ACTIVE:
  in diesem Zustand bearbeitet der POA eine Anfrage.
- DISCARDING:
  der POA lässt eintreffende Anfragen unberücksichtigt.
- INACTIVE:
  neue und noch nicht gestartete Anfragen werden zurück gewiesen.

© J.M.Joller

Weitere Informatione finden Sie in der API Dokumentation zu `org.omg.PortableServer.POAManagerPackage` zur Klasse `State`.

Ein neu kreierter POA wird nicht automatisch aktiv. Das folgende Programmfragment zeigt, wie der POA Manager aktiviert wird. Das Beispiel stammt aus dem Persistenten Server Beispiel. Standardzustand des POA Managers ist HOLD

```
// Aktivieren des POA Managers
persistentPOA.the_POAManager().activate( );
```

1.8.1.8. Aktivierung des Servants

Die folgenden Informationen stammen aus der CORBA Spezifikation:

Ein CORBA Objekt kann (muss aber nicht) einem aktiven Servant Objekt zugeordnet sein. Falls der POA mit der RETAIN Policy kreiert wurde, werden die Objekt ID's der Servant Objekte in die Active Object Map des POAs eingetragen. Diese Aktivierungsart kann auf folgende Arten realisiert werden:

- die Server Applikation aktiviert die individuellen Objekte mit Hilfe einer der Methoden

  ```
  org.omg.PortableServer Interface POAOperations
  activate_object(Servant servant_obj)
  ```

  oder

  `activate_object_with_id``(`byte[] id, Servant p_servant)

- die Server Applikation instruiert den POA, wie das Objekt zu aktivieren ist, falls Anfragen eintreffen. Diese Aktivierung geschieht mit einem benutzerspezifischen Servant Manager, der also auch noch erstellt werden muss.
- falls die IMPLICIT_ACTIVATION Policy gesetzt wurde, und die Sprachbindung dies erlaubt, kann der POA implizit ein Objekt aktivieren, sofern die Server Applikation eine Referenz auf das Objekt benötigt und das Servant Objekt noch nicht aktiv ist (also noch keine Objekt ID für das gwsuchte Objekt existiert).
- falls die Policy USE_DEFAULT_SERVANT gesetzt wurde, kann die Server Applikation den POA beauftragen den Standardservant zu aktivieren, unabhängig von der Objekt ID. Dieser Servant muss vorgängig vom Server mit der Methode

  ```
  void set_servant(Servant servant_obj)
  ```

  spezifiziert werden.
- Falls der POA mit einer NON_RETAIN kreiert wurde, kann der POA bei Anfragen entweder den Standardservant oder aber einen Servant-Manager einsetzen. Aus sicht des POA ist in diesem Fall der Servant lediglich für eine Anfrage aktiviert worden und ist anschliessend nicht mehr bekannt.^, da keine Active Object Map existiert.

Das folgende Programmfragment aus dem Persistenten Server Beispiel zeigt den Einsatz der `activate_object()` Methode :

```
// 4) Servant einem persistenten POA zuordnen
persistentPOA.activate_object( servant );
```


© J.M.Joller

Falls man RMI-IIOP oder die Tie / Delegations Technik einsetz, basiert die Implementation auf einem Interface. Falls Sie in diesem Falle ein Servant Objekt kreieren müssen Sie auch ein Tie Objekt zum CORBA Interface kreieren.

Das folgende Programmfragment stammt aus einem RMI IIOP Programm (siehe unten) und setzt voraus, dass der POA mit einer USE_ACTIVE_OBJECT_MAP_ONLY Policy kreiert wurde.

```
_HalloImpl_Tie tie = (_HalloImpl_Tie)Util.getTie( halloImpl );
String halloId = "hello";
byte[] id = halloId.getBytes();
tPOA.activate_object_with_id( id, tie );
```

Sie finden mehr Details zu diesem Beispiel im Abschnitt über "RMI over IIOP mit POA". Zusätzliche Informationen erhalten Sie auch aus der CORBA 2.3.1 Spezifikation.

1.8.1.9. Kreieren einer Objektreferenz

Objektreferenzen werden im Server kreiert. Sobald sie kreiert sind, werden sie zum Client exportiert. Die Objektreferenzen kapseln Objektinformationen und Informationen, welche vom ORB benötigt werden, um den Server und den dazugehörigen POA zu finden.

Objektreferenzen können auf zwei Arten kreiert werden:

- ein Servant wird explizit aktiviert und ihm eine Objektreferenz zugeordnet. Das folgende Programmfragment zeigt wie dies im Persistenten Server Beispiel gemacht wird:
  die Methode

  ```
  Object servant_to_reference(Servant p_servant)
  ```

  vom `Interface POAOperations` (im `Package org.omg.PortableServer`) liefert eine Objektreferenz bei gegebenem Servant, sofern dieser Active Object Map gesteuert wird:

  ```
  // 5) Auflösen des Root Naming Services und binden des Servants
  org.omg.CORBA.Object obj=orb.resolve_initial_references("NameService" );
  NamingContextExt rootContext = NamingContextExtHelper.narrow( obj );
  NameComponent[] nc=rootContext.to_name(   "PersistentServerTutorial" );
  rootContext.rebind( nc, persistentPOA.servant_to_reference(   servant )
  );
  ```

- die Server Applikation kann auch direkt eine Referenz kreieren.
  Das folgende Programmfragment stammt aus dem RMI oder IIOP Beispiel weiter hinten.

  ```
  // 4) Publizieren der Objektreferenz mit der selben Objekt ID
  //    wie beim Aktivieren des Tie Objekts
  Context initialNamingContext = new InitialContext();
  initialNamingContext.rebind("HelloService",
  tPOA.create_reference_with_id(id,
    tie._all_interfaces(tPOA,id)[0]) );
  Server application causes a servant to implicitly activate itself.
  ```

  sofern die IMPLICIT_ACTIVATION Policy gesetzt wurde (die Standard Policy).

Sobald eine Objekt Referenz existiert, kann sie dem Client zur Verfügung gestellt werden. Sie finden Details zu diesem Mechanismus im Kapitel 11.2.3 der CORBA Spezifikation.

© J.M.Joller

1.8.2. Adapter Activators

Adapter Activatoren sind optional. Activators sind dann sinnvoll, wenn POAs während Anfragen kreiert werden müssen. Falls die POAs bereits existieren wenn die Applikation ausgeführt werden soll, werden Activators nicht benötigt.

Ein Adapter Activator liefert einen POA mit der Fähigkeit weitere Unter-POAs zu kreieren, falls dies nötig wird.

Bei einer Anfrage an den ORB wird dieser zuerst einen Server suchen oder starten. In diesem wird dann der POA benötigt. Falls der POA nicht existiert, kann die Applikation den benötigten POA mit dem Activator kreieren. Falls dies nicht gelingt, wird eine OBJECT_NOT_EXIST Exception geworfen. Erst wenn der passende POA vorhanden ist, kann der ORB die Anfrage an diesen weiterleiten.

Details zu diesem Mechanismus finden Sie im Kapitel 11.3.3 der CORBA Spezifikation: Adapter Activator Operations API.

1.8.3. Servant Managers

Auch Servant Managers sind optional. Einen Servant Manager kann man immer dann einsetzen, wenn ein POA die Möglichkeit haben sollte, einen Servant auf Verlangen zu aktivieren, falls Anfragen für den Servant eintreffen. Falls ein Server alle Objekte gleich beim Starten lädt, wird kein Servant Manager benötigt!

Ein Servant Manager ist ein Callback Objekt, welcher der Applikationsentwickler einem POA zuordnen kann. Der ORB wird Methoden des Servant Managers nutzen, um Servants auf Verlangen zu aktivieren oder zu deaktivieren. Der Servant Manager ist auch für die Zuordnung der Objekt ID's zu bestimmten Servants zuständig.

Servant Managers haben im wesentlichen zwei Funktionen:
- finden und aktivieren von Servants
- deaktivieren von Servants.

Um einen Servant Manager einsetzen zu können, muss die USE_SERVANT_MANAGER Policy gesetzt sein. Falls dies der Fall ist, können zwei Arten von Servant managers definiert werden:
1) Servant Activators
   Falls die RETAIN Policy gesetzt wurde, können mit diesem Servant Manager Typ persistente Objekte aktiviert werden.
2) Servant Locators
   Falls die NON_RETAIN Policy gesetzt wurde, kann mit diesem Servant Manager Typ einmalig eingesetzt werden, also typischerweise für transiente Objekte.

Referenz: Kapitel 11.3.4 der CORBA Spezifikation 2.3.1

© J.M.Joller

1.8.4. Naming Service

Die Java Naming und Directory Interface wird in einer anderen Kurseinheit besprochen.
Hier beschränken wir uns auf eine kurze Zusammenfassung von

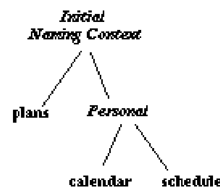
- COS Naiming Services
- Einsatz des Naming Services

1.8.4.1. Übersicht über den COS Naming Service

Der CORBA COS (Common Object Services) Naming Service stellt eine baumartige Verzeichnisstruktur zur Verfügung. JavaIDL benutzt diesen Namens- und Verzeichnisdient gemäss der COS Naming Service Spezifikation.

Unter einer Namensbindung versteht man eine Zuordnung eines Namens zu einem Objekt. Namensbindungen werden immer relativ zu einem Namenskontext definiert. Der Namenskontext ist ein Objekt, welches aus Namensbindungen besteht, wobei jeder Name eindeutig ist. Unterschiedliche Namen können gleichzeitig an ein Objekt gebunden werden. Zudem kann ein Name in unterschiedlichen Namenskontexten erscheinen.

Um einen Namen aufzulösen, muss bestimmt werden, welches Objekt in einem bestimmten Namenskontext zu dem Namen gehört. Ein Name wird also immer relativ zu einem Namenskontext aufgelöst. Es gibt keine absoluten Namen.



Einen Kontext kann man auch wie ein Objekt an einen Namen in einem Namenskontext binden. Damit kann man Namensgraphen kreieren, baumartige Graphen. Mit diesem Konstrukt ist es leicht möglich komplexe Namenskonstrukte aufzubauen, sogenannte Compound Names, oder zusammengesetzte Namen, welche aus Namenssequenzen bestehen.

Damit eine Applikation den COS Namensdienst einsetzen kann, muss dem ORB der Host und der Port des Services bekannt sein. In JavaIDL umfasst der ab J2SE1.4 neue ORB Daemon (`orbd`) einen persistenten und einen transienten COS Namensdienst.

- Der ***Persistent*** *Naming Service* stellt persistente Namenskontexte zur Verfügung. Das heisst also, dass beispielsweise das Binden von Objekten an Namen abgespeichert wird. Immer wenn ORBD neu gestartet wird, werden die Informationen aus dem Datenspeicher gelesen und der Bindungsbaum neu kreiert.
- Aus Rückwärtskompatibilitätsgründen wird immer noch ein ***Transient*** *Naming Service* (`tnameserv`) mitgeliefert. Der Namenskontext bleibt in diesem Fall aber nur solange gültig, wie der Server läuft. Alle Informationen gehen beim Herunterfahren des `tnameserv` verloren.

© J.M.Joller

Damit die Bindung richtig funktioniert, müssen Client und Server sich darüber verständigen, welches Root verwendet werden soll, welcher Baum die Bindungen enthält: der root Context muss beim Client und beim Server der gleiche sein!

Mit der Methode

```
orb.resolve_initial_references(String name_of_service)
```

wird der Root Namenskontext bestimmt.

Falls Sie als Namen des Services `"PNameService"` eingeben, erhalten Sie einen persistenten Namensdienst;
falls Sie aber "`NameService"` eingeben, erhalten Sie einen transienten Namensdienst. Die Bindungsinformationen werden also nach dem Herunterfahren des Namens-Servers verloren gehen.

Beispiel:
```
// Root des Namenskontextes
org.omg.CORBA.Object objRef=orb.resolve_initial_references("PNameService");
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
```

Die Bezeichnung "NameService" gilt für alle ORBS! Die Zeichenkette "PnameService" gilt nur im JavaIDL Umfeld.

Die Methode liefert ein allgemeines CORBA Objekt, welches Sie noch in ein brauchbares `NamingContextExt` Objekt umwandeln müssen, mit der `narrow()` Methode..

Mit dem `ncRef` Objekt vom Typ `org.omg.CosNaming.NamingContextExt` können wir nun auf den Namenskontext zugreifen.

`NamingContextExt` und `NamingContextExtHelper` sind neu in J2SE1.4 `NamingContextExt` ist eine Erweiterung des `NamingContext`. Neu ist die Verwendung des *Interoperable Naming Service* als Basis.

1.8.4.2. Interoperable Naming Service

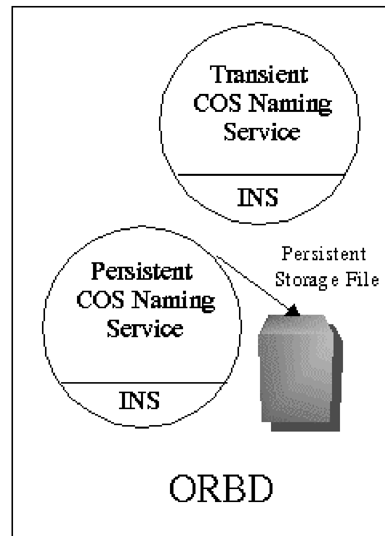
Der *Interoperable Naming Service* (**INS**) ist ein URL basiertes Namenssystem auf CORBA Naming Service Basis. Zudem stellt der Service einen Bootstrap mechanismus zur Verfügung, der es den Applikationen gestattet, Namenskontexte gemeinsam zu nutzen.

Der Interoperable Naming Service (INS) stellt grob folgende Funktionalität zur Verfügung:

- die Möglichkeit zeichenkettenbasierte Namen aufzulösen (Beispiel., a/b.c/d)
- URLs für CORBA Objekt Referenzen (`corbaloc:` und `corbaname:` Formate)
- Standard APIs in `NamingContextExt`, um beispielsweise die Konversion zwischen CosNamen, URLs und Strings zu ermöglichen.
- ORB Argumente zum Bootstrappen (`ORBInitRef` und `ORBDefaultInitRef`)

© J.M.Joller

Das folgende Diagramm zeigt, wie das Zusammenspiel zwischen dem Namensdienst und ORBD schematisch aussieht:



Eine vollständige Objektreferenz enthält mindestens drei bestandteile: eine Adresse, den Namen des POA und die Objekt ID. Mit INS kann man URLs für Objektreferenzen definieren, welche leichter lesbar sind als stringified IORs (Sie sahen Beispiele dieser IORs in den POA Beispielen, immer wenn der Bildschirm halb mit 01001001... gefüllt wurde).

Folgende stringified Objektreferenzen sind im INS erlaubt:

- **Interoperable Object References (IOR)**
  Eine IOR ist eine Objektreferenz, welche von einem ORB verstanden wird und der auf der Basis von IIOP oder GIOP (OMG definiert) kommuniziert. Als Client kann man diese Referenz mit der Methode

  ```
  orb.object_to_string(objRef),
  ```

  erhalten, wie wir gleich in Beispielen nochmals zeigen werden.

- lesbare URL Formate für CORBA Objekt Referenzen

  `corbaloc:`
  Das `corbaloc:` Format ist in CORBA Clients sehr hilfreich und wird beispielsweise für das Auflösen der Referenzen im GIOP Umfeld eingesetzt:

  ```
  corbaloc:iiop:1.2@MeineBank.com:2050/TraderService
  ```

  Dieses Beispiel zeigt, wie eine Objektreferenz auf den Trader Service der Bank MeineBank erhalten werden könnte, an Port 2050.

  `corbaname:`
  Dieses Format wird oft eingesetzt, um einen Client zu bootstrappen:

  ```
  corbaname::meineBank.com:2050#Personal/schedule
  ```

  Der Host ist `meineBank.com`, `2050` ist der Port.

© J.M.Joller

Der Teil vor dem # Zeichen ist die URL, welche den Root Namenskontext liefert.

Dieses Beispiel kombiniert zwei Aktivitäten:
1) es bestimmt den Root Namenskontext
2) es löst den Namen `Personal/schedule` im Namensdienst auf.

1.8.4.2.1. Bootstrap Optionen für den ORB

INS gestattet es dem ORB beim Booten bestimmte Optionen zu setzen. Diese Optionen sind aber plattformabhängig, obschon sie von der OMG standardisiert wurden.

1.8.4.3. Einsatz des Naming Service

Um den Namensdienst einsetzen zu können, muss man Client, Server und Namensdienst Aktivierung programmieren und Objekte in den Namensraum einlagern / hineinschreiben. Bevor Client und Server auf den Namensraum zugreifen können, muss dieser gestartet werden und dem Client / Server mitteilen, wo er zu finden ist. Die folgende Abfolge beschreibt grob den Ablauf beim Zugriff des Servers / Clients auf den Namensdienst:

- der Server ruft eine bind() oder rebind() Methode auf, um einen logischen Namen mit einer Objektreferenz zu verknüpfen.
- der Namensdienst trägt diese Objektreferenz- Bindung in den Namensraum ein
- die Client Applikation ruft die resolve() Methode auf, um eine Objektreferenz mit diesem Namen zu finden.
- der Client benutzt diese Objektreferenz, um Methoden auf dem Zielobjekt (dem Servant) zu nutzen.

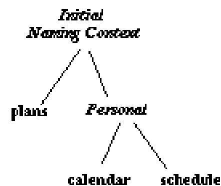
Nun wollen wir dieses Schema genauer untersuchen, anhand von einigen Beispielen:

- ein (Namensdienst-) Client trägt ein Objekt in einen Namensraum ein
- ein Client sucht ein Objekt im Namensraum
- ein Client durchsucht den Namensraum
- der Namensdienst wird gestartet
- der Namensdienst wird gestoppt

© J.M.Joller

1.8.4.3.1. Eintragen eines Objekts in den Namensraum

Das folgende Beispiel illustriert, wie ein Name (eine Namensbindung) in den Namensraum eingetragen wird. Der Namensdienst Client kreiert folgenden Namensbaum. Im Diagramm bedeutet ein *kursive* geschriebener Namen einen *Namenskontext*, die anderen Knoten im Baum stellen Objektreferenzen dar: **plan** ist eine Objektreferenz, ***Personal*** ist ein Namenskontext, welcher die zwei .Objektreferenzen **calendar** und **schedule** enthält.




```
package objekteinnamensraumeintragen;

// 1. importiere die benötigten libraries:
import java.util.Properties;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

/**
 * Title:        Namensraum Einträge
 * Description:  Kreieren eines Namenskontextes
 * Eintragen mehrerer Objekte in diesen Kontext.
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class Client {
   public static void main(String args[]) {
      try {
        // 2. Port und Host setzen und ORB instanzieren
        // 1050 / localhost
        Properties props = new Properties();
        props.put("org.omg.CORBA.ORBInitialPort", "1050");
        props.put("org.omg.CORBA.ORBInitialHost", "localhost");

        ORB orb = ORB.init(args, props);
        // 3. initialer Namenskontext : ctx
        //    und zuordnen an eine dummy Objektreferenz
        NamingContextExt ctx =
          NamingContextExtHelper.narrow(orb.resolve_initial_references(
            "NameService"));
        org.omg.CORBA.Object objref = ctx;

        // 4. Binden des Namens "plans" an die Objektreferenz
        //    "plans" wird an die dummy Objektreferenz gebunden.
        //    rebind statt bind: damit werden allfällig bereits vorhandene
        //    Bindungen einfach erneuert / überschrieben.
        NameComponent name1[] = ctx.to_name("plans");
        ctx.rebind(name1, objref);
        System.out.println("plans rebind erfolgreich!");

        // 5. Kreieren eines neuen Namenskontextes "Personal".
```


© J.M.Joller

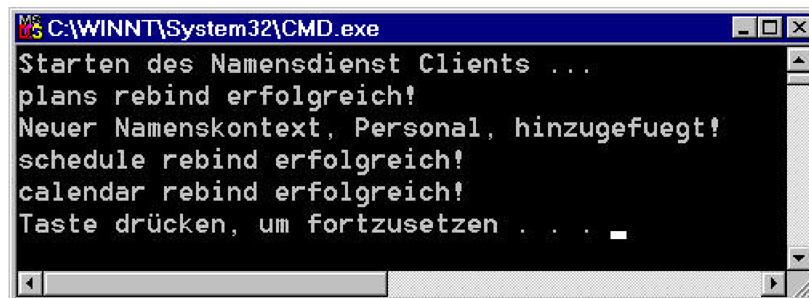
```
        //    Binden weiterer Objektreferenzen
        // neuer Context
        NameComponent name2[] = ctx.to_name("Personal");
        // binden des Contextes ans root (plans)
        // NamingContextExt ctx2 = ctx.bind_new_context(name2);
        NamingContext ctx2 = ctx.bind_new_context(name2);
        System.out.println("Neuer Namenskontext, Personal, hinzugefuegt!");

        // 6. Binde "schedule" und "calendar" an die dummy Objektreferenz.
        //    'Binden' heisst, dass eine neue Namensbindung in einem
        //     gegebenen
        //     Namenskontext eingetragen wird.
        //     Namen werden immer relativ zu einem Kontext aufgelöst -
        //     es gibt also keine absoluten Namen.
        //     Jetzt binden wir die dummy Objektreferenz an "schedule" und
        //     "calendar" im "Personal" Namenskontext (ctx2).
        NameComponent name3[] = ctx.to_name("schedule");
        ctx2.rebind(name3, objref);
        System.out.println("schedule rebind erfolgreich!");

        NameComponent name4[] = ctx.to_name("calendar");
        ctx2.rebind(name4, objref);
        System.out.println("calendar rebind erfolgreich!");

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

Übersetzen des Beispiels geschieht im JBuilder. Zum Starten steht ein Skript zur Verfügung.



Zum Testen müssen Sie zuerst den Namensdienst starten. Dafür steht Ihnen ein Batch Skript zur Verfügung (orbd mit Localhost und Port 1050).
Sie können aber den Client nach dem Starten des Namensdienstes auch im JBuilder starten.

© J.M.Joller

1.8.4.3.2. Objekt im Namensraum auflösen

Das folgende Programmbeispiel zeigt, wie ein Name in einem Namensraum aufgelöst werden kann. Falls Sie mit persistenten Namensdiensten arbeiten, können Sie auf die recovery bei einem Absturz verzichten; im transienten Fall müssen Sie nach einen eventuellen Absturz die Namen erneut auflösen.

Client.java

```
package objekteimnamensraumaufloesen;

//  1. importieren der benötighten Packages und Klassen
import java.util.Properties;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

/**
 * Title:        Objekt Referenzen auflösen
 * Description:  Lesen einer Objektreferenz in einem
 * Namensraum
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class Client {

   public static void main(String args[])
    {
      try {
        // 2. Port und Host setzen
        Properties props = new Properties();
        props.put("org.omg.CORBA.ORBInitialPort", "1050");
        props.put("org.omg.CORBA.ORBInitialHost", "localhost");
        // ORB instanzieren
        ORB orb = ORB.init(args, props);

        // 3. Initialer Namenskontext
        //    ctx zuordnen

        NamingContextExt nc =
          NamingContextExtHelper.narrow(orb.resolve_initial_references(
            "NameService"));

        // 4. Auflösen der einzelnen Namensräume
        org.omg.CORBA.Object sched = nc.resolve_str("Personal/schedule");
        org.omg.CORBA.Object cal = nc.resolve_str("Personal/calendar");
        org.omg.CORBA.Object plan = nc.resolve_str("plans");

      if (sched == null){
        System.out.println("Schedule ist null");
      } else System.out.println("Schedule Objektreferenz wurde gefunden");

      if (cal == null){
        System.out.println("Calendar ist null");
      } else System.out.println("Calendar Objektreferenz wurde gefunden");

      if (plan == null){
        System.out.println("Plans ist null");
```


© J.M.Joller

```
      } else System.out.println("Plans Objektreferenz wurde gefunden");

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

Ablauf:
1) Starten des Namensdienstes, falls er aus dem ersten Beispiel nicht bereits gestartet ist.
2) Client starten (JBuilder oder Skript)

Ausgabe:

```
Schedule Objektreferenz wurde gefunden
Calendar Objektreferenz wurde gefunden
Plans Objektreferenz wurde gefunden
```

1.8.4.3.3. Namensraum durchsuchen

Das folgende Beispiel zeigt, wie man einen Namensraum durchsuchen kann:

```
package namensraumdurchsuchen;

/**
* 1. Importieren der benötigten Packages und Klassen
*/

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

/**
 * Title:        Namensraum durchsuchen
 * Description:  untersuchen eines Namensraumes
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class Client {
   public static void main(String args[])
   {
      try {
        /**
         * 2. Setzen von Host und Port
         */
        Properties props = new Properties();
        props.put("org.omg.CORBA.ORBInitialPort", "1050");
        props.put("org.omg.CORBA.ORBInitialHost", "localhost");
        ORB orb = ORB.init(args, props);

        /**
         * 3. bestimmen des initialen Namenskontextes
         *    'NameService' gilt bei jedem ORB!
         */

        NamingContextExt nc =
          NamingContextExtHelper.narrow(orb.resolve_initial_references(
```


© J.M.Joller

```
            "NameService"));

        /**
         * 4. Auflisten aller Bindungen im Kontext
         *    list() bezieht sich immer auf einen Kontext!
         */

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

        /**
         * 5. Bestimme das Array mit den Bindings
         */

        Binding bindings[] = bl.value;

        /**
         * 6. Bindungs auflisten und ausgeben
         */

        for (int i=0; i < bindings.length; i++) {

            int lastIx = bindings[i].binding_name.length-1;

            // ist die Bindung ein Kontext?
            if (bindings[i].binding_type == BindingType.ncontext) {
              System.out.println( "Context: " +
                bindings[i].binding_name[lastIx].id);

            // nein: dann ist es ein Objekt
            } else {
                System.out.println("Objekt: " +
                  bindings[i].binding_name[lastIx].id);
            }
        }

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

Ablauf:

1) Starten des Namensdienstes, falls er aus dem ersten Beispiel nicht bereits gestartet ist.
2) Client starten (JBuilder oder Skript)

Ausgabe:

```
Context: Personal
Objekt: plans
```


© J.M.Joller

1.8.4.3.4. Starten des Namensdienstes

In den obigen Beispielen haben wir immer den ORB Daemon mit einem Port und einem Host gestartet. Weitere Optionen

Der Daemon besitzt einige weitere Möglichkeiten:

MUSS Option

-ORBInitialPort ***nameserverport***

Angabe des Standard-Ports (siehe Beispiele

ANDERE Optionen

-port ***port***

ein Aktivierungsport, in der Regel 1049, an dem ORBD Anfragen für persistente Objekte (deren Aktivierung) erhält. Sie sehen dies auch im Debug Modus!

-defaultdb ***directory***

spezifiziert das Verzeichnis, in dem die ORBD Datenbank kreiert wird.
Standardwert ist "./orb.db".

-serverPollingTime ***milliseconds***

gibt an, wieoft der Daemon prüft, ob ein mittels Server-Tool registrierter Server noch am Leben ist.
Standardwert ist 1,000 ms.
Gültige Werte sind: positive ganze Zahl.

-serverStartupDelay ***milliseconds***

eine Zeitspanne, die der ORBD wartet, bevor er eine Exception wirft, weil ein persistenter Server nicht wieder gestartet werden kann.
Standardwert ist 1,000 ms. Die Angabe ist in *milliseconds* und muss eine positive ganze Zahl sein.

-J*option*

Java Virtual Machine Parameter.
Beispiel: **-J-Xms48m** setzt das Start Memory auf 48 Megabytes.

1.8.4.3.5. Stoppen des Namensservices

Der Namensservice muss unter Windows mit CTRL/C, unter Solaris / Unix mit kill gestoppt werden.

© J.M.Joller

1.8.5. General InterORB Protocol (GIOP)

Das GIOP ist ein sogenanntes Wire Protokoll, also ein Protokoll,welches der ORB für die interne Kommunikation zwischen Client und Server einsetzt.

Dieses Protokoll wurde von der OMG standardisiert. IIOP, das Internet Inter-ORB protokoll ist ein Spezialfall von GIOP, welches speziell auf die TCP / IP basierte ORB Kommunikation ausgerichtet ist.

J2SE v.1.3.1 und höher unterstützen das GIOP 1.2 Protokoll

Enterprise JavaBeans (EJB) 2.0 verwendet IIOP für die Interoperabilität von J2EE Servern. IIOP verwendet ein request/response Pattern. Dies gestattet eine sichere Verknüpfung von Transaktionen und Kontexten über mehrere J2EE Server.

Für den Anwendungsprogrammierer ist der Aufbau von GIOP nicht wesentlich, für den Implementierer eines ORBs schon.

1.8.6. Offizielle Spezifikationen für CORBA und Support in J2SE 1.4

Die formale Spezifikation von CORBA umfasst mehrere hundert Seiten. Es gibt kaum kommerzielle Produkte, welche alles implementieren. JDK1.4 berücksichtigt im Wesentlichen die

- idl to Java Mapping Spezifikation
- den Interoperable Naming Service und
- den Portable Interceptor (im Beta noch nicht voll implementiert)

1.8.6.1. Spezifikationen im Detail

Die folgenden Teile der CORBA Spezifikation sind in JDK implementiert:

- CORBA 2.3.1 chapters 1-3 and 6-9
- Revised IDL to Java language mapping, section 1.21.8.2, The `orb.properties` File
- CORBA 2.3.1 chapter 4 mit folgenden Änderungen aus der Portable Interceptors Specification:
    - section 4.2.3.5 `destroy`
    - section 4.5.2 `CodeFactory` und `PICurrent`
    - Section 4.5.3.4 `register_initial_reference`
- CORBA 2.3.1 chapter 5 mit Updates aus der Portable Interceptors Specification:
    - 5.5.2 `StringSeq` und `WStringSeq` in `org.omg.CORBA`:
        - `StringSeqHolder`
        - `StringSeqHelper`
        - `WStringSeqHolder`
        - `WStringSeqHelper`
- CORBA 2.3.1 sections 10.6.1 und 10.6.2
- CORBA 2.3.1 section 10.7 für `TypeCode` APIs.
- CORBA 2.3.1 chapter 11 mit Updates aus der Portable Interceptors Specification:
    - Section 11.3.7 POAs must preserve all policies
    - Section 11.3.8.2 POAs must preserve all policies

© J.M.Joller

    - Section 11.3.8.26 POA::id is required.
- CORBA 2.3.1 chapters 13 und 15 definieren GIOP 1.0, 1.1, und 1.2.
  J2SE 1.4. ORB unterstützt alle Versionen von GIOP, ausser das bidirectional GIOP aus 15.8 und 15.9.
- Interoperable Naming Service wird unterstützt
- Portable Interceptors 13.8

1.8.6.2. Tools

- IDL to Java compiler (`idlj`) gemäss
    - CORBA 2.3.1 chapter 3 (IDL definition)
    - CORBA 2.3.1 chapters 5 und 6 (semantics of Value types)
    - CORBA 2.3.1 section 10.6.5 (pragmas)
    - The IDL to Java mapping specification
- Java to IDL compiler (IIOP backend für rmic)
    - CORBA 2.3.1 chapters 5 und 6 (value types)
    - The Java to IDL language mapping.
    - IDL mit dem -idl Flag entspricht CORBA 2.3.1 chapter 3.

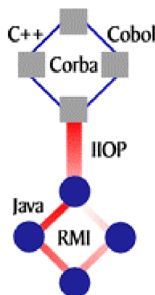
© J.M.Joller

1.9. Ergänzung - RMI over IIOP mit POA

Die Remote Methode Invocation über Internet Inter-ORB Protokoll (RMI-IIOP) ist Teil der J2SE Plattform. Mit RMI kann man CORBA ähnliche Applikationen 100% in Java entwickeln. RMI setzt standardmässig JRMP, das Java Remote Method Protocol für die Kommunikation ein. CORBA taugliche Anwendungen setzen aber das IIOP Protokoll ein. RMI-IIOP setzt IIOP und den Java CORBA Object Request Broket (ORB) ein. Sie können damit also in Java programmieren, `rmic` einsetzen (ohne IDL) und CORBA konforme Applikationen entwickeln. Damit Sie die IDL Spezifikation gleich auch noch zur Verfügung haben, können Sie mit dem rmic auch IDL generieren: setzen Sie einfach das `-idl` Flag beim `rmic`. Falls Sie IIOP einsetzen wollen, müssen Sie das `-iiop` Flag verwenden. In der Kurseinheit RMI Praxis finden Sie ein RMI over IIOP Beispiel, mit Generierung der IDL Beschreibung.

1.9.1. Wann macht RMI oder IIOP Sinn?

RMI-IIOP ist für Java Programmierer gedacht, die das RMI Interface in Java spezifizieren wollen, aber als Transportprotokoll IIOP einsetzen wollen, nicht JRMP (Java Remote Method Protocol).



Mit IIOP kann eine bestimmte Interoperabilität mit CORBA erreicht werden. IIOP wird auch bei Enterprise Java Beans eingesetzt, da die EJBs RMI basiert sind (das remote Modell von EJB).

RMI IIOP versucht, das Beste von RMI (Einfachheit im Vergleich mit CORBA) mit dem Besten von CORBA (Standard, Heterogenität) zu verknüpfen.

Mit RMI IIOP braucht man kein neues IDL zu lernen. Auch Clients (oder Server / Servants) können in irgend einer anderen Sprache geschrieben werden. Das IDL für die entsprechende Interface Beschreibung kann sogar aus der Java Spezifikation generiert werden.

1.9.2. Welche anderen Möglichkeiten zur Programmierung Verteilter Systeme habe ich noch?

Generell unterscheidet man die zwei grundsätzlichen Techniken:
- synchrone Kommunikation (Sockets, RMI, CORBA, JINI)
- asynchrone Kommunikation (JMS, MQSeries: Message basierte Middelware)

Innerhalb der synchronen Welt (auf die andere kommen wir in der Kurseinheit über JMS, Java Messaging Services, zurück), kann man grob folgende Aussagen machen:
- JAVA RMI :
  falls die gesamte Applikation in java geschrieben ist, wird RMI der effizienteste Mechanismus sein, die Kommunikation aufzubauen und zu unterhalten. RMI bietet Security, Garbage Collection, Objekt Aktivierung, ... Alternative : Messaging.
- Java IDL / CORBA:
  falls Sie in einem (sprachlich) heterogenen Umfeld Applikationen realisieren müssen, ist CORBA (oder Messaging) die beste Lösung

© J.M.Joller

- Enterprise Java Beans:
  falls Sie einen Applikationsserver (Weblogic, WebSphere, ORACLE) zur Verfügung haben, sind EJBs eine gute Lösung (Sie können auch Message Beans einsetzen).

1.9.3. Was ist neu in J2SE 1.4 RMI IIOP

JDK J2SE1.4 bringt einige Neuerungen im IDL / CORBA Bereich, wie wir gesehen haben (POA, Interceptors, ...). Diese Features kann man auch mit RMI IIOP nutzen.

- Mit der neusten Version des `rmic` Tools können Sie auch den *Portable Object Adapter* (POA) einsetzen.
  rmic -iiop -poa generiert Java Code, welcher nicht mehr `org.omg.CORBA_2_3.portable.ObjectImpl`, sondern `org.omg.PortableServer.Servant`. basierten Code ist.

- Die neuste Version des ORB Daemons `orbd`, enthält einen Transient Name Service, einen Persistent Name Service und einen Servant Manager. Damit erweitern sich die Möglichkeiten von RMI IIOP entsprechend.

1.9.4. RMI IIOP Beispiel

Um den Einsatz von IIOP und POA in RMI besser verstehen zu können, schauen wir das Ganze an einem Beispiel an.

Das Beispiel wird in folgenden Schritten entwickelt:

1. Definition der remote Klassen als Interfaces (in Java)
2. Implementieren der Interfaces
3. Schreiben des Servers
4. Schreiben des Clients

1.9.4.1. Definition der remote Klassen als Interfaces in Java

In Java ist ein remote Objekt eine Klasse, welche ein Remote Interface implementiert. Das remote In terface definiert alle Methoden, welche von anderen Maschinen genutzt werden könnten. Remote Interfaces besitzen folgende Charaktersitiken:

- das remote Interface muss als `public` deklariert werden, sonst wird eine Ausnahme geworfen, sobald der Client versucht die Methode einzusetzen, ausser der Client und das remote Interface sind im selben Package.
- das remote Interface erweitert `java.rmi.Remote` (Interface).
- jede Methode muss die Ausnahme `java.rmi.RemoteException` (oder eine Oberklasse davon) deklarieren.
- der Datentyp jedes remote Objekts, welches als Argument oder Rückgabewert eingesetzt wird (direkt oder in ein anderes Objekt eingebettet) muss mit dem *remote Interface Datentyp* (Beispiel `HalloInterface`), nicht der Implementationsklasse (`HalloImpl`) deklariert werden.

Nach der langen Einleitung schreiben wir nun endlich das Interface und die Methoden Deklarationen:

© J.M.Joller

```
//Interface.java
package rmioveriiopeinfuehrendesbeispiel;

import java.rmi.Remote;

/**
 * Title:        RMI over IIOP
 * Description:einfaches Beispiel für den Einsatz von RMI IIOP im JDK1.4
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public interface Interface extends Remote {
/**
 * Methode wieGehts(String frage) :
 * Eingabeparameter : frage (ein beliebiger Text als Zeichenkette
 * Rückgabe: eine Zeichenkette (Text), welcher die Antwort auf die Frage
 *           enthält
 */
   public String wieGehts(String frage) throws java.rmi.RemoteException;
}
```

Die remote Exception muss immer dabei sein, weil es beliebig viele Gründe gibt, dass die remote Methode nicht korrekt ausgeführt werden kann.

1.9.4.2. Implementieren der Interfaces

Die Implementation eines remote Objekts muss mindestens folgendes umfassen:

- sie muss mindestens eine remote *Schnittstelle* implementieren
- sie muss den *Konstruktor* für remote Objekte definieren
- sie muss Implementationen für die remote *Methoden* liefern.

```
//Servant.java
package rmioveriiopeinfuehrendesbeispiel;

/**
 * Hier wird der POA bereits vorbereitet! Bedingt JDK1.4+
 */
import javax.rmi.PortableRemoteObject;

public class Servant extends PortableRemoteObject implements Interface {

  public Servant() throws java.rmi.RemoteException {
  /**
   * mit super() wird das RMI Linking und
   * die RMI Objektserialisierung ermöglicht
   */
  }
  public String wieGehts(String frage) throws java.rmi.RemoteException {
    System.out.println("[Servant]wieGehts()");
    return "Mir geht's gut!\nDanke für Deine Frage : "+frage;
  }
}
```


© J.M.Joller

1.9.4.2.1. Mindestens eine remote *Schnittstelle* implementieren

In Java, wenn ein Klasse deklariert, dass sie ein Interface implementiert, wird eine Art Kontrakt zwischen der Klasse und dem Compiler abgeschlossen.

Die Klasse verspricht mit diesem Kontrakt, dass sie die Methoden im Interface zur Verfügung stellt, mit der korrekten Signatur. Interface Methoden sind implizit `public`. und `abstract`. Falls also die Implementationsklasse eine der Methoden nicht implementiert, wird die Implementationsklasse automatisch abstrakt.

```
public class Servant extends PortableRemoteObject implements Interface {
```

In diesem Beispiel erweitern wir die Klasse `javax.rmi.PortableRemoteObject`.

Damit steht uns automatisch der IIOP basierte Transport zur Verfügung.

1.9.4.2.2. *Konstruktor* für remote Objekte definieren

Der Konstruktor für eine remote Klasse funktioniert genau so, wie der Konstruktor einer nicht remote Klasse: die Variablen jeder neu kreierten Instanz der Klasse werden initialisiert und es wird eine Instanz der Klasse zurückgegeben.

Zusätzlich muss beim remote Objekt das Objekt 'exportiert' werden. Exportieren eines (remote) Objekts stellt es ankommenden remote Methodenaufrufen zur Verfügung.

In unserem Fall geschieht der Export implizit, weil dieser Schritt durch das `PortableRemoteObject` erledigt wird, automatisch beim Kreieren des Objekts.Wegen diesem Export muss allerdings der Konstruktor bereits mit der remote Exception ausgestattet werden.

```
public Servant() throws java.rmi.RemoteException {
  /**
   * mit super() wird das RMI Linking und
   * die RMI Objektserialisierung ermöglicht
   */
    super();
  }
```

Dabei ist zu beachten:

- Der Aufruf von `super()` führt den Konstruktor von `javax.rmi.PortableRemoteObject` aus. Dieses Objekt exportiert (macht bekannt) das remote Objekt.
- Der Konstruktor muss `java.rmi.RemoteException` werfen, weil RMI beim Versuch das Objekt zu exportieren unterbrochen werden könnte.

Eigentlich geschieht der Aufruf des Konstruktors der Oberklasse (`super()`) automatisch. Aber es ist besser, wenn Sie diesen Schritt sichtbar machen!

© J.M.Joller

1.9.4.2.3. Implementationen für die remote *Methoden* liefern.

Nun müssen wir also die einzelnen Methoden (alle!) des oder der remote Interfaces implementieren.

```
public String wieGehts(String frage) throws java.rmi.RemoteException {
    System.out.println("[Servant]wieGehts()");
    return "Mir geht's gut!\nDanke für Deine Frage : "+frage;
  }
```

Als Argument bei den Methoden können Sie alle Datentyen von Java einsetzen, also auch Objekte, sofern das Objekt das Interface `java.io.Serializable` implementiert. Dies ist für die meisten Basisklassen in Java der Fall.

In RMI geschieht nun folgendes:

- standardmässig werden Objekte **'by copy'** übergeben. Das heisst, dass alle Datenfelder, ausser den statischen und transienten, kopiert werden (siehe *Java Object Serialization Specification* für die Details, wie dies geschieht).
- **Remote Objecte** werden **'by reference'** übergeben. Eine Referenz auf ein remote Objekt besteht eigentlich aus einer Referenz auf einen Stub, den clientseitigen Proxy des remote Objekts. (siehe *Java Remote Method Invocation Specification*).

1.9.4.3. Schreiben des Servers

Eine Server Klasse besteht aus einer main() Methode, in der eine Instanz der remote Implementation erstellt wird (des Servants). Diese wird dann in einem Namensdienst eingetragen.

```
package rmioveriiopeinfuehrendesbeispiel;

import javax.naming.InitialContext;
import javax.naming.Context;

/**
 * Title:        RMI over IIOP
 * Description:  einfaches Beispiel für den Einsatz von RMI over IIOP im
JDK1.4
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class Server {
    public static void main(String[] args) {
        try {
            System.out.print("[Server]Start");
            // 1: Instanzieren des Servants
            System.out.print("[Server]Instanzieren des Servants");
            Servant remoteRef = new Servant();

            // 2) Eintragen in den Namensdienst
            // (JNDI API)
            Context initialNamingContext = new InitialContext();
            System.out.print("[Server]Eintrag in den Namensraum");
            initialNamingContext.rebind("RemoteService", remoteRef );
```


© J.M.Joller

```
                System.out.print("[Server]Der Server ist bereit ");
                System.out.print("und wartet auf Anfragen...\n");

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

1.9.4.3.1. Instanzieren des Servants

In der main() Methode des Servers wird als erstes eine Instanz des Servants, der remote Objektimplementation gebildet. Mit dem Konstruktor wird das Objekt auch gleich exportiert, also bekannt gemacht.

1.9.4.3.2. Eintragen in den Namensdienst

Damit ein Client auf die remote Objekte zugreifen kann, muss der Aufrufer (der Client) irgendwie eine Referenz auf das remote Objekt erhalten.

Falls wir das Objekt in einen Namensdienst eintragen, wird dieser Schritt einfach: der Client kann im Namensraum nach der Referenz suchen, eine Referenz verlangen und dann dessen Methoden aufrufen.

In unserem Beispiel verwenden wir den neuen ORB daemon als Namensserver.

```
// 2) Eintragen in den Namensdienst
           // (JNDI API)
           Context initialNamingContext = new InitialContext();
           System.out.print("[Server]Eintrag in den Namensraum");
           initialNamingContext.rebind("RemoteService", remoteRef );
```

Die Methode `rebind()` besitzt zwei Argumente: den Namen, unter dem die Objektreferenz gespeichert wird und eine Objektreferenz (die zum remote Objekt, welche wir im ersten Schritt erstellt haben).

© J.M.Joller

1.9.4.4. Schreiben des Clients

Jetzt bleibt uns nur noch die Erstellung des Clients. Dieser wird dann die Methode(n) des Servants aufrufen.

```
package rmioveriiopeinfuehrendesbeispiel;

import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;
import javax.rmi.*;
import java.util.Vector;
import javax.naming.NamingException;
import javax.naming.InitialContext;
import javax.naming.Context;

/**
 * Title:        RMI over IIOP
 * Description:  einfaches Beispiel für den Einsatz von RMI over IIOP im
JDK1.4
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class Client {
    public static void  main( String args[] ) {
        Context ic;
        Object objRef;
        Interface iFace;
        System.out.println("[Client]Start");
        try {
            ic = new InitialContext();

        // 1: Bestimmen der Objektreferenz im Namensraum
        // (JNDI call).
            System.out.print("[Client]Bestimmen der Objektreferenz");
            System.out.print("        im Namensraum / Naming Service\n");
            objRef = ic.lookup("RemoteService");

        // 2: Umwandeln der Objektreferenz (narrow())
        // und Methodenaufruf.
            System.out.print("[Client]Umwandeln der Objektreferenz");
            System.out.print("        in ein Interface Objekt\n");
            iFace = (Interface)PortableRemoteObject.narrow(
                objRef, Interface.class);
            System.out.print("[Client]Aufruf der remote Methode");
            iFace.wieGehts( "Wie geht's Dir? " );

        } catch( Exception e ) {
            System.err.println( "[Client]Exception " + e + "Caught" );
            e.printStackTrace( );
            return;
        }
    }
}
```


© J.M.Joller

Die Client Applikation bestimmt als erstes eine Referenz auf das Remote Objekt (welches unter dem Namen 'RemoteService' vom Naming Service angeboten wird). Als Naming Service verwenden wir das JNDI (Java Naming and Directory Interface, welches wir als `orbd` nutzen):

```
Naming.lookup()
```

Die Methode liefert eine remote Implementation des Interfaces.
Der anschliessende Aufruf der remote Methode liefert die von uns so gewünschte Ausgabe einer Zeichenkette.

Jetzt müssen wir uns noch mit der Mechanik befassen:

1.9.4.5. Übersetzen des Beispiels

Da das gesamte Beispiel mit dem JBuilder erstellt wurde, geschieht dieser Schritt im JBuilder. Ich habe beim Erstellen jeweils alle Informationen ausgenutzt, also den Servant als Implementation des Interfaces deklariert. In diesem Fall muss man nach der Erstellung des Interfaces dieses sofort übersetzen, sonst findet der JBuilder die Interface Klasse nicht und gibt eine Fehlermeldung aus.

Der einzige nicht banale Schritt ist die Generierung von Stub und Skeleton:

```
rmic -v1.2 -iiop Servant
```

dazu ändern Sie beim Servant die Properties so, dass Stub und Skeleton V2 konform mit der zusätzlichen Option -iiop kreiert werden. Die Generierung von Stub und Skeleton dauert einige Zeit!

1.9.4.6. Testen der Applikation

Das Testen geschieht in folgenden drei Schritten

1. Starten des Namens Dienstes (ORBD)
2. Starten des Servers
3. Starten der Client Applikation

1.9.4.6.1. Starten des Namensdienstes

Wir verwenden die bereits in den POA Beispielen verwendeten Namensdienst, mit der selben Batch Datei. Der ORB Daemon ist der RMI Registry bei weitem überlegen: wir können den Dienst irgendwo starten, nicht wie bei der Registry im Serververzeichnis.

Etwas müssen Sie unbedingt beachten:
immer wenn Sie beim Interface oder der Implementation irgend etwas ändern, müssen Sie den Namensdienst neu starten bzw. das Objekt neu einbinden, sonst haben Sie ein altes Referenzobjekt!

© J.M.Joller

1.9.4.6.2. Starten des Servers

Dieser Schritt ist banal, sofern Sie alle Flags kennen und korrekt angeben. Der Einfachheit halber steht Ihnen ein Skript zur Verfügung:

```
%JAVA_HOME%\bin\java -classpath . -
Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory      -
Djava.naming.provider.url=iiop://localhost:1050
rmioveriiopeinfuehrendesbeispiel.Server
```

Dies liefert folgende Ausgabe:

```
Starten des Servers...
[Server]Start
[Server]Instanzieren des Servants
[Server]Eintrag in den Namensraum
[Server]Der Server ist bereit und wartet auf Anfragen...
```

1.9.4.6.3. Starten des Clients

Auch beim Client benötigen wir einige Flags:

```
%JAVA_HOME%\bin\java -classpath . -
Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory -
Djava.naming.provider.url=iiop://localhost:1050
rmioveriiopeinfuehrendesbeispiel.Client
```

Dies liefert folgende Ausgabe:

```
Starten des RMI Clients mit IIOP...
[Client]Start
[Client]Bestimmen der Objektreferenz        im Namensraum / Naming Service
[Client]Umwandeln der Objektreferenz        in ein Interface Objekt
[Client]Aufruf der remote Methode : Wie geht's Dir?
Mir geht's gut!
Danke fuer Deine Frage : Wie geht's Dir?
Taste drücken, um fortzusetzen . . .
```

wobei auch serverseitig eine Ausgabe, durch den Servant, erfolgt:

```
[Servant]wieGehts()
```


© J.M.Joller

1.9.5. RMI-IIOP mit POA-basiertem Servermodell

Nachdem wir eine RMI -IIOP Applikation zum Laufen gebracht haben, wollen wir nun auch noch den POA einsetzen, in RMI!

1. schreiben der Applikation
2. übersetzen der Java Dateien
3. starten des Namensdienstes, des Servers und des Clients

1.9.5.1. Schreiben der Applikation

Dies geschieht wie in den vorangegangenen Beispielen in folgenden Schritten:
1. Definition des remote Interfaces in Java
2. schreiben einer Implementationsklasse (Servant)
3. schreiben des Servers
4. schreiben des Client Programms welcher remote Services benutzt.

1.9.5.1.1. Definition des remote Interfaces in Java

In Java ist ein remote Objekt eine Klasse, welche ein Remote Interface implementiert. Das remote In terface definiert alle Methoden, welche von anderen Maschinen genutzt werden könnten. Remote Interfaces besitzen folgende Charaktersitiken:

- das remote Interface muss als `public` deklariert werden, sonst wird eine Ausnahme geworfen, sobald der Client versucht die Methode einzusetzen, ausser der Client und das remote Interface sind im selben Package.
- das remote Interface erweitert `java.rmi.Remote` (Interface).
- jede Methode muss die Ausnahme `java.rmi.RemoteException` (oder eine Oberklasse davon) deklarieren.
- der Datentyp jedes remote Objekts, welches als Argument oder Rückgabewert eingesetzt wird (direkt oder in ein anderes Objekt eingebettet) muss mit dem *remote Interface Datentyp* (Beispiel `HalloInterface`), nicht der Implementationsklasse (`HalloImpl`) deklariert werden.

Nach der langen Einleitung schreiben wir nun endlich das Interface und die Methoden Deklarationen:

```
//Interface.java
package rmioveriiopeinfuehrendesbeispiel;

import java.rmi.Remote;

/**
 * Title:        RMI over IIOP
 * Description:einfaches Beispiel für den Einsatz von RMI IIOP im JDK1.4
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public interface Interface extends Remote {
```


© J.M.Joller

```
/**
 * Methode wieGehts(String frage) :
 * Eingabeparameter : frage (ein beliebiger Text als Zeichenkette
 * Rückgabe: eine Zeichenkette (Text), welcher die Antwort auf die Frage
 *           enthält
 */
   public String wieGehts(String frage) throws java.rmi.RemoteException;
}
```

Die remote Exception muss immer dabei sein, weil es beliebig viele Gründe gibt, dass die remote Methode nicht korrekt ausgeführt werden kann.

1.9.5.1.2. Implementieren der Interfaces

Die Implementation eines remote Objekts muss mindestens folgendes umfassen:

- sie muss mindestens eine remote *Schnittstelle* implementieren
- sie muss den *Konstruktor* für remote Objekte definieren
- sie muss Implementationen für die remote *Methoden* liefern.

```
//Servant.java
package rmioveriiopmitpoa;

/**
 * Hier wird der POA bereits vorbereitet! Bedingt JDK1.4+
 */
import javax.rmi.PortableRemoteObject;

public class Servant extends PortableRemoteObject implements Interface {

  public Servant() throws java.rmi.RemoteException {
  /**
   * mit super() wird das RMI Linking und
   * die RMI Objektserialisierung ermöglicht
   */
  }
  public String wieGehts(String frage) throws java.rmi.RemoteException {
    System.out.println("[Servant]wieGehts()");
    return "Mir geht's gut!\nDanke für Deine Frage : "+frage;
  }
}
```


© J.M.Joller

1.9.5.1.2.1. Mindestens eine remote Schnittstelle *implementieren*

In Java, wenn ein Klasse deklariert, dass sie ein Interface implementiert, wird eine Art Kontrakt zwischen der Klasse und dem Compiler abgeschlossen.

Die Klasse verspricht mit diesem Kontrakt, dass sie die Methoden im Interface zur Verfügung stellt, mit der korrekten Signatur. Interface Methoden sind implizit `public`. und `abstract`. Falls also die Implementationsklasse eine der Methoden nicht implementiert, wird die Implementationsklasse automatisch abstrakt.

```
public class Servant extends PortableRemoteObject implements Interface {
```

In diesem Beispiel erweitern wir die Klasse `javax.rmi.PortableRemoteObject`.

Damit steht uns automatisch der IIOP basierte Transport zur Verfügung.

1.9.5.1.2.2. Konstruktor *für remote Objekte definieren*

Der Konstruktor für eine remote Klasse funktioniert genau so, wie der Konstruktor einer nicht remote Klasse: die Variablen jeder neu kreierten Instanz der Klasse werden initialisiert und es wird eine Instanz der Klasse zurückgegeben.

Zusätzlich muss beim remote Objekt das Objekt 'exportiert' werden. Exportieren eines (remote) Objekts stellt es ankommenden remote Methodenaufrufen zur Verfügung.

In unserem Fall geschieht der Export implizit, weil dieser Schritt durch das `PortableRemoteObject` erledigt wird, automatisch beim Kreieren des Objekts.Wegen diesem Export muss allerdings der Konstruktor bereits mit der remote Exception ausgestattet werden.

```
public Servant() throws java.rmi.RemoteException {
  /**
   * mit super() wird das RMI Linking und
   * die RMI Objektserialisierung ermöglicht
   */
    super();
  }
```

Dabei ist zu beachten:

- Der Aufruf von `super()` führt den Konstruktor von `javax.rmi.PortableRemoteObject` aus. Dieses Objekt exportiert (macht bekannt) das remote Objekt.
- Der Konstruktor muss `java.rmi.RemoteException` werfen, weil RMI beim Versuch das Objekt zu exportieren unterbrochen werden könnte.

Eigentlich geschieht der Aufruf des Konstruktors der Oberklasse (`super()`) automatisch. Aber es ist besser, wenn Sie diesen Schritt sichtbar machen!

© J.M.Joller

1.9.5.1.2.3. Implementationen für die remote Methoden *liefern.*

Nun müssen wir also die einzelnen Methoden (alle!) des oder der remote Interfaces implementieren.

```
public String wieGehts(String frage) throws java.rmi.RemoteException {
    System.out.println("[Servant]wieGehts()");
    return "Mir geht's gut!\nDanke für Deine Frage : "+frage;
  }
```

Als Argument bei den Methoden können Sie alle Datentyen von Java einsetzen, also auch Objekte, sofern das Objekt das Interface `java.io.Serializable` implementiert. Dies ist für die meisten Basisklassen in Java der Fall.

In RMI geschieht nun folgendes:

- standardmässig werden Objekte **'by copy'** übergeben. Das heisst, dass alle Datenfelder, ausser den statischen und transienten, kopiert werden (siehe *Java Object Serialization Specification* für die Details, wie dies geschieht).
- **Remote Objecte** werden **'by reference'** übergeben. Eine Referenz auf ein remote Objekt besteht eigentlich aus einer Referenz auf einen Stub, den clientseitigen Proxy des remote Objekts. (siehe *Java Remote Method Invocation Specification*).

1.9.5.1.3. Schreiben der Server Klasse

Eine Serverklasse ist eine Klasse, mit einer `main()` Methode, welche eine Instanz der remote Objektimplementation (Servant) kreiert und diese an einen Namen in einem Namensraum (durch einen Namensdienst) bindet. Die Servant Klasse kann auch gleich im Server implementiert werden, da sie ja beim Server zur Verfügung stehen muss. In unserem Beispiel trennen wir aber die beiden Klassen.

Und hier die einzelnen Schritte im Server:

1. kreieren eines Portable Objekt Adapters (POA) mit der entsprechenden Policy.
2. aktivieren des POA Managers.
3. kreieren einer Instanz des Servants und aktivieren der Tie,
   kreieren einer Instanz des remote Objekts.
4. publizieren der Objektreferenz
5. warten auf ankommende Anfragen

```
package rmooveriiopmitpoa;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.rmi.PortableRemoteObject ;
import com.sun.corba.se.internal.POA.POAORB;
import org.omg.PortableServer.*;
import java.util.*;
import org.omg.CORBA.*;
import javax.rmi.CORBA.Stub;
import javax.rmi.CORBA.Util;

/**
 * Title:        RMI over IIOP
 * Description:  RMI over IIOP mit POA
```


© J.M.Joller

```
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class Server {
  public Server(String[] args) {
    try {
      System.out.println("[Server]Start");
      System.out.println("[Server]Setzen der Laufzeit Eigenschaften");
      Properties p = System.getProperties();
      // Laufzeit Eigenschaften / Properties
      p.put("org.omg.CORBA.ORBClass",
          "com.sun.corba.se.internal.POA.POAORB");
      p.put("org.omg.CORBA.ORBSingletonClass",
          "com.sun.corba.se.internal.corba.ORBSingleton");

      ORB orb = ORB.init( args, p );

      System.out.println("[Server]POA");
      // POA Definition und Instanzierung
      System.out.println("[Server]POA - setzen des Roots");
      // 1) rootPOA bestimmen
      POA rootPOA = (POA)orb.resolve_initial_references("RootPOA");

      // 2) kreiere den POA mit den entsprechenden Policies
      System.out.println("[Server]POA - Definition der Policies");
      Policy[] tpolicy = new Policy[3];
      tpolicy[0] = rootPOA.create_lifespan_policy(
        LifespanPolicyValue.TRANSIENT );
      tpolicy[1] = rootPOA.create_request_processing_policy(
        RequestProcessingPolicyValue.USE_ACTIVE_OBJECT_MAP_ONLY );
      tpolicy[2] = rootPOA.create_servant_retention_policy(
        ServantRetentionPolicyValue.RETAIN);

      System.out.println("[Server]POA - Definition von
'MeinTransienterPOA'");
      POA tPOA = rootPOA.create_POA("MeinTransienterPOA", null, tpolicy);


      // 3) Aktivieren des POAs
      // der POAManager ist dann im HOLD Zustand
      System.out.println("[Server]POA - Aktivierung");
      tPOA.the_POAManager().activate();

      // 4) Instanzieren des Servants und aktivieren der Tie
      // POA Policy : USE_ACTIVE_OBJECT_MAP_ONLY
      System.out.println("[Server]Servant - Instanzierung");
      Servant servantRef = new Servant();
      System.out.println("[Server]Servant - Tie Instanzierung");
      _Servant_Tie tie = (_Servant_Tie)Util.getTie( servantRef );
      String halloId = "RMIoverIIOPmitPOA";
      byte[] id = halloId.getBytes();
      System.out.println("[Server]Servant - Aktivierung");
      tPOA.activate_object_with_id( id, tie );

      // 5) Publizieren der Objektreferenz mit der selben Objekt ID wie
      //    das Tie Objekt
      System.out.println("[Server]NamingContext - Instanzierung");
      Context initialNamingContext = new InitialContext();
      System.out.println("[Server]NamingContext - POAServer eintragen");
```


© J.M.Joller

```
      initialNamingContext.rebind("POAServer",
        tPOA.create_reference_with_id(id,
          tie._all_interfaces(tPOA,id)[0]) );

      // 6) waren auf Clients
      System.out.println("[Server]warten auf Clients");
      orb.run();
    }
      catch (Exception e) {
        System.out.println("[Server]Probleme im Server: " + e);
        e.printStackTrace();
      }
  }


  public static void main(String args[]) {
    new Server( args );
  }
}
```

1.9.5.1.3.1. Kreieren des POA mit passender Policy

In unserem Beispiel haben wir folgende Policy gwählt:

```
      Policy[] tpolicy = new Policy[3];
      tpolicy[0] = rootPOA.create_lifespan_policy(
        LifespanPolicyValue.TRANSIENT );
      tpolicy[1] = rootPOA.create_request_processing_policy(
        RequestProcessingPolicyValue.USE_ACTIVE_OBJECT_MAP_ONLY );
      tpolicy[2] = rootPOA.create_servant_retention_policy(
        ServantRetentionPolicyValue.RETAIN);
      POA tPOA = rootPOA.create_POA("MeinTransienterPOA", null, tpolicy);
```

1.9.5.1.3.2. Aktivieren des POA Managers

Jeder POA besitzt einen ihm zugeordneten POA Manager. Ein POA Manager kann auch für mehrere POA Objekte zuständig sein. Der POA Manager kapselt den Zustand des / der POAs. Beim Kreieren wird der POA Manager in den Zustand HOLD versetzt, er muss also aktiviert werden:

```
tPOA.the_POAManager().activate();
```

1.9.5.1.3.3. Kreieren eines remote Objekts

Die Hauptaufgabe des Servers ist das Instanzieren eines remote Objekts und das Publizieren dieses Objekts. Beim Konstruieren wird gleichzeitig das Objekt 'exportiert', also für eintreffende Anfragen vorbereitet.

In RMI setzen wir die Tie Technik / das Delegations Pattern ein.

```
_ServantTie tie = (_Servant_Tie)Util.getTie( servantRef );
      String hservantId = "RMIoverIPmitPOA";
      byte[] id = servantId.getBytes();
      tPOA.activate_object_with_id( id, tie );
```


© J.M.Joller

1.9.5.1.3.4. Piublizieren der remote Referenz mit fixer ID

Das Objekt, die Instanz des Servants, wird im Namensraum registriert und somit publiziert:

```
// 5) Publizieren der Objektreferenz mit der selben Objekt ID wie
      //    das Tie Objekt
      System.out.println("[Server]NamingContext - Instanzierung");
      Context initialNamingContext = new InitialContext();
      System.out.println("[Server]NamingContext - POAServer eintragen");
      initialNamingContext.rebind("POAServer",
        tPOA.create_reference_with_id(id,
          tie._all_interfaces(tPOA,id)[0]) );
```

Zu den Argumenten des `rebind()` Aufrufes:

- das erste Argument der Methode ist der Name, unter dem das remote Objekt auf dem Server angesprochen werden kann.
- das zweite Argument ist die Objekt-ID des zu bindenden Objekts.

1.9.5.1.3.5. Warten auf Anfragen

Nun müssen wir nur noch die Verbindung zum ORBD als Namensdienst starten.

1.9.5.1.4. Client Programm

Unser Client ruft die Methode des im Server instanzierten Servant auf:

```
package rmooveriiopmitpoa;

import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;
import javax.rmi.*;
import java.util.Vector;
import javax.naming.NamingException;
import javax.naming.InitialContext;
import javax.naming.Context;

/**
 * Title:        RMI over IIOP
 * Description:  RMI over IIOP mit POA
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class Client {
  public static void  main( String args[] ) {
    Context ic=null;
    Object objRef=null;
    Interface hi;
    System.out.println("[Client]Start");

    try {
      System.out.println("[Client]Naming Context - Instanzieren");
      ic = new InitialContext();
    } catch (NamingException e) {
```


© J.M.Joller

```
      System.out.println("[Client]Naming Context - Instanzieren schlug
fehl" + e);
      e.printStackTrace();
      System.exit(2);
    }

    // 1) Objektreferenz für den Namensdienst
    //    (JNDI)
    try {
      System.out.println("[Client]Naming Context - Lookup des POAServers");
      objRef = ic.lookup("POAServer");
    } catch (NamingException e) {
      System.out.println("[Client]Naming Context - Lookup des POAServers
schlug fehl "+e);
      e.printStackTrace();
      System.exit(4);
    }

    // 2) Narrow der Objektreferenz auf den gewünschten Datentyp
    try {
      hi = (Interface) PortableRemoteObject.narrow(
        objRef, Interface.class);
      System.out.println("[Client]Aufruf der remote Methode - "+
                              hi.wieGehts("Wie geht's?") );
    } catch (ClassCastException e) {
      System.out.println("[Client]Aufruf der remote Methode schlug fehl "+
e);
      e.printStackTrace();
      System.exit(4);
      } catch( Exception e ) {
            System.out.println("[Client]Aufruf der remote Methode schlug
total fehl ");
            System.out.println("[Client]Exception "+e);
            e.printStackTrace( );
            System.exit(6);
        }
  }
}
```

Beim Schritt `Naming.lookup()` geschieht folgendes:

- der Namensdienst liefert eine `_Servant_Stub` , welche an den Namen gebunden ist.
- dieser Stub wird an den Client geliefert.

Der Client ruft dann die remote Methode auf und erhält die gewünschte Antwort.

1.9.5.2. Übersetzen der Java Programme

Das Übersetzen der Java Klassen geschieht gleich im JBuilder. Allerdings hat JBuilder zum Teil noch Probleme mit dem -POA Switch beim RMIC. Deswegen steht für diesen Schritt eine (eventuell nicht benötigte) Batch Datei zur Verfügung.

1.9.5.2.1. RMIC zur Generierung von Stubs

Der passende Befehl für die Generierung der Stubs mit POA und IIOP lautet:

```
rmic -poa -iiop HelloImpl
```


© J.M.Joller

1.9.5.3. Starten des Namensdienstes, von Server und Client

Für diesen Schritt stehen Batch Skripte zur Verfügung:

1.9.5.3.1. Starten des Namensdienstes

```
@echo off
Rem
Rem ----------------------------------------------------------------------
call UmgebungSetzen.bat
Rem ----------------------------------------------------------------------
Rem
echo Der Name Server wird an Port 1050 (ORBInitialPort) gestartet
echo Der Server kann nur mit CTRL/C gestoppt werden
%JAVA_HOME%\bin\orbd -ORBInitialPort 1050 -ORBInitialHost localhost -
ORBDebug orbd
pause
```

1.9.5.3.2. Starten des Servers

```
@echo off
Rem
Rem ----------------------------------------------------------------------
call UmgebungSetzen.bat
Rem ----------------------------------------------------------------------
Rem
@echo Starten des Servers...
Rem
Rem set JAVA_HOME=d:\jdk1.4
%JAVA_HOME%\bin\java -classpath . -
Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory      -
Djava.naming.provider.url=iiop://localhost:1050 rmooveriiopmitpoa.Server
pause
```

1.9.5.3.3. Starten des Clients

```
@echo off
Rem
Rem ----------------------------------------------------------------------
call UmgebungSetzen.bat
Rem ----------------------------------------------------------------------
Rem
@echo Starten des RMI Clients mit IIOP...
Rem set JAVA_HOME=d:\jdk1.4
%JAVA_HOME%\bin\java -classpath . -
Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory -
Djava.naming.provider.url=iiop://localhost:1050 rmooveriiopmitpoa.Client
pause
```


© J.M.Joller

1.9.5.4. Ausgaben

1.9.5.4.1. Namensdienst

```
Der Name Server wird an Port 1050 (ORBInitialPort) gestartet
Der Server kann nur mit CTRL/C gestoppt werden
ORBD begins initialization.
ORBD is ready.
ORBD serverid: 1
activation dbdir:
D:\UnterrichtsUnterlagen\ParalleleUndVerteilteSysteme\CORBAmitPOA\Beispiele
\RMIoverIIOP\RMOoverIIOPmitPOA\.\orb.db
activation port: 1049
activation Server Polling Time: 1000 milli-seconds
activation Server Startup Delay: 1000 milli-seconds
```

1.9.5.4.2. Server

```
Starten des Servers...
[Server]Start
[Server]Setzen der Laufzeit Eigenschaften
[Server]POA
[Server]POA - setzen des Roots
[Server]POA - Definition der Policies
[Server]POA - Definition von 'MeinTransienterPOA'
[Server]POA - Aktivierung
[Server]Servant - Instanzierung
[Server]Servant - Tie Instanzierung
[Server]Servant - Aktivierung
[Server]NamingContext - Instanzierung
[Server]NamingContext - POAServer eintragen
[Server]warten auf Clients
[Servant]wieGehts()
```

1.9.5.4.3. Client

```
Starten des RMI Clients mit IIOP...
[Client]Start
[Client]Naming Context - Instanzieren
[Client]Naming Context - Lookup des POAServers
[Client]Aufruf der remote Methode - Mir geht's gut!
Danke fuer Deine Frage : Wie geht's?
```


© J.M.Joller

CORBA - MIT DEM PORTABLE OBJECT ADAPTER (POA) 1
1.1. CORBA - ARCHITEKTUR, BASISMECHANISMEN, DIENSTE 1
1.1.1. Einführung 1
1.1.2. Die Object Management Group (OMG) 4
1.1.3. Die Object Management Architecture (OMA) 4
1.1.4. Die CORBA-Architektur 6
1.1.5. Die CORBA 2.0 Spezifikation 7
1.1.6. Die Interface Definition Language (IDL) 8
1.1.7. CORBA-Objektreferenzen 9
1.1.8. Statisches und dynamisches CORBA-Objektmodell 9
1.1.9. CORBA Request-Typen und deren Semantik 11
1.1.10. Der Basic Object Adapter (BOA) 12
1.1.11. Das Interface Repository 13
1.1.12. Der Portable Object Adapter (POA) 14
1.1.13. Die CORBA-Dienste (COSS) 16
1.1.13.1. Naming Service 16
1.1.13.2. Relationship Service 16
1.1.13.3. Life Cycle Service 16
1.1.13.4. Query Service 16
1.1.13.5. Collection Service 16
1.1.13.6. Event Service 16
1.1.13.7. Trader Service 17
1.1.13.8. Transaktions Service 17
1.1.13.9. Concurrency Control Service 17
1.1.13.10. Persistence Service 18
1.1.13.11. Externalization Service 18
1.1.13.12. Time Service 18
1.1.13.13. Security Service 18
1.1.14. Zusammenfassung 18
1.1.15. Literatur 20
1.2. BEISPIELE MIT DEM PORTABLE OBJECT ADAPTER (POA) 21
1.2.1. Definition des Interfaces 22
1.2.1.1. Übersetzen des Interfaces 23
1.2.1.2. Generierte Dateien 23
1.2.1.2.1. Inhalt der generierten Dateien - ein Beispiel 24
1.2.2. Implementation des Servers und des Servants 32
1.2.2.1. Den Server verstehen 34
1.2.2.2. Server.java verstehen 35
1.2.2.2.1. Vorbereitende Arbeiten 35
1.2.2.3. Übersetzen des Servers 39
1.2.3. Implementation der Client Applikation - Transienter Client 40
1.2.3.1. Übersetzen des Clients 41
1.2.4. Generierung der Applikation 41
1.2.5. Starten der Applikation 42
1.2.5.1. Starten des Name Service 42
1.2.5.2. Starten des Servers 43
1.2.5.3. Starten des Clients 43
1.2.5.4. Beispielausgabe des Servers 44
1.2.5.5. Beispielausgabe des Clients 44
1.3. SERVER-SEITIGES POA MODELL FÜR PERSISTENZ 45
1.3.1. Transiente und Persistente Objekte 45
1.3.2. Definition des Interfaces 46
1.3.2.1. Übersetzen des Interfaces 46
1.3.2.2. Generierte Klassen 46
1.3.3. Implementation des Servants 47
1.3.4. Implementation des Servers 48
1.3.4.1. Übersetzen des Servers 50
1.3.5. Implementation der Clients Applikation 51
1.3.5.1. Übersetzen des Clients 52
1.3.6. Starten der Applikation 53
1.4. SERVERSEITIGES POA MODELL MIT TRANSIENTEM SERVER 57

© J.M.Joller

1.5. POA TIE (DELEGATION) MODELL MIT TRANSIENTEM SERVER 57
1.5.1. Definition des Interfaces *57*
1.5.1.1. Übersetzen des Interfaces 58
1.5.1.2. Generierte Klassen 58
1.5.2. Implementation des Servants *60*
1.5.2.1. Übersetzen des Servants 60
1.5.3. Implementation des Servers *61*
1.5.3.1. Übersetzen des Servers 62
1.5.4. Implementation der Client Applikation *63*
1.5.4.1. Übersetzen des Clients 64
1.5.5. Starten der Applikation *64*
1.6. IMPLBASE (VERERBUNG) BASIERTER TRANSIENTER SERVER 66
1.6.1. Definition des Interfaces *66*
1.6.1.1. Übersetzen des Interfaces 66
1.6.1.2. Generierte Dateien 67
1.6.2. Implementation des Servants *68*
1.6.3. Implementation des Servers *69*
1.6.3.1. Übersetzen des Servers 70
1.6.4. Implementation der Client Applikation *71*
1.6.4.1. Übersetzen der Clie nt Applikation 72
1.6.5. Starten der Applikation *72*
1.7. VERTEILEN DER KLASSEN - BEISPIELE MIT MEHREREN RECHNERN 72
1.8. NEUERE FEATURES 73
1.8.1. Der Portable Object Adapter *73*
1.8.1.1. Was ist der POA? 73
1.8.1.2. Das abstrakte POA Modell 73
1.8.1.3. Einsatz des POA 75
1.8.1.4. Bestimmen des root POA 75
1.8.1.5. Definieren der POA Policies 75
1.8.1.5.1. Thread Policy 76
1.8.1.5.2. Lifespan Policy 76
1.8.1.5.3. Object Id Uniqueness Policy 76
1.8.1.5.4. Id Assignment Policy 77
1.8.1.5.5. Servant Retention Policy 77
1.8.1.5.6. Request Processing Policy 77
1.8.1.5.7. Implicit Activation Policy 77
1.8.1.6. Kreieren des POA 78
1.8.1.7. Aktivieren des POAManager 78
1.8.1.8. Aktivierung des Servants 79
1.8.1.9. Kreieren einer Objektreferenz 80
1.8.2. Adapter Activators *81*
1.8.3. Servant Managers *81*
1.8.4. Naming Service *82*
1.8.4.1. Übersicht über den COS Naming Service 82
1.8.4.2. Interoperable Naming Service 83
1.8.4.2.1. Bootstrap Optionen für den ORB 85
1.8.4.3. Einsatz des Naming Service 85
1.8.4.3.1. Eintragen eines Objekts in den Namensraum 86
1.8.4.3.2. Objekt im Namensraum auflösen 88
1.8.4.3.3. Namensraum durchsuchen 89
1.8.4.3.4. Starten des Namensdienstes 91
1.8.4.3.5. Stoppen des Namensservices 91
1.8.5. General InterORB Protocol (GIOP) *92*
1.8.6. Offizielle Spezifikationen für CORBA und Support in J2SE 1.4 ..... *92*
1.8.6.1. Spezifikationen im Detail 92
1.8.6.2. Tools 93
1.9. ERGÄNZUNG - RMI OVER IIOP MIT POA 94
1.9.1. Wann macht RMI oder IIOP Sinn? *94*
1.9.2. Welche anderen Möglichkeiten zur Programmierung Verteilter Systeme habe ich noch? *94*
1.9.3. Was ist neu in J2SE 1.4 RMI IIOP *95*
1.9.4. RMI IIOP Beispiel *95*
1.9.4.1. Definition der remote Klassen als Interfaces in Java 95
1.9.4.2. Implementieren der Interfaces 96
1.9.4.2.1. Mindestens eine remote *Schnittstelle* implementieren 97
1.9.4.2.2. *Konstruktor* für remote Objekte definieren 97

© J.M.Joller

1.9.4.2.3. Implementationen für die remote *Methoden* liefern. 98
1.9.4.3. Schreiben des Servers 98
1.9.4.3.1. Instanzieren des Servants 99
1.9.4.3.2. Eintragen in den Namensdienst.... 99
1.9.4.4. Schreiben des Clients 100
1.9.4.5. Übersetzen des Beispiels 101
1.9.4.6. Testen der Applikation 101
1.9.4.6.1. Starten des Namensdienstes.... 101
1.9.4.6.2. Starten des Servers.... 102
1.9.4.6.3. Starten des Clients 102
1.9.5. RMI-IIOP mit POA-basiertem Servermodell 103
1.9.5.1. Schreiben der Applikation 103
1.9.5.1.1. Definition des remote Interfaces in Java.... 103
1.9.5.1.2. Implementieren der Interfaces 104
1.9.5.1.2.1. Mindestens eine remote Schnittstelle implementieren 105
1.9.5.1.2.2. Konstruktor für remote Objekte definieren 105
1.9.5.1.2.3. Implementationen für die remote Methoden liefern. 106
1.9.5.1.3. Schreiben der Server Klasse 106
1.9.5.1.3.1. Kreieren des POA mit passender Policy 108
1.9.5.1.3.2. Aktivieren des POA Managers.... 108
1.9.5.1.3.3. Kreieren eines remote Objekts 108
1.9.5.1.3.4. Piublizieren der remote Referenz mit fixer ID 109
1.9.5.1.3.5. Warten auf Anfragen.... 109
1.9.5.1.4. Client Programm.... 109
1.9.5.2. Übersetzen der Java Programme 110
1.9.5.2.1. RMIC zur Generierung von Stubs 110
1.9.5.3. Starten des Namensdienstes, von Server und Client.... 111
1.9.5.3.1. Starten des Namensdienstes.... 111
1.9.5.3.2. Starten des Servers.... 111
1.9.5.3.3. Starten des Clients 111
1.9.5.4. Ausgaben.... 112
1.9.5.4.1. Namensdienst 112
1.9.5.4.2. Server.... 112
1.9.5.4.3. Client 112

© J.M.Joller