Diplomarbeit

Entwurf und Implementierung eines Portable Object Adapters für JacORB

Reimo Tiedemann

15. Januar 2000

Freie Universität Berlin
Fachbereich Mathematik und Informatik
Institut für Informatik

Diplomarbeit von

Reimo Tiedemann
Mainzer Str. 21a
D - 10247 Berlin
email: tiedeman@inf.fu-berlin.de

Inhaltsverzeichnis

1. Einleitung

Durch die fortschreitende Vernetzung von Rechnersystemen liegt der Wunsch nahe, die verbundenen Ressourcen auch in einer verteilten Anwendungssoftware zu nutzen. Die Einsatzmöglichkeiten sind vielfältig; das Spektrum reicht von der reinen Nutzung der im Netz verfügbaren Rechenkapazität, über die Bereitstellung von Diensten, die nur an einem bestimmten Ort zur Verfügung stehen (bsw. ein Bankrechner, der den Zugriff auf den Kontostand ermöglicht), bis hin zur Gewährleistung von Fehlertoleranz durch die Einführung redundanter Server, die eine Dienstleistung übernehmen können, wenn eine andere Station im Netz ausfällt.

Verteilte Anwendungen sind schwieriger zu entwickeln und zu pflegen, je mehr sich der Entwickler mit den Details der Netzwerkprogrammierung beschäftigen muß. Selbst heutzutage weit verbreitete Programmiersprachen, wie C/C++, bieten dem Entwickler nur ein Abstraktionsniveau auf der Ebene (zuverlässiger) Netzverbindungen zu einem anderen Rechner (Serverport), für die Zusammenstellung und Kodierung einzelner Nachrichten muß er selbst sorgen. Die Benutzung des Nachrichtenparadigmas (send, receive) steht im Konflikt mit dem gewohnten prozeduralen bzw. objektorientierten Paradigma (Programmiermodell) und ist zudem mühsam und fehleranfällig. Der Entwickler möchte eine verteilte Anwendung in dem selben Programmiermodell entwickeln, das er im lokalen Anwendungsfall gewohnt ist. Legt man der Implementierungssprache das Objektmodell zugrunde, dann soll ein verteiltes Objektmodell die Zusammenarbeit mehrerer Objekte auf potentiell unterschiedlichen Rechner in einem Netzwerk ermöglichen, ohne daß sich der Entwickler mit der Existenz von Netzwerkverbindungen beschäftigen und das von ihm bevorzugte Programmiermodell durch neue verteilungsspezifische Instruktionen erweitern muß.

Um auf einem höheren Abstraktionsniveau entwickeln zu können, als es das Nachrichtenparadigma zuläßt, wird die Unterstützung durch eine Systemsoftware benötigt, von der ein Teil auf jeder beteiligten Station des Netzes läuft, die sich untereinander kennen und wissen, wie sie miteinander kommunizieren müssen. Diese zusätzliche Softwareschicht ist auch unter der Bezeichnung *Middleware* bekannt. Ein Beispiel für eine einfache Middleware-Architektur ist Suns *Remote Procedure Call* (Sun RPC) [38]. Der RPC-Mechanismus erlaubt die Implementierung eines Client/Server-Systems, wobei der Client einen Dienst eines entfernten Servers aufrufen kann. Auf beiden Seiten wird die Abstraktion eines normalen Prozeduraufrufes geschaffen. Das *Distributed Component Object Model* (DCOM) [33] der Firma Microsoft ist ein Beispiel für ein verteiltes Objektmodell, das zwar wesentlich mehr Möglichkeiten bietet, aber auf die Windows-Welt beschränkt ist.

Eine weitere Eigenschaft dieser Softwareschicht sollte darin bestehen, die Heterogenität der darunterliegenden Systeme zu verbergen. Dazu muß diese Software auf den einzelnen Systemen unterschiedlich realisiert sein. Nach oben werden dem

Entwickler auf jedem System die gleichen Dienste angeboten (Verteilungsplattform). Die Realität sieht so aus, daß auch in Zukunft die Netzwerk- und Computerhardware bzw. System- und Anwendungssoftware von verschiedenen Anbietern bereitgestellt werden. Ebenso sind die ständige Weiterentwicklung der Systeme, die Veränderung der Organisationsstrukturen und Kommunikationsbedürfnisse Quelle für diese Heterogenität. Die Interoperabilität zwischen den verteilten Systemkomponenten ist die Voraussetzung, damit Anwendungen die existierenden Infrastrukturen nutzen können.

Die Portabilität der Anwendungen ist ein weiteres wichtiges Ziel, so daß die Konzepte, die bei der Entwicklung einer Anwendung benutzt wurden, auch auf unterschiedlichen Systemen zur Verfügung stehen. Die Voraussetzung für Interoperabilität und Portabilität sind Standards. Um die Heterogenität existierender System zu überwinden, sind offene Standards von großem Vorteil, da weniger die kaufmännischen Interessen einzelner Firmen der Gesamtentwicklung entgegenwirken.

Mit der *Common Object Request Broker Architecture* (CORBA) existiert ein offener Standard für ein verteiltes Objektsystem, der die Entwicklung interoperabler und portabler Anwendungen erlaubt. Neben der Offenheit eines Standards ist auch seine Akzeptanz in der Industrie und bei den Anwendern und die daraus folgende Marktstellung der Produkte von Bedeutung. Durch einen Zusammenschluß vieler namhafter Systemanbieter, die die Entwicklung von CORBA fördern, sind die Voraussetzungen dafür gegeben. DCOM ist ein Beispiel für einen proprietären Standard, dessen Entwicklung von einem einzelnen Unternehmen betrieben wird, aber aufgrund der großen Verbreitung der Microsoft-Betriebssystemsoftware eine vergleichbare Stellung auf dem Markt einnimmt. Einen Vergleich zwischen CORBA und DCOM findet man in [31].

Die CORBA-Spezifikation definiert ein Framework, das die plattform- und programmiersprachenunabhänge Entwicklung verteilter objektorientierer Anwendungen ermöglicht. Ein *Object Request Broker* (ORB) dient als Vermittler zwischen einem Client, der eine CORBA-Objektreferenz in den Händen hält, und einer Objekt-Implementierung, die von einem Server bereitgestellt wird. Die Plattform- und Programmiersprachenunabhängigkeit wird durch die Definition der Objekt-Schnittstellen in einer *Interface Definition Language* (IDL) gewährleistet. Aus dieser IDL-Definition generiert ein IDL-Compiler programmiersprachenspezifische Schnittstellen und Vertreter-Klassen, die für die Anwendungsentwicklung und zur Realisierung einer CORBA-Objektreferenz verwendet werden. Die Abbildung der in IDL definierten Schnittstellen ist für eine Programmiersprache in einem *Language Mapping* festgelegt.

Ein Client kann auf einer CORBA-Objektreferenz Aufrufe durchführen, die vom ORB an die Objekt-Implementierung weitergeleitet werden. Diese Aufrufe unterscheiden sich nicht von ganz normalen Objektaufrufen auf einer lokalen Objektreferenz. Auf der Serverseite muß die Anwendung die Implementierung der Objekte beim ORB registrieren. Um für die Anwendung den bestmöglichen Service bereitzustellen, benutzt der ORB das Konzept eines Objekt-Adapters. Dieser dient als Verbindungstück zwischen der Objekt-Implementierung und dem ORB. Der Objekt-Adapter erlaubt auf der eine Seite die flexible Verwaltung der Objekt-Implementierungen, und auf der anderen leitet er die vom ORB vermittelten Objektaufrufe an die Implementierung weiter.

Die Offenheit des CORBA-Standards ermöglichte dessen Implementierung an der Freien Universität Berlin. Seit 1997 wird JacORB im wesentlichen von Gerald Brose als eine freie Java-Implementierung des CORBA-Standards entwickelt [12]. Diese enthielt eine einfache Version eines als Standard-Objekt-Adapters spezifizierten *Basic Object Adapters* (BOA). Der BOA wurde allgemein als unterspezifiziert angesehen. Erweiterungen verschiedener ORB-Hersteller führten zu ernsten Problemen hinsichtlich der Portabilität von Serveranwendungen. Mit dem Voranschreiten des Spezifikationsprozesses wurde der BOA aus der CORBA-Spezifikation entfernt und durch einen neuen, von der Funktionalität her wesentlich erweiterten Objekt-Adapter, ersetzt. Da dieser die Probleme mit dem BOA in puncto Portabilität beheben sollte, bekam er den Namen *Portable Object Adapter* (POA).

Ziel dieser Arbeit war es, einen POA für JacORB zu entwickeln. Sie war damit Bestandteil des Bestrebens, JacORB an die mittlerweile fortgeschrittene Standardisierung von CORBA anzupassen. Bei der Implementierung eines POAs stand die Umsetzbarkeit der Spezifikation auf dem Prüfstand. Außerdem mußten die Java-spezifischen Aspekte bei der Integration des POAs in den ORB beachtet werden. Maßgeblich dafür war die Einbeziehung des aktuellen IDL/Java-Mappings.

Der POA ist Bestandteil der ORB-Implementierung. Demzufolge war der Entwurf des POAs in die Gesamtarchitektur von JacORB einzupassen. Da der POA mit dem ORB kooperiert, mußten die Schnittstellen zwischen POA und ORB neu entworfen und implementiert werden. Für die Implementierung des POAs wurde die zum Zeitpunkt des Schreibens aktuelle CORBA-Version herangezogen (*Version 2.3-Core final*) [22]. Gleiches gilt für das Java/IDL-Mapping (Juni 1999) [26].

Dieser Einleitung wird sich ein Überblickskapitel zu CORBA anschließen, in dem die Architektur vorgestellt und das Konzept des Objekt-Adapters eingeführt wird. Dieses Kapitel soll die Gesamtzusammenhänge darstellen und eine Grundlage für die folgenden detaillierten Betrachtungen geben. Am Ende dieses Kapitels werden einige Anforderungen formuliert, die an einen Objekt-Adapter gestellt werden.

Im Kapitel 3 folgt eine Vorstellung des BOAs, dem ersten Versuch der OMG einen Standard-Objekt-Adapter zu etablieren. Die Erfahrungen mit dem BOA bilden den Hintergrund für die Entwicklung zu einem zweiten Objekt-Adapter, dem POA. Dieses Kapitel soll die Unzulänglichkeiten des BOAs aufzeigen und somit die Entwicklung zum POA erklären.

Das Kapitel 4 beschäftigt sich ausführlich mit dem Entwurf des POAs. Neben einer gründlichen Beschreibung der POA-Spezifikation werden alle relevanten Prozesse im Lebenszyklus eines POAs und deren Konsequenzen für eine CORBA-Anwendung dargestellt. Mit diesem Bezug zur Anwendungsentwicklung sollen die Einsatzmöglichkeiten des POAs aufgezeigt werden. Am Ende dieses Kapitels steht eine Bewertung des POA-Entwurfs und damit eine Einschätzung, ob die Spezifikation den Erwartungen an einen Standard-Objekt-Adapter gerecht wird. In Verbindung mit dem Kapitel 5, das die POA-relevanten Erläuterungen zum IDL/Java-Mapping liefert, bildet es die Grundlage für die darauffolgende Beschreibung der POA-Implementierung für JacORB.

Die Beschreibung der POA-Implementierung im Kapitel 6 liefert einen Überblick über den vorgenommenen Implementierungsentwurf, beschreibt die Schnittstellen zur JacORB-Implementierung und geht im Anschluß daran ausführlich auf einzelne Implementierungsprobleme ein. Am Ende des Kapitels wird eine Bewertung der POA-Implementierung und eine Beschreibung ihrer Erweiterungsmöglichkeiten geboten.

Die Arbeit schließt mit einem Vergleich existierender POA-Implementierungen. Nach einem Blick auf die momentane Situation auf dem Java/CORBA-Markt wird es interessant sein, herauszufinden, ob JacORB und vor allem sein POA einem Vergleich mit anderen Java-ORBs standhält.

2. CORBA-Einführung

Im Jahr 1989 schlossen sich einige namhafte Softwarefirmen zusammen, um Objekttechnologien zu fördern und um die Komplexität und Kosten von Software zu reduzieren. Es sollte ein offener Standard für die Entwicklung portabler verteilter Objekt-Anwendungen in heterogenen Umgebungen geschaffen werden. Die so gegründete *Object Management Group* (OMG), die bis heute mehr als 800 Mitglieder vereinigt, von großen Industrieunternehmen[1] bis hin zu Institutionen aus dem akademischen Umfeld, ist damit das weltweit größte herstellerübergreifende Software-Konsortium. Im Mittelpunkt ihrer Arbeit steht die *Object Management Architecture* (OMA), die eine Infrastruktur für verteilte Objekt-Systeme definiert.

Bevor eine Spezifikation verabschiedet wird, publiziert die OMG ein entsprechendes Papier, ein *Request for Proposals* (RFP). Die Mitglieder können daraufhin Vorschläge einreichen. Nach einer Phase, in der mehrere unterschiedliche Vorschläge eingehen und in der diese in Foren diskutiert werden, wird letztendlich unter den Mitgliedern über jeden dieser Vorschläge abgestimmt. Im Unterschied zu anderen Industriestandards, wie sie meist von führenden Softwareunternehmen auf Grund ihrer Monopolstellung am Markt gesetzt werden, handelt es sich ausschließlich um Spezifikationen und keine fertigen Produkte. Die OMG beschließt offene Standards, die resultierenden Dokumente sind über das Internet frei verfügbar und können von jedem implementiert werden. Natürlich hat ein Vorschlag, dessen Realisierbarkeit mit einem konkret vorliegenden Produkt schon bewiesen wurde, in dem beschriebenen Auswahlverfahren bessere Chancen angenommen zu werden, als eine rein akademische Lösung. Dies ist allein durch die hohe Anzahl an Industrieunternehmen unter den Mitgliedern sichergestellt.

Die OMA liefert ein Objekt-Modell und ein Referenz-Modell zur plattformunabhängigen Realisierung verteilter Software-Objekte [17]. Das Objekt-Modell beschreibt die Konzepte eines verteilten Objekt-Systems und gibt auch eine Definition für ein Objekt innerhalb des Modells: *"An object is an identifiable, encapsulated entity that provides one or more services that can be requested by a client".* Ein Client stellt eine Anfrage an ein Objekt und greift auf die Objekt-Dienste über eine wohldefinierte Schnittstelle zu. Die Implementierung eines Objektes und sein momentaner Standort bleiben einem Client verborgen.

Das Kernstück der OMA bildet der *Object Request Broker* (ORB). Dieser erlaubt die Kommunikation zwischen Clients und Server-Objekten. Anfragen werden vom Client über diesen "Objekt-Bus" an ein Objekt transportiert. Der ORB bietet zudem weitere Basisdienste, wie z.B. die für den Client transparente Aktivierung von Objekten. Der ORB stellt sowohl Client- als auch Server-Anwendungen eine Schnittstelle bereit, um die Basisdienste nutzen zu können. Die Spezifikation des ORBs inklusive seiner Basisdienste ist in der *Common Object Request Broker Architecture*

[1] IBM, Sun, DEC, Hewlett-Packard, Siemens, Microsoft u.v.a.

(CORBA) festgelegt [27]. Mit dieser Kernfunktionalität wird die Abstraktion eines CORBA-Objektes geschaffen, auf der weitergehende Dienste aufbauen können. Das Referenz-Modell gruppiert die denkbaren Objekt-Schnittstellen innerhalb von vier Kategorien und verbindet sie durch den ORB (siehe Abbildung 2.1).

Object Services Grundlegende, in einer *CORBA Service Specification* definierte, horizontale Dienste, die von vielen Anwendungen genutzt werden (z.B. Name-Service [24], Lifecycle-Service [18], Transaction-Service [18] usw.)

Common Facilities Höherwertige horizontale Dienste, deren Einsatz für ein breites Spektrum von Anwendungen denkbar ist (z.B. GUI, Druckdienst)

Domain Interfaces Anwendungs- bzw. branchenspezifische Dienste, für die eine spezielle *Domain Task Force Group* eingerichtet wird (z.B. für Finanzdienstleistungen)

Application Interfaces Diese Kategorie versammelt alle nicht von der OMG standardisierten Anwendungsobjekte. Tauchen bestimmte Objekte bzw. deren Funktionalität in mehreren Anwendungen auf, so sind sie Kandidaten für eine Standardisierung und Einordnung in die oberen drei Katagorien.

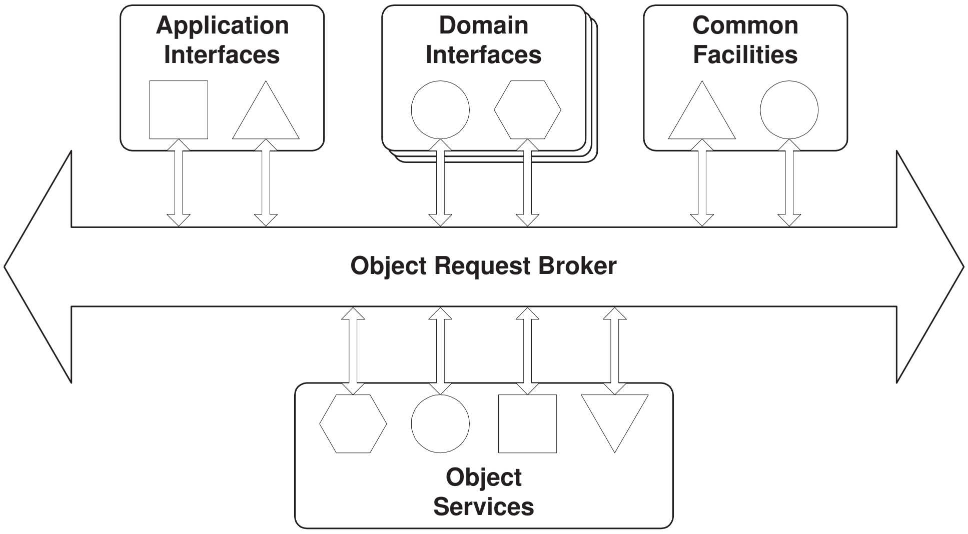


Abbildung 2.1.: OMA Referenz-Modell

Die CORBA-Architektur folgt dem klassischen Client/Server-Prinzip. Serverprozesse "führen Objekte" aus. Ein Client kann aber auch gleichzeitig als Server-Objekt für ein weiteres Objekt fungieren. So lassen sich komplexe, verteilte Objekt-Systeme realisieren. Um eine Operation auf einem Server-Objekt aufrufen zu können, muß ein Client auf eine CORBA-Objektreferenz zugreifen können. Dabei handelt es sich um ein Vertreter-Objekt innerhalb des Client-Prozesses *(Proxy* bzw. *Stub)*, das die entfernte Objekt-Implementierung im lokalen Adressraum vertritt und die Informationen zur Identifizierung eines CORBA-Objektes und zur Adressierung und Lokalisierung der Objekt-Implementierung enthält. Die Details der notwendigen Netzwerkadresse werden in der Objektreferenz gekapselt. Die Zusammensetzung einer Objektreferenz ist für einen Client *opaque*. Er kann weder auf den Inhalt einer Objektreferenz zugreifen, noch ihn modifizieren. Um Lokalitätstransparenz für den Client zu erreichen, ist es notwendig, daß dieser eine Schnittstelle referenziert, die anstelle einer Objekt-Implementierung ebenso durch das Vertreter-Objekt bedient werden kann. Die Schnittstelle, die der Client sieht, ist unabhängig von der Position des Zielobjektes und der Programmiersprache, in der das Objekt realisiert ist. Mehrere Clients können durch den Austausch der Objektreferenz dasselbe entfernte Objekt aufrufen.

Neben der Adressierung der Objekt-Implementierung übernimmt der Client-Stub-Code die Zusammenstellung und Verpackung der Aufruf-Parameter *(Marshaling)*. Die clientseitige ORB-Runtime-Umgebung kann nun die Anfrage an den Server versenden. Die serverseitige ORB-Runtime-Umgebung liest die Anfrage vom Netzwerk und vermittelt sie an die zuständige Objekt-Implementierung. Für die Serverseite wird auf ähnliche Weise Code zur Verfügung gestellt, der das *Unmarshaling* der Aufruf-Parameter übernimmt und die Anfrage auf die gewünschte Operation der Objekt-Implementierung abbildet *(Skeleton)*. Nach Durchführung des Operationsaufrufes werden die Ergebnisse wieder verpackt und an den Client zurückgesendet. Der Stub-Code sorgt für das Entpacken der Ergebnisse und die Rückgabe an den Aufrufer der Operation. Die Objektreferenz bietet dem Client ein *Handle* auf ein abstraktes CORBA-Objekt, vergleichbar mit einem Programmiersprachen-Zeiger auf ein lokales Objekt. Über den physikalischen Anfragefluß wird der Aufruf auf dem Objekt verwirklicht.

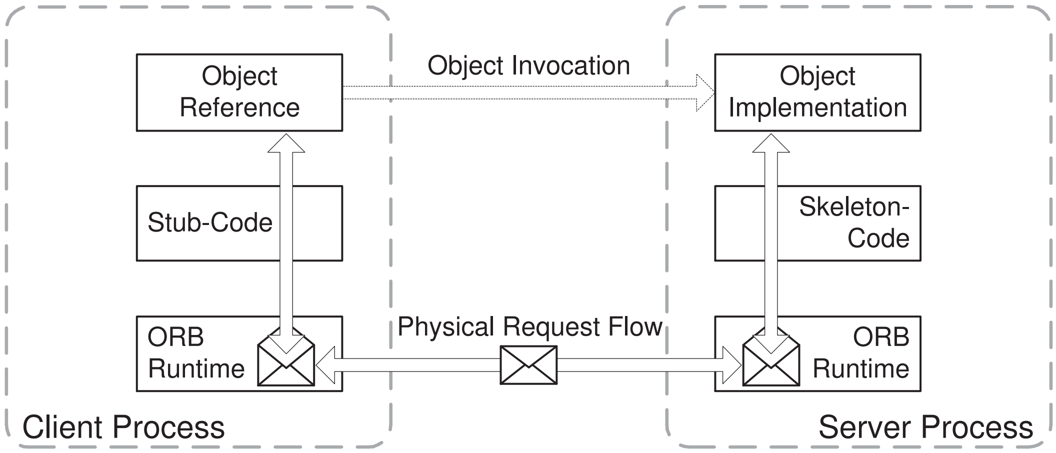


Abbildung 2.2.: Grundlegendes Vertreter-Prinzip

Das in der Abbildung 2.2 dargestellte Vertreter-Prinzip stellt keine Besonderheit von CORBA dar, vielmehr bildet es die Gemeinsamkeit mit anderen Middleware-Plattformen, wie *Suns Remote Procedure Call* (RPC) [38] oder dem *Distributed Component Object Model (*DCOM) von Microsoft [33]. Die Unterschiede zwischen diesen Systemen liegen im wesentlichen in dem zugrunde liegenden Objektmodell, sofern es sich um ein verteiltes Objektsystem handelt, den zusätzlich standardisierten Diensten, dem Grad der von ihnen gebotenen Verteilungstransparenz und in solchen Punkten wie Plattformenunabhängigkeit, Flexibilität und Konfigurierbarkeit hinsichtlich ihrer Anwendung.

2.1. Interface-Definition

Um Aufrufe auf einem entfernten Objekt durchführen zu können, muß ein Client die öffentliche Schnittstelle des Objektes kennen. Zur Definition einer Objekt-Schnittstelle dient die *Interface Definition Language* (IDL) der OMG.

2.1.1. Interface Definition Language

Die IDL ist eine rein deklarative Sprache. Die Verwendung der IDL anstelle einer konkreten Programmiersprache zur Beschreibung von Objekt-Schnittstellen ist der Schlüssel für die programmiersprachen- und plattformunabhängige Bereitstellung von Server-Objekten. Ein IDL-Interface definiert einen Satz von Operationen und Attributen.

```
module finance {
    interface Account {
        exception Overdraft {
            float deficit;
        };
        readonly attribute string holder;
        readonly attribute float balance;
        void deposit(in float amount);
        void withdraw(in float amount) raises (Overdraft);
    };
    interface Bank {
        Account createAccount(in string holder);
    };
};
```

Um die Plattform- und Programmiersprachenunabhängigkeit zu erreichen, müssen sich Client und Server auf ein einheitliches Datenaustauschformat mit fest definierten Datentypen einigen. Die IDL unterstützt eine Anzahl von Basistypen, die für Parameter- bzw. Ergebniswerte verwendet werden können. Neben einfachen Typen wie *Integers*, *Floats*, *Booleans* und *Strings* stehen auch zusammengesetzte Typen zur Verfügung, wie *Enums*, *Structures*, *Unions* und *Sequences* (eindimensionale Felder) und *Exceptions*. Außerdem können Interfaces selbst als Parameter- oder Ergebnistyp eingesetzt werden. Es ist auch ein allgemeiner Typ `any` vorgesehen, der Werte eines beliebigen anderen Typs zuläßt.

Vererbung ist auf der Ebene von Schnittstellen realisiert. Das CORBA-Objektmodell unterstützt Mehrfachvererbung. Jedes IDL-Interface erbt implizit vom Basis-Interface `CORBA::Object`, das einige Operationen zur Verfügung stellt, die von allen CORBA-Objekten unterstützt werden. An den Stellen, an denen Objekte mit einem bestimmten Interface erwartet werden, können somit auch Instanzen eines abgeleiteten bzw. spezialisierten Interfaces verwendet werden. Ob die Vererbungshierarchie im Falle einer objektorientierten Implementierungssprache auch auf der Implementierungsebene nachvollzogen wird bzw. werden kann, ist davon vollständig unabhängig und Sache des Implementierungsentwurfs.

Für Übergabeparameter muß eine Richtung angegeben werden. Sie sind in Eingabe- (`in`), Ausgabe- (`out`) und Eingabe/Ausgabe-Parameter (`inout`) unterteilt, je nachdem, ob die Werte nur übergeben oder nur bzw. auch zurückgeliefert werden sollen. Die IDL unterstützt ein Modulkonzept, um die Sichtbarkeitsbereiche von Bezeichnern einzuschränken. Attribute und Methoden können allerdings nicht überladen werden, da Methodennamen innerhalb eines Interfaces eindeutig sein müssen.

2.1.2. Language Mapping

Aus der IDL-Interface-Definition generiert ein vom ORB-Hersteller bereitgestellter IDL-Compiler die schon erwähnten Client-Stubs und Server-Skeletons in der gewünschten Programmiersprache. Außerdem werden dem Entwickler programmiersprachenspezifische Schnittstellen-Deklarationen zur Verfügung gestellt, bsw. innerhalb von Header-Dateien, die er für die Anwendungsentwicklung verwenden kann. Die in einem IDL-Interface verwendeten Attribute werden automatisch in entsprechende Lese- bzw. Schreiboperationen umgesetzt.

Die Abbildung der IDL-Interfaces, aller möglichen IDL-Typen und die konkrete Stub- bzw. Skeleton-Architektur sind in einem *Language Mapping* festgelegt. Es existieren standardisierte Abbildungen für die Sprachen C, C++, Smalltalk, Ada, COBOL und Java. Für eine Reihe weiterer Sprachen ist die Standardisierung in Vorbereitung, u.a. für die beliebten Script-Sprachen Perl, Python und Tcl. Die Existenz verschiedener *Language Mappings* ermöglicht die Realisierung einer verteilten Anwendung in jeweils unterschiedlichen Programmiersprachen für die Client- bzw. Serverseite.

Für die Programmiersprache Java wird ein IDL-Interface auf ein gleichnamiges Java-Interface abgebildet. Dieses wird von der ebenfalls generierten Stub-Klasse implementiert. Somit sind CORBA-Objektreferenzen in Java Exemplare dieser generierten Stub-Klasse. Neben dem IDL-Interface selbst werden auch die Schnittstellen aller geerbten IDL-Interfaces zur Verfügung gestellt, inklusive des implizit geerbten `Object`-Interfaces. Für das oben eingeführte Beispiel würden aus dem `Bank`-Interface für die Clientseite folgende Java-Interfaces und Klassen generiert werden.

```
public interface Bank extends finance.BankOperations,
            org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity {
}
public interface BankOperations {
    finance.Account createAccount(String holder);
}
public class _BankStub extends org.omg.CORBA.portable.ObjectImpl
                         implements finance.Bank {
    ...
    public finance.Account createAccount(String holder) {...}
}
```

Module werden auf Java-Packages abgebildet. Alle von der OMG spezifizierten Module bilden einheitlich Unterpakete von `org.omg`. Dabei handelt es sich um eine willkürliche Festlegung der OMG, um Namenskollisionen mit anderen Schnittstellenanbietern zu vermeiden. Die Interfaces und Klassen, die im Java-Mapping als API zur Sicherstellung von Portabilitätsanforderungen festgelegt werden, befinden sich in den Paketen `org.omg.CORBA.portable` (Clientseite) und `org.omg.CORBA.PortableServer.portable` (Serverseite).

Beim Java-Interface `org.omg.CORBA.portable.IDLEntity` handelt es sich um ein leeres *Marker Interface*, das zur Kennzeichnung einer Stub-Klasse dient, die mit Java-Objektserialisierungsfunktionalität ausgestattet ist. Dies ist nur für den umgekehrten Abbildungsprozeß (*Java to IDL*) von Bedeutung und soll hier deshalb nicht weiter betrachtet werden [29]. Die Klasse `org.omg.CORBA.portable.ObjectImpl` stellt im wesentlichen die Implementierung von `org.omg.CORBA.Object` und grundlegende Stub-Funktionalität bereit. Die Ausgliederung in eine Oberklasse vermeidet lediglich das Aufblähen des generierten Codes, da diese Funktionalität von allen Client-Stubs benötigt wird.

Die Ausgliederung der Operationen der IDL-Schnittstelle in ein separates `BankOperations`-Interface anstelle der direkten Definition innerhalb des `Bank`-Interfaces, ist für die Clientseite weniger von Bedeutung. Durch diese Trennung innerhalb der Vererbungshierarchie besteht auf der Serverseite die Möglichkeit, die für die Skeleton-Klassen nicht benötigte Funktionalität von `CORBA::Object` auszublenden. Die vom IDL-Compiler erzeugte abstrakte Skeleton-Klasse `BankPOA` übernimmt nur das `BankOperations`-Interface, dessen Implementierung der Entwickler bereitstellen muß und fügt lediglich die Skeleton-Funktionalität hinzu *(Marshaling, Dispatching)*. Auf die Skeleton-Architektur möchte ich an dieser Stelle nicht tiefer eingehen. Sie wird in dieser Arbeit noch genauestens unter die Lupe genommen (siehe Abschnitt 5.1).

2.1.3. Pseudo Objects

Eine wesentliche Eigenschaft der CORBA-Spezifikation ist es, CORBA-Dienste und selbst Kernkomponenten konsequent in IDL zu definieren. Nicht alle Interface-Definitionen beschreiben CORBA-Objekte im Sinne eines fernaufrufbaren Objektes.

Sogenannte Pseudo-Objekte wurden in CORBA eingeführt, um sprachunabhängig Komponenten beschreiben zu können, die zwar keine echten CORBA-Objekte sind, aber dennoch Objekt-Eigenschaften versammeln. Es handelt sich um programmiersprachenspezifische Objekte, deren Verwendung nur innerhalb des Prozeßraums einer Anwendung sinnvoll ist und nicht durch die allgemeinen Regeln des Language-Mappings abgebildet werden können. Ihre Abbildung wird im Language-Mapping explizit festgelegt. Diese IDL-Interfaces bekommen das Schlüsselwort "`pseudo`" vorangestellt (PIDL). Mit ihrer Definition als Pseudo-Objekt wird die Existenz einer solchen Komponente und ihre strukturelle Rolle innerhalb des CORBA-Modells festgelegt. Bsw. läßt sich auf diese Weise ihre ausschließlich lokale Verwendung oder die Beschränkung auf eine *singleton* Instanz sicherstellen. Weiterere Gründe für die Definition einer Komponente als Pseudo-Objekt wären die Verwendung von Parametertypen, die nicht in IDL definiert sind oder einfache Effizienz-Überlegungen. Prominentestes Beispiel für ein Pseudo-Objekt ist sicher der ORB selbst.

Dieses Vorgehen, "alles was nicht nach einem Objekt im CORBA-Sinne aus-schaut" zu einem Pseudo-Objekt zu machen, führte alsbald zu einer unübersichtlichen Anzahl von PIDL-Interface-Definitionen. PIDL-Interfaces offenbaren oft einen Mangel an Spezifikation, der auf das jeweilige Language-Mapping abgewälzt wurde. Sie geben den Anschein einer regulären Spezifikation. In Wirklichkeit werden aber häufig wichtige Details versteckt. Unter den Teppich gekehrt, tauchen die Probleme später an unvorhergesehener Stelle wieder auf.

In zukünftigen Spezifikationen will die OMG auf die Definition von Pseudo-Objekten verzichten [25]. Für CORBA-Dienste und selbst für Kernfunktionalität soll die Definition explizit im Rahmen der Spezifikation erfolgen. In diesem *Trade-Off* zwischen Klarheit und Effizienz soll also künftig die Klarheit die Oberhand erhalten. Wenn es darum geht, die Sichtbarkeit und Lokalität für bestimmte Systemkomponenten einzuschränken, werden diese Vorgaben innerhalb des Spezifikationstextes definiert. Eine Fehlanwendung eines solchen CORBA-Objektes zieht dann eine definierte CORBA-Exception nach sich.

2.2. Aufruf eines CORBA-Objektes

Der CORBA-Standard definiert ein Framework, das es dem Entwickler einer verteilten Objektanwendung ermöglicht, Objekt-Implementierungen beim ORB anzumelden und über den Zugriff auf CORBA-Objektreferenzen Aufrufe auf diesen Objekten durchzuführen. Auf dem bisherigen Stand der Erläuterungen könnte man CORBA als einen objektorientierten, statischen RPC-Mechanismus betrachten. Ziel dieses Abschnittes ist es, die Sicht auf einen Objekt-Aufruf zu verfeinern und weitere Aspekte des CORBA-Frameworks einzubeziehen.

In dem bisher beschriebenen Ansatz werden aus der IDL-Interface-Definition programmiersprachenspezifische, statische Stubs und Skeletons generiert, die zusammen mit der Anwendung übersetzt werden. Voraussetzung für dieses Vorgehen ist aber, daß die Schnittstellen eines Objektes zur Übersetzungszeit bekannt sind.

2.2.1. Dynamic Invocation Interface

Alternativ kann eine Anfrage an ein Objekt auch dynamisch zusammengesetzt und abgeschickt werden. Zu diesem Zweck definiert die CORBA-Spezifikation das *Dynamic Invocation Interface* (DII). Eine Anfrage an ein Zielobjekt besteht im wesentlichen aus dem Namen der auszuführenden Operation und den zu übergebenen Parameter-Werten. Das DII definiert eine generische `create_request()` Operation innerhalb des `CORBA::Object`-Interfaces, der die Anfrageparameter übergeben werden können. Für die Übergabe der Parameter-Werte sind eine Named-Value-Liste und Behälter vom Typ `Any` vorgesehen. Die `create_request()` Operation liefert ein `Request`-Objekt zurück, auf dem der eigentliche Aufruf durchgeführt werden kann (`invoke`, `send`, `get_response`).

2.2.2. Interface Repository

Um eine Anfrage zur Laufzeit zusammenstellen zu können, muß ein Client die Schnittstelle des Objektes kennen, auf deren Objektreferenz er die `create_request` Operation aufruft. Um die Parameteranzahl und -typen in Erfahrung zu bringen, kann er einen speziellen Dienst nutzen, das *Interface Repository* (IR). Das IR ist eine

Datenbank, in der IDL-Typinformationen abgelegt werden. Da es selbst als CORBA-Objekt realisiert ist, ist ein Zugriff auch für entfernte Applikationen möglich.

Die enthaltenen Typinformationen sind hierarchisch organisiert. Die einzelnen Teilkomponenten, wie Modul- und Interface-Definitionen, sind ebenfalls als CORBA-Objekte repräsentiert. Angefangen vom IR-Objekt lassen sich auf diese Weise die Typinformationen traversieren. Die Objektreferenz eines IR-Objektes muß ein Client-Programm nicht selbst in Erfahrung bringen. Sie gehört mit zu den initialen Objektreferenzen, auf die jede ORB-Instanz Zugriff hat. Jede Objektreferenz stellt mit der Schnittstelle von `CORBA::Object` die `get_interface()` Operation bereit, mit der ein Client ein `InterfaceDef`-Objekt ermitteln kann, das wiederum zu Ermittlung der Typinformationen dient. Anhand der gelesenen Beschreibung einer Objekt-Schnittstelle baut der Client eine zur Operation passende Parameterliste auf und führt die Anfrage über das DII aus.

Damit stellt sich noch eine Frage. Wie kommen die Typinformationen in das IR? Ein üblicher Weg dafür ist der, daß der IDL-Compiler bei der Generierung von Stubs und Skeletons diese Arbeit mit übernimmt, denn er besitzt durch das Parsen der IDL-Definitionen alle dafür notwendigen Informationen. Die Schnittstelle des IR erlaubt neben dem Lesen auch das Einbringen und Ändern von Typinformationen. IDL-Interface-Definitionen besitzen zur globalen eindeutigen Identifizierung eine Repository-ID. Die Repository-ID wird üblicherweise ebenfalls durch den IDL-Compiler festgelegt und in die statischen Stubs und Skeletons hineingeneriert und kann somit auch dem IR bereitgestellt werden. Die vom IDL-Compiler generierte Repository-ID für unser Bank-Interface hätte bsw. folgendes Aussehen: `IDL:finance/Bank:1.0`.

An dieser Stelle sei auch bemerkt, daß alle von der OMG spezifizierten Schnittstellen-Definitionen mit einem *Pragma* `prefix "omg.org"` versehen sind, das dem eigentlichen Paketnamen, resultierend aus der in IDL durch Modularisierung vorgenommenen Einteilung der Sichtbarkeitsbereiche, vorangestellt wird. Der Einsatz dieses Pragmas dient der Vermeidung von Namenskollisionen für Repository-IDs, ohne eine umständlich geschachtelte Modulhierarchie aufbauen zu müssen. Das Prefix-Pragma kann somit von einer Organsisation eingesetzt werden, um es einheitlich auf alle von ihr definierten Repository-IDs anzuwenden.

2.2.3. Dynamic Skeleton Interface

Als Gegenstück zum DII bietet das *Dynamic Skeleton Interface* (DSI) für die Serverseite eine Möglichkeit, eine Anfrage an eine Objekt-Implementierung auszuliefern, ohne die Schnittstellen zur Übersetzungszeit zu kennen, die ein Objekt implementiert. Die grundsätzliche Idee dahinter ist, eine einheitliche *Dynamic Implementation Routine* (DIR) zur Verfügung zu stellen, die von der serverseitigen ORB-Umgebung aufgerufen werden kann, um eine Anfrage auf die auszuführende Operation abzubilden. Die DIR besitzt eine einheitliche Signatur, die durch das jeweilige Language-Mapping festgelegt ist, um sie für alle denkbaren Objekt-Schnittstellen verwenden zu können.

Der DIR werden der Operationsname und alle Operationsparameter in einem `ServerRequest`-Objekt übergeben, vergleichbar mit dem `Request`-Objekt beim DII. Dieses `ServerRequest`-Objekt wird von der Objekt-Implementierung auch für die Rückgabe der Ergebnisse genutzt (`set_result`, `set_exception`). Weitere Informationen über das Zielobjekt und die zu implementierende Schnittstelle können über das IR und die ORB-Schnittstelle bzw. über den Objekt-Adapter (Abschnitt 2.3) bezogen werden.

2.2.4. Statischer vs. dynamischer Objekt-Aufruf

Statisch generierte Stubs und Skeletons sind für die Anwendungsentwicklung in einer statisch getypten Programmiersprache, wie Java oder C++, und bei Kenntnis der Schnittstellen zur Übersetzungszeit der einfachste und natürlichste Weg, um Operationen auf einer Objektreferenz aufzurufen und eine Anfrage an eine Objekt-Implementierung zu vermitteln. Ein Fernaufruf auf einem CORBA-Objekt ist damit aus Clientsicht nichts weiter als ein normaler Prozeduraufruf mit einer etwas längeren Laufzeit, verglichen mit einem lokalen Objektaufruf. Deshalb wird dieser Ansatz auch in den meisten Anwendungsfällen zum Einsatz kommen.

Ein weiterer Punkt, der für den Einsatz statisch generierter Stubs und Skeletons spricht, ist der, daß effizientere Schnittstellen zwischen Stubs/Skeletons und ORB genutzt werden können. Diese Schnittstellen sind i. allg. nicht durch die CORBA-Spezifikation festgelegt, sondern der ORB-Implementierung überlassen. Eine Ausnahme bildet hier das Language-Mapping für Java. Im Rahmen der *Java ORB Portability Architecture* werden diese Schnittstellen explizit festgelegt. Da Java-Klassen über das Netz geladen werden können, um sie in einem Applet laufen zu lassen, stellt das Java-Mapping höhere Anforderungen an die Portabilität der Stubs/Skeletons. Hier soll vermieden werden, daß zusätzlich zu den Anwendungsklassen die ORB-Runtime-Umgebung geladen werden muß. Java-Stubs/Skeletons sollten mit jeder CORBA-konformen ORB-Umgebung lauffähig sein (siehe auch Kapitel 5).

Trotzdem können auch DII/DSI-basierte Stubs/Skeletons vom IDL-Compiler statisch generiert werden. In diesem Fall braucht das IR natürlich nicht einbezogen werden, da die Objekt-Schnittstellen zur Übersetzungszeit bekannt sind. Üblicherweise werden aber effizientere Lösungen als das generische DII bzw. DSI für das *(Un)marshaling* innerhalb statischer Stubs/Skeletons eingesetzt. Für Java existiert bsw. ein stream-basierter Standard. Die Möglichkeiten der dynamischen Anfrageerzeugung bzw. -auslieferung werden hauptsächlich bei der Realisierung von generischen System-Level-Komponenten genutzt (z.B. *gateways*, *bridges*), die nicht jedesmal neu übersetzt werden können, wenn ein Interface der ORB-Umgebung hinzugefügt wird.

Für die Serverseite ist es transparent, ob eine empfangene Anfrage von einem statisch generierten, Typ-spezifischen Stub eingepackt oder dynamisch über das DII zusammengesetzt wurde. Gleiches gilt für die Clientseite. Ihr bleibt vollständig verborgen, in welcher Weise die Auslieferung der Anfrage an eine Objekt-Implementierung erfolgte.

2.2.5. Aufrufmodi

CORBA unterstützt mehrere Aufrufmodi für die Anfrage eines Client an ein Objekt. Nur zwei der nachfolgend angegebenen Modi sind momentan für statisch generierte Stubs verfügbar. Dafür ist in der IDL-Definition einer Operation (bzw. eines Attributes) das Schlüsselwort `oneway` vorgesehen. Es legt die Aufrufart für *alle* Aufrufe dieser Operation auf einer Objektreferenz fest. Fehlt dieses Schlüsselwort, wird standardmäßig der synchrone Aufrufmodus angenommen.

- *synchronous:* Identisch mit einem Prozeduraufruf, blockiert einen Client bis zum Erhalt der Antwort.
- *deferred synchronous:* Der Client wartet nicht auf das Ergebnis, sondern fährt mit seinem Prozeß fort. Zu einem späteren Zeitpunkt sammelt er die Antworten ein. Diese Variante kann momentan nur über das DII genutzt werden.
- *oneway:* Die Anfrage wird vom Client nach dem Prinzip *"best effort"* abgeschickt. Der Client fährt mit seinem Prozeß fort. Wann die Anfrage an einen Client geschickt wird und ob sie diesen auch erreicht, wird vom ORB nicht garantiert. Der Client bekommt keine Rückmeldung und darf demnach auch weder Ergebnisse noch Ausnahmen erwarten.

2.2.6. Grundlegende Vermittlung eines CORBA-Aufrufes

Nun möchte ich das Augenmerk auf die Serverseite lenken und die Darstellung der dort verwendeten CORBA-Technologien bei der Vermittlung einer serverseitig eingehenden Anfrage an die Objekt-Implementierung verfeinern. In der Abbildung 2.2 verbarg sich die Vermittlungsfunktionalität im Serverprozeß noch innerhalb der sehr allgemein gehaltenen ORB-Runtime-Komponente. Dieser Abschnitt dient zur Einführung zweier wesentlicher Begriffe, *Servant* und *Objekt-Adapter*. Damit ist eine erste Annäherung an das eigentliche Thema dieser Arbeit verbunden. Die Abbildung 2.3 stellt die verfeinerte Sicht auf den Auslieferungsprozeß einer Anfrage dar.

Für die in den bisherigen Ausführungen als Objekt-Implementierung bezeichnete Komponente wird in der CORBA-Terminologie auch der Begriff *Servant* verwendet. Dabei handelt es sich um eine Instanz eines programmiersprachlichen Konstruktes, das die Implementierung für ein oder mehrere CORBA-Objekte bereitstellt. In der objektorientierten Welt, und von dieser werde ich in Zukunft ausgehen, wäre das das Exemplar einer vom Anwendungsentwickler zur Verfügung gestellten Implementierungsklasse. Die Begriffe werden oftmals synonym verwendet, wobei aber bei einer Objekt-Implementierung die Implementierung der Operationen des IDL-Interfaces im Vordergrund steht und ein Servant diese Funktionalität aber noch erweitert. Für den Moment ist es ausreichend darauf hinzuweisen, daß ein Servant zusätzlich die Skeleton-Funktionalität bereitstellt, bsw. durch Erben von einer statisch generierten Skeleton-Klasse oder durch die Implementierung der DIR. Da Skeleton und Objekt-Implementierung aus der Sicht des Objekt-Adapters gewöhnlich eine Einheit bilden, wird in Zukunft meist nur noch von einem Servant die Rede sein.

Ein *Objekt-Adapter* (OA) fungiert innerhalb einer Serveranwendung als Verbindungsstück zwischen Servants und ORB. Er adaptiert das Programmiersprachenkonzept von Servants für den ORB, so daß dieser die Servants nicht zu kennen braucht. ORB und OA können damit auf einer abstrakteren Ebene kommunizieren. Diese Funktionalität errinnert an das Adapter-Muster aus [6]. Ein OA übernimmt einen Großteil der Funktionen, die für die Umsetzung der Abstraktion eines CORBA-Objektes benötigt werden. Eine wichtige Aufgabe eines OAs ist die Auslieferung der vom ORB empfangenen Anfrage an den verantwortlichen Servant, der die Inkarnation des CORBA-Objektes darstellt. Der OA bietet dem abstrakten CORBA-Objekt eine Art Ablaufumgebung für den Servant-Aufruf und übergibt das Ergebnis nach Fertigstellung wieder dem ORB für die Rücksendung an den Client. Neben der architektonischen Rolle als Vermittler zwischen ORB und Servants besitzt ein OA noch weitere Aufgaben, auf die ich im Abschnitt 2.3 zu sprechen komme.

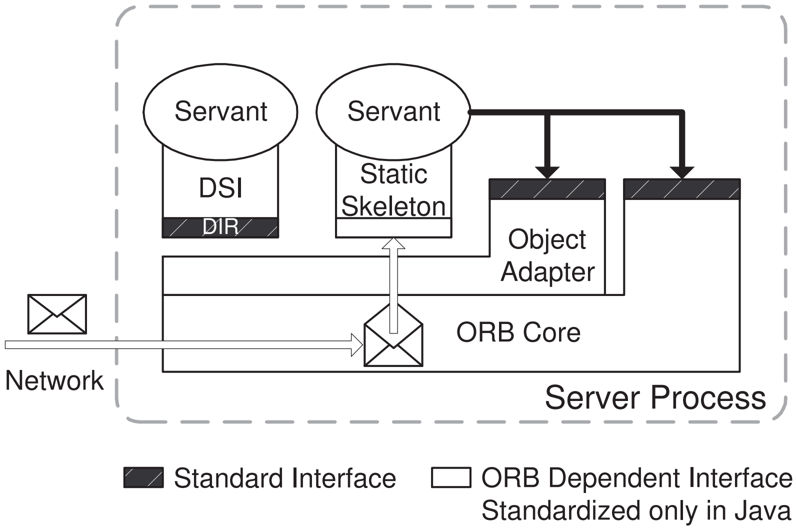


Abbildung 2.3.: Grundlegende Vermittlung eines CORBA-Aufrufes

2.2.7. Netzwerkprotokolle und Adressierung

Zur vollständigen Beschreibung des Anfrageflußes bei einem Objekt-Fernaufruf fehlt noch die Darstellung der Kommunikationsmechanismen zwischen clientseitiger und serverseitiger ORB-Umgebung. Vor dem Erscheinen der CORBA-Version 2.0 waren sowohl das zu verwendende Netzwerkprotokoll als auch die Adressierung eines Zielobjektes vollständig der ORB-Implementierung überlassen, was dazu führte, daß seine eigenen Verfahren entwickelte und in der Konsequenz keine Objekt-Aufrufe zwischen ORBs verschiedener Hersteller möglich waren. Diese Situation besserte sich mit der Verabschiedung des CORBA 2.0 Standards, der zwei Inter-ORB Protokolle festlegte, nach denen ORBs verschiedener Hersteller kommunizieren können.

Das *General Inter-ORB Protocol* (GIOP) definiert als ein abstraktes Protokoll, wie konkrete Protokolle die Kommunikation zwischen ORBs regeln. Es werden Annahmen über die darunterliegende Transportschicht getroffen, die sich exakt an dem verbindungsorientierten Internet-Protokoll TCP orientieren. GIOP legt acht verschiedene Nachrichtentypen fest. Zwei dieser Nachrichtentypen reichen aus, um den RPC-Mechanismus zu realisieren *(Request, Reply)*. Alle weiteren Typen dienen der Verbindungskontrolle oder zu Optimierungszwecken. Außerdem definiert GIOP eine *Common Data Representation* (CDR), also das *on-the-wire* Format der zu übertragenden Nachrichten und Parameter für jeden IDL-Typ.

Das *Internet Inter-ORB Protocol* (IIOP) ist eine konkrete Implementierung von

GIOP über TCP/IP. Jeder CORBA 2.0 konforme ORB muß IIOP implementieren. Darüber hinaus können auch weitere Implementierungen unterstützt werden, prinzipiell ist jede Kommunikation über ein verbindungsorientiertes Protokoll denkbar (z.B. über ATM). Da sich GIOP in seinen Annahmen an TCP orientiert, ist IIOP fast eine 1:1 Umsetzung von GIOP. Zusätzlich werden nur die protokollspezifischen Adreßinformationen zur Kodierung einer Objektreferenz festgelegt.

Um einen Aufruf auf einem entfernten Objekt durchführen zu können, muß eine Objektreferenz Informationen enthalten, wie das entfernte Objekt über ein Netzwerk zu erreichen ist. Außerdem muß eine Repräsentation einer solchen Objektreferenz auf dem Netzwerk gefunden werden, die es erlaubt, eine Objektreferenz als Parameter bzw. Ergebnis eines Aufrufes zu übertragen und nach dem Empfang und der Dekodierung wiederum Aufrufe darauf auszuführen. Das Standardformat dafür ist die *Interoperable Object Reference* (IOR). Sie wird von einer Serveranwendung generiert, um ein Server-Objekt zu adressieren. Eine IOR ist so flexibel aufgebaut, daß sie die protokollspezifischen Adreßinformationen für jedes erdenkliche Inter-ORB-Protokoll speichern kann. Darüber hinaus kann eine IOR innerhalb sogenannter *Profiles* mehrere Adreßinformationen auf einmal aufnehmen, bsw. für verschiedene Protokolle (Optimierung durch Auswahl des bevorzugten Protokolls) oder mit sich unterscheidenden Adressen (Ausfallsicherheit durch redundante Adressierung mehrerer Server-Objekte). Der IOR-Mechanismus ist auch für neu hinzukommende Profile-Formate gerüstet. Damit ein Client eine IOR verarbeiten kann, muß nur sichergestellt sein, daß sie ein ihm bekanntes Profile-Format enthält, ihm unbekannte Profiles werden einfach ignoriert. Die folgende Übersicht zeigt den Inhalt einer IOR in lesbarer Form.

```
TypeId         : IDL:finance/Bank:1.0
Profile
  Profile Id   : TAG_INTERNET_IOP
  IIOP Version : 1.1
  Host         : 127.0.0.1
  Port         : 33333
  Object Key   : 0x66 6f 6f 2f 66 61 63 74 6f 72 79 50 4f 41 ...
Profile
  ...
```

Ein IIOP-Profile enthält neben einer Profile-ID und der Protokoll-Versionsnummer die Host-Adresse und die TCP-Portnummer, unter der ein Server-Objekt erreichbar ist. Jede andere GIOP-Implementierung würde ein eigenes Profile mit den für dieses Protokoll relevanten Adreßinformationen definieren. Außerdem ist in dem Profile ein Objekt-Key enthalten. Dieser identifiziert das Server-Objekt innerhalb der serverseitigen ORB-Umgebung. Anhand des Objekt-Keys hat der Server-ORB die Möglichkeit, den passenden OA zu identifizieren und diesem die Anfrage auszuhändigen. Der OA ermittelt wiederum anhand einer im Objekt-Key enthaltenen Objekt-ID den verantwortlichen Servant. Auf diesem kann dann die Operation ausgeführt werden. Die Zusammensetzung eines Objekt-Keys ist nicht standardisiert, sondern bleibt der ORB-Implementierung überlassen. Für einen Client ist diese Byte-Folge völlig uninteressant, er schickt sie bei einem Aufruf uninterpretiert innerhalb einer Request-Message an den Server.

Außerdem ist in jeder IOR die Repository-ID des am meisten spezialisierten IDL-Interfaces *(most drived type)* enthalten, das das adressierte Objekt implementiert. Diese Information ist sehr wichtig, um auf der Clientseite die passende Stub-Klasse zur Erzeugung einer Objektreferenz ermitteln zu können (down-cast). Anhand der Repository-ID können weitere Information zur Objekt-Schnittstelle über das *Interface Repository* erfragt werden (z.B. für einen dynamisch zusammengestellten Aufruf). Die folgende Übersicht zeigt den Inhalt einer IOR in lesbarer Form.

Zur Übertrag einer IOR über das Netzwerk, wird die Datenstruktur in eine Zeichenkette transformiert. Diese String-Version kann jederzeit wieder in eine geparste Version einer IOR transformiert werden. Dadurch ist ein allgemeiner Mechanismus gegeben, um Objektreferenzen weiterzugeben. Ob sie in eine Datei geschrieben wird, die über ein verteiltes Dateisystem zugreifbar ist, oder ob sie in einer E-Mail verschickt wird; aus einer so übertragenen IOR kann ein Empfänger wieder eine Objektreferenz erstellen. Der ORB bietet für die Transformationen zwei Standard-Operationen an, eine `object_to_string()` Operation, der eine Objektreferenz übergeben wird und die einen String zurückliefert, und die `string_to_object()` Operation, die die Umkehrfunktion bereitstellt. Ein Beispiel für die String-Version einer IOR möchte ich an dieser Stelle einmal angeben, auch wenn es sich für den menschlichen Betrachter nur um eine "sinnlose" Aneinanderreihung von Hexadezimalzahlen handelt, der die Zeichenkette "`IOR:`" vorangestellt ist.


```
IOR:000000000000002849444c3a6f6d672e6f72672f436f734e616d696e672f4e61
6d696e67436f6e6d696e67436f6e746578743a312e3000000000003000000000000000
3c000100000000000a3132372e302e302e3100055f00000245374616e646172644e0
532f4e616d655365727665722d504f412f4e616d655365727665720000000000000
40000101000000000a3132372e302e302e3100055f000000245374616e646172644e
532f4e616d655365727665722d504f412f4e616d655365727665720000000000000
010000002c0000000000000001000000010000001c000000005010001000000001 00
01000100010109000000000105010001
```


Aus dieser Darstellung wird deutlich, daß auch diese Version einer Objektreferenz für den Client *opaque* ist. Es stellt für ihn einen unvertretbaren Aufwand dar, eine solche Zeichenkette selbst zusammenzustellen, mit der sich dann eine Objektreferenz erzeugen läßt. Der *opaque* Charakter einer IOR führt zu einem Problem für den Client, der nach dem Start des Clientprozesses den Zugriff auf eine initiale Objektreferenz benötigt, um durch einen Aufruf (z.B. auf einem Factory-Objekt) evtl. weitere Objektreferenzen zu erhalten. Damit stellt sich die Frage, ob es einen komfortableren Weg für den Erwerb initialer Objektreferenzen gibt als die externe Vermittlung einer IOR.

CORBA definiert zwei Vermittlungsdienste als *Object Services*, einen *Naming Service* [24] und einen *Trading Service* [20], die dieses Problem reduzieren. Sie treten selbst als Server-Objekt gegenüber einer Anwendung auf, die sich mit Objektreferenzen versorgen will. Vor allem der *Naming Service* ist ein häufig eingesetzter Dienst, der es einer Serveranwendung gestattet, die von ihr bereitgestellten Objekte unter einem "lesbaren" Namen zu registrieren, so daß ein Client über diesen logischen Bezeichner eine Objektreferenz ermitteln kann. Da diese Dienste selbst als CORBA-Objekte realisiert sind, ist das Problem damit nur verschoben worden. Eine Lösung stellt die ORB-Initialisierungsschnittstelle bereit. Häufig verwendete Dienste und Basis-Schnittstellen des ORBs können über die ORB-Funktion `resolve_initial_references` bezogen werden. Ihr wird ein standardisierter Bezeichner übergeben. Im Falle des Namensdienstes ist es der String "`NameService`".

Solange die ORB-Implementierung und die verwendeten Basisdienste aus einer Hand kommen, ist für die Anwendung das aufgeworfene Problem gelöst. Da die initiale Versorgung des ORBs mit diesen Bootstrapping-Objektreferenzen vor dem Erscheinen der CORBA Version 2.3 noch ein Implementierungsdetail war, konnte man gewöhnlich keinen ORB in der Art konfigurieren, daß er die Dienste eines anderen Herstellers über die ORB-Initialisierungsschnittstelle bereitstellt. Mit der CORBA Version 2.3 und der Spezifikation eines *Interoperable Naming Service* wurde dieses Problem durch die Einführung "lesbarer" IOR-Strings gelöst [24]. Die Syntax orientiert sich dabei an der eines *Uniform Resource Locators* (URL) für das WWW.

2.3. Objekt-Adapter

Ein Objekt-Adapter ist der Teil des ORBs, der die Verbindung zwischen der Implementierung einer IDL-Schnittstelle und dem ORB herstellt. In Abschnitt 2.2.6 ist eine wesentliche Aufgabe eines OA schon angesprochen worden: die Auslieferung einer Anfrage an den verantwortlichen Servant. Ein OA stellt die Ablaufumgebung für den Aufruf auf der Objekt-Implementierung bereit und nimmt die Zuteilung der benötigten System-Ressourcen (Prozesse, Threads) vor.

Neben seinem Anteil an der Anfragebearbeitung stellt ein OA auch die Laufzeitumgebung zur Verfügung, in der CORBA-Objekte erzeugt werden können, und weist diesen Objekten systemweit eindeutige Objektreferenzen zu. Damit ist ein OA am Lebenszyklus eines CORBA-Objektes beteiligt. Die Erzeugung einer Objektreferenz und deren Herausgabe an einen Client ist der Zeitpunkt, an dem ein CORBA-Objekt zu existieren beginnt.

Um Anfragen an einen Servant weiterleiten zu können, muß ein OA der Anwendung die Möglichkeit geben, den verantwortlichen Servant für ein Objekt zu registrieren. Die Erzeugung von Objekten und die Registrierung von Servants sind zwei verschiedene Dinge. Die Trennung zwischen einem Objekt und dessen Implementierung verlangt die Verwaltung der Assoziationen zwischen diesen Entities. Mit der Registrierung eines Servants für ein Objekt wird ein Objekt aktiviert. Man spricht auch von einer Inkarnation eines Objektes durch einen Servant. Eine Objektdeaktivierung wird durch die Trennung dieser Verbindung vollzogen.

Damit sind die grundlegenden Aufgaben eines OAs skizziert. Für die Entwicklung anspruchsvoller Serveranwendungen ist diese Funktionalität allein nicht ausreichend. Zur Versinnbildlichung sei ein kleines Szenario aufgeführt. Eine Anwendung besteht aus vielen tausend Objekten, deren Zustand in einer Datenbank gespeichert ist, und auf die ein Client, auch über mehrere Server-Instanzen hinweg, über die selbe Objektreferenz zugreifen möchte (*persistente* Objekte). Einige Objekte sind im selben Prozeß implementiert, andere befinden sich auf entfernten Rechnern. Objekte können auf andere Rechner migrieren, ohne daß ein Client etwas davon bemerkt. Auch eine solche Anwendung soll von einer ORB-Implementierung optimal unterstützt werden. Im folgenden möchte ich noch auf einige Konzepte zu sprechen kommen, die CORBA als Verteilungsplattform bietet und für den Entwurf eines OAs von Bedeutung sind.

2.3.1. Dynamisches Binden

Wenn eine Anfrage an ein Objekt von der Serverseite empfangen wird, versucht der Server-ORB in Zusammenarbeit mit einem OA den registrierten Servant zu ermitteln, auf dem der Aufruf durchgeführt werden kann. Falls kein Servant für dieses Objekt aktiviert wurde, kann dem Client nur eine Fehlermeldung in Form einer **OBJECT_NOT_EXIST** System-Exception gesendet werden. Um in einer Serveranwendung mit vielen tausend Objekten jede potentielle Anfrage beantworten zu können, müßten bei der Server-Initialisierung alle Objekte im voraus aktiviert werden. Es wäre jedoch wünschenswert, die Bereitstellung eines Servants erst bei Bedarf vornehmen zu können. Der OA sollte einen Mechanismus zur Verfügung stellen, der das dynamische Binden eines Servants an eine Objektreferenz ermöglicht (*late binding*).

2.3.2. Thread-Zuweisung

Die Bereitstellung von Threads, in denen Anfragen bearbeitet werden, wird vom OA vorgenommen. Es ist durchaus vorstellbar, für jede Anfrage einen eigenen Thread zu starten, so daß mehrere Anfragen innerhalb eines Servers nebenläufig ausgeführt werden können. Das Zulassen von Nebenläufigkeit innerhalb eines Servers vermindert das Risiko von Deadlocks, die durch Rückrufe des selben Servers während einer Ausführung einer Anfrage entstehen können. Auf der anderen Seite können Anwendungen, weil sie nicht thread-sicher programmiert wurden, diese Nebenläufigkeit vermeiden wollen. Der Anwendungsentwickler sollte auf das eingesetzte Verfahren Einfluß nehmen können. Die Frage ist, wie flexibel und wie komfortabel das Verfahren konfiguriert werden kann, ob der ORB mehrere Strategien zur Auswahl stellt oder ob die Anwendung für die Implementierung dieser Strategien sorgen muß.

2.3.3. Persistenz

Die Objekte einer Serveranwendung sollten auch nach dem Herunterfahren des Servers und einem Neustart wieder auf ihren alten Zustand zugreifen können, und ein Client, der eine Objektreferenz auf ein persistentes Objekt in den Händen hält, soll diese auch dann noch benutzen können. Die Abstraktion eines CORBA-Objektes verlangt, daß ein Client auf einer Objektreferenz jederzeit Aufrufe durchführen kann, auch wenn der Serverprozeß, in dem die Objekt-Implementierung bereitgestellt wird, nicht existiert. Zum Erreichen dieser Abstraktion müssen gleich mehrere Verfahren eingesetzt werden.

Erstens müssen Serveranwendungen bei Bedarf gestartet werden können. Zur Unterstützung dieses Konzeptes sieht CORBA eine serverseitige Datenbasis vor, das *Implementation Repository* (ImR). Darin können Informationen zur Lokalisierung und zum Starten von Serverprozessen für eine Menge von Server-Objekten gespeichert werden. Ein permanent erreichbarer Daemon-Prozeß auf dem Server-Rechner muß dafür sorgen, daß mit Hilfe des ImRs ein Serverprozeß gestartet wird, dem die Anfrage übergeben werden kann. Dieses hat natürlich Konsequenzen für den Inhalt einer Objektreferenz. Diese muß als Adressierungsinformation die Adresse des Activator-Daemons enthalten. Diese Indirektion ist nur für persistente Objekte nötig und kann bei Objekten, die nur innerhalb einer Server-Instanz gültig sind (*transiente* Objekte), weggelassen werden.

Zweitens müssen Objekte die Möglichkeit besitzen, ihren Zustand bei einer Deaktivierung zu speichern und bei einer folgenden Aktivierung wieder zu laden. Da dies auch für Objekte gelten soll, die erst als Folge einer Anfrage aktiviert werden, muß der OA in diesem Punkt mit der Anwendung kooperieren.

Drittens müssen Server in einer geordneten Weise heruntergefahren werden können. Während eines Server-Shutdowns müssen alle persistenten Objekte deaktiviert werden, damit diese ihrerseits ihren persistenten Zustand speichern können.

2.3.4. Aufrufumleitung

Die GIOP-Spezifikation definiert den **LOCATION_FORWARD** Reply-Status zur Umleitung einer Anfrage an ein anderes Objekt. Eine solche Reply-Message muß die neue Objektreferenz enthalten, an die der Client-Stub nach dem Empfang gebunden werden kann. Das Neu-Binden einer Objektreferenz an ein anderes Objekt geschieht für den Client vollständig transparent. Der Client-Stub wiederholt seine Anfrage mit den aktualisierten Daten, bevor der Kontrollfluß wieder an den Aufrufer zurückgeht.

Der Location-Forward-Mechanismus kann für verschiedene Zwecke eingesetzt werden. Ein Beispiel wäre das Neu-Binden einer Objektreferenz, wenn bei einer Anfrage an ein persistentes Objekt ein neuer Serverprozeß hochgefahren werden mußte und dieser in Zukunft die Anfragen entgegennehmen kann. Hierbei handelt es sich noch um eine ORB-interne Lösung. Interessanter ist die Frage, ob eine Anwendung diesen Mechanismus ebenfalls nutzen kann. Eine Objekt-Implementierung sollte die Möglichkeit haben, die Bearbeitung der Anfrage an ein anderes Objekt zu delegieren. Damit könnte man die "Migration" von Objekten unterstützen.

2.3.5. Lokale Objekte

Aufrufe auf einem lokalen CORBA-Objekt, d.h. Servant und Objektreferenz befinden sich im selben Adressraum, sollen sich in ihrer Semantik nicht von Aufrufen auf einem entfernten Objekt unterscheiden. Eine unterschiedliche Behandlung dieser Aufrufe wäre aus architektonischer Sicht gar nicht notwendig. Objektreferenzen, die an lokale Objekte gebunden sind, würden den lokalen Server adressieren und ebenfalls eine Socketverbindung (im Falle von IIOP), in diesem Fall aber zu einem lokalen Port, unterhalten (*loopback*). Eine Anfrage wäre ein ganz normaler IIOP-Request und die Lokalitätstransparenz wäre gegeben.

Nun ist es aus Performance-Gründen durchaus wünschenswert, einen solchen lokalen Aufruf gesondert zu behandeln. Die Verpackung der Anfrage in einem IIOP-Request *(Marshaling)* stellt einen unnötigen Overhead dar, da ein Aufruf innerhalb eines Prozesses auch als normaler Objekt-Aufruf auf der Objekt-Implementierung realisiert werden kann. Eine entsprechende Stub/Skeleton-Architektur, die ein direktes Binden einer Implementierung an eine Objektreferenz gestattet, wird von den meisten ORB-Implementierungen angeboten. Die interessante Frage ist aber nun, ob die Lokalitätstransparenz derartiger Aufrufe für alle Aspekte der Anfragebearbeitung gewährleitet ist. Beispiele potentiell kritischer Fragen wären, ob eine Thread-Policy in dem selben Maße angewendet wird, wie für konkurrierende entfernte Aufrufe auf diesem Objekt, ob ein OA die Auslieferung einer Anfrage noch beinflussen kann oder ob die erzielten Ergebnisse bei einem Auftreten eines Fehlerzustandes exakt die selben sind, wie bei einem Fernaufruf.

2.3.6. Standards

Mit der ersten CORBA-Version wurde ein einfacher OA eingeführt, der *Basic Object Adapter* (BOA). Diesem werde ich mich im nächsten Kapitel zuwenden. Der BOA steht für die erste Generation von CORBA-Serverprogrammen. Einige Zeit der Erfahrung mit dem BOA führte zu dem Ergebnis, daß eine zweite Spezifikationsrunde für die CORBA-Serverarchitektur eingeleitet wurde. In einem *Portability Enhancement RFP* [15] der OMG wurden 1995 die Probleme skizziert, die mit der BOA-Spezifikation bestanden. Heraus kam ein weiterer OA, der *Portable Object Adapter* (POA), der auf den Erfahrungen mit dem BOA beruht und diesen als Standard-OA für eine CORBA-Serveranwendung ersetzt. Der POA wurde mit der CORBA-Version 2.2 eingeführt. Die POA-basierten Serveranwendungen bilden somit die zweite Generation von CORBA-Serveranwendungen. Im Kapitel 4 werde ich ausführlich auf den POA zu sprechen kommen.

3. Basic Object Adapter

Dieses Kapitel gibt einen Überblick über den *Basic Object Adapter* (BOA), der bis zur CORBA-Version 2.1 der einzige von der OMG spezifizierte OA war. Der BOA sollte für die Mehrheit der Anwendungen genügen und neben weiteren OAs existieren, die dann für spezielle Anforderungen entwickelt werden. Die Überlegung der OMG war, kein OA würde für jede mögliche Anwendung die idealen Lösungen bieten können, so sollten für spezielle Anwendungsbereiche maßgeschneiderte (nicht-standardisierte) OAs eingesetzt werden. Die Entwicklung nahm aber einen anderen Verlauf. Jeder ORB-Hersteller hat zwar eine BOA-Implementierung bereitgestellt, die BOA-Funktionalität wurde aber an vielen Stellen erweitert, um ihn für alle möglichen Anwendungen passend zu machen, so daß im Endeffekt kein einziger spezialisierter OA hinzukam, sondern alle nichtstandardisierten Erweiterungen ebenfalls unter der Bezeichnung BOA liefen. Im folgenden werde ich die Basiskonzepte der BOA-Spezifikation beschreiben, um dann später auf die Unzulänglichkeiten eingehen zu können, die die Portabilität von BOA-basierten Serverprogrammen einschränken.

3.1. Funktionalität

Aus ORB-Sicht sorgt ein BOA dafür, daß eine Anfrage an ein Objekt vermittelt werden kann. Dazu besitzt der BOA die Fähigkeit, in Zusammenarbeit mit dem *Implementation Repository* (ImR), Prozesse zur Bearbeitung einer Anfrage zu starten, auf die Beendigung ihrer Initialisierung zu warten und diesen die Anfrage auszuhändigen. Der Inhalt des ImR ist nicht in CORBA spezifiziert, bleibt also der ORB-Implementierung überlassen. Ein Eintrag im ImR enthält bsw. den Pfadnamen zum ausführbaren Programm und einen Wert für die *Activation Policy*. Die CORBA-Spezifikation definiert vier verschiedene Werte für die *Activation Policy*, die bestimmt, wann ein neuer Server aktiviert werden muß, um eine Anfrage zu bearbeiten (siehe weiter unten).

Aus der Sicht des Anwendungsentwicklers ist der BOA die Programmkomponente, die es ihm ermöglicht, seine Objekt-Implementierungen beim ORB zu registrieren. Wenn Anfragen an ein Objekt eintreffen, startet er ggf. Serverprozesse zur Bearbeitung bzw. reicht die Anfragen an die registrierten Objekt-Implementierungen weiter. Die CORBA-Spezifikation definiert den BOA als ein Pseudo-Objekt, d.h. ein Entwickler kann über die ORB-Schnittstelle ein `BOA`-Objekt erhalten, wobei es sich aber um ein lokales Programmiersprachen-Objekt handelt, das Kernfunktionalität bereitstellt. Die Schnittstelle eines Pseudo-Objektes wird zwar in IDL definiert, ist aber zur Kennzeichnung mit dem Zusatz `PIDL` versehen.[1] Die Definition in Pseudo-IDL (PIDL) dient einzig der Spezifikation der Schnittstelle und stellt klar,

[1] Die ORB-Schnittstelle ist ebenfalls in PIDL definiert. Damit ist der ORB selbst ein Pseudo-Objekt.

daß es sich nicht um ein fernaufrufbares Objekt handelt. Die Abbildung von PIDL-Schnittstellen in die jeweilige Implementierungssprache wird im Language-Mapping explizit angegeben. Für die Abbildung der Operationen werden aber i. allg. die normalen Regeln des Language-Mappings angewendet.

```
module CORBA { //PIDL
    interface BOA {
        Object create(in ReferenceData id,
                      in InterfaceDef intf,
                      in ImplementationDef impl);
        void dispose(in Object obj);
        void impl_is_ready(in ImplementationDef impl);
        void deactivate_impl(in ImplementationDef impl);
        void obj_is_ready(in Object obj, in ImplementationDef impl);
        void deactivate_obj(in Object obj);
        void change_implementation(in Object obj,
                                   in ImplemenationDef impl);
        ReferenceData get_id(in Object obj);
        Principal get_principal(in Object obj, Environment env);
        void set_exception (
              in exception_type  major, // NO, USER, SYSTEM_EXCEPTION
              in string userid,         // exception type id
              in any param_LOOK_AT_SOURCE
       );
    };
};
```

Die Schnittstelle des BOAs enthält insgesamt nur 10 Operationen. Die ersten zwei Operationen sind elementar für den Lebenszyklus eines Objektes. Mit der `create()` Operation wird ein Objekt erzeugt. Damit ist ebenfalls die Erzeugung einer Objektreferenz verbunden, die nun auch exportiert werden kann. Zur Erzeugung einer Objektreferenz werden drei Komponenten benötigt, eine ID, die die Objekt-Implementierung innerhalb des Serverprozesses eindeutig identifiziert, die Interface-Definition innerhalb des *Interface Repositories*, und eine Beschreibung, wie der Serverprozess erreicht werden kann (z.B. Hostadresse, Portnummer). Die Zusammensetzung einer ID ist nicht spezifiziert, sie wird von der Serveranwendung festgelegt. Sie kann bsw. einen Datenbank-Schlüssel enthalten, der auf den persistenten Zustand eines Objektes verweist. Im ImR wird somit eine Assoziation zwischen der Objektreferenz und der Objekt-Implementierung eingerichtet. Mit der `dispose()` Operation kann diese Assoziation wieder entfernt werden, um eine Objektreferenz zu löschen, die nicht mehr benötigt wird. Die Objekt-Implementierung bleibt davon unberührt, sie muß separat abgeräumt werden. Üblicherweise werden diese beiden Operationen nicht vom Entwickler selbst aufgerufen, sondern vom IDL-Compiler in den Skeleton-Code generiert, so daß sie automatisch beim Erzeugen (Konstruktor) und Zerstören (Destruktor) eines Servants ausgeführt werden. Die Spezifikation schreibt die Endgüligkeit der Zerstörung eines Objektes vor. Es ist aber meist möglich, nach der Zerstörung ein neues Objekt mit der gleichen Objektreferenz zu erzeugen. Ein Client würde davon nichts merken, denn die Referenz, die er in der Hand hält, wäre damit an das neue Objekt gebunden. Um dies zu verhindern, vor allem über mehrere Server-Instanzen hinweg, müßte ein sehr großer Aufwand betrieben werden.

Mit weiteren vier Operationen des `BOA`-Interfaces lassen sich alle von einem Serverprozeß implementierten Objekte bzw. einzelne Objekte aktivieren bzw. deaktivieren. Die Aktivierung eines Objektes erfolgt üblicherweise nach der vollständigen Initialisierung der Objekt-Implementierung (`impl_is_ready`, `obj_is_ready`). Erst danach können Anfragen an das Objekt ausgeliefert werden. Vorher werden sie mit einer Fehlermeldung zurückgewiesen. Hingegen werden Anfragen an inaktive Objekte vom BOA solange zurückgehalten, bis eine erneute Aktivierung erfolgt. Aktivierungen und Deaktivierungen können abwechselnd und mehrmals hintereinander erfolgen, ohne daß ein Client etwas davon bemerkt. Eine Deaktivierung kann zum Schutz eines Objektes eingesetzt werden, um ungestört vor zwischenzeitlichen Aufrufen bestimmte Verwaltungsaufgaben erledigen zu können (`deactivate_impl`, `deactivate_obj`). Es gibt auch die Möglichkeit, eine Objekt-Implementierung für ein bestimmtes Objekt auszutauschen (`change_implementation`). Eine zwischenzeitliche Deaktivierung des Objektes ist in diesem Fall angebracht.

Auf die restlichen Operationen des `BOA`-Interfaces möchte ich nur kurz eingehen. Die Operation `get_id` gestattet den Zugriff auf die serverinterne ID eines Objektes (Referenz-Daten). Die `get_principal()` Operation wird genutzt, um die Identität des Clients zu ermitteln, dessen Anfrage die Aktivierung eines Objektes bewirkt hat. In CORBA gibt es die Möglichkeit, Kontext-Informationen mit einem Objekt-Aufruf zu versenden. Diese können genutzt werden, um den Client eines Aufrufes auf der Serverseite zu identifizieren. Die Informationen werden vom ORB in einem `Principal`-Objekt bereitgestellt. Höher angesiedelte CORBA-Dienste nutzen diesen Mechanismus zur Übertragung kontextspezifischer Informationen (z.B. ein *Sercurity Service* zur Realisierung von Zugriffskontrolle).[2] Die `set_exception()` Operation wird von der Serveranwendung verwendet, um das Auftreten eines Fehlers anzuzeigen.

Damit ist die `BOA`-Schnittstelle im wesentlichen beschrieben. Ich möchte nun zu der Frage zurückkehren, welche Möglichkeiten ein BOA hat, um bei Bedarf Serverprozesse zu starten. Der Wert für die schon erwähnte *Activation Policy* einer Serveranwendung legt fest, für welche Anfrage ein neuer Serverprozeß gestartet werden soll, der dann die Bearbeitung der Anfrage übernimmt, oder ob der Serverprozeß von einer externen Instanz zu starten ist. Der verwendete Aktivierungsmodus bestimmt auch, welche Objekt-(De)aktivierungen der BOA von der Server-Implementierung erwarten kann.

Shared Server Mehrere Objekte, mit möglicherweise unterschiedlichen Schnittstellen, können von einem *Shared Server* bereitgestellt werden. Wenn noch kein Serverprozeß aktiv ist, wird er vom BOA gestartet. Die `impl_is_ready()` Operation benachrichtigt den BOA, daß alle Objekte initialisiert sind und Anfragen entgegennehmen können. Die Operation `obj_is_ready` muß beim Hochfahren nicht für jedes Objekt einzeln aufgerufen werden. Nachfolgend können einzelne (De)aktivierungen

[2] Die Principal-Schnittstelle ist mit dem Erscheinen der CORBA-Version 2.2 veraltet *(deprecated)*. *Object Services*, die Kontext-Informationen verwenden, sollen einen flexibleren Mechanismus zur Übertragung und zum Zugriff auf diese Informationen nutzen. Die Standardisierung entsprechender ORB-APIs steht noch aus (portable *Interceptors*).

auf Objektebene erfolgen. Alle Anfragen an Objekte dieses Servers werden dem BOA ausgehändigt, der sie seinerseits an die Objekt-Implementierungen weiterreicht.

Unshared Server Von einem *Unshared Server* wird genau ein Objekt bereitgestellt. Nach der Initialisierung der Objekt-Implementierung ruft die Server-Implementierung `obj_is_ready()` auf. Bei einer Anfrage an ein weiteres noch nicht existierendes bzw. nicht aktiviertes Objekt mit gleicher Schnittstelle wird eine neue Server-Instanz gestartet. Die Deaktivierung eines Objektes zieht also bei einer nachfolgend eintreffende Anfrage den Start eines neuen Servers nach sich.

Server per Method Für jede eintreffende Anfrage wird ein neuer Serverprozeß gestartet. Im CORBA-Standard sind für diesen Fall keine Registrierungen beim BOA vorgeschrieben. Einige ORB-Implementierungen verlangen aber die Benachrichtigung des BOAs durch `impl_is_ready()`, damit die Operation aufgerufen werden kann.

Persistent Server Das Starten eines *Persistent Servers* wird nicht durch den BOA, sondern von einer externen Instanz übernommen. Typischerweise wird ein solcher Serverprozeß vom Anwender gestartet. Ansonsten gleicht dieser Servertyp dem *Shared Server.* Falls Anfragen eintreffen, für die kein Serverprozeß gestartet ist, wird ein Fehler an den Client gesendet.

Die BOA-Funktionalität ist in zwei wesentliche Bereiche aufgeteilt. Bei einer typische BOA-Implementierung lassen sich somit auch zwei voneinander getrennte Teile ausmachen. Ein Teil ist in einem ORB-Daemon-Prozeß enthalten, der die Anfragen vom ORB über eine ORB-interne Schnittstelle entgegennimmt, das ImR konsultiert und verwaltet, ggf. neue Serverprozesse startet und die Anfrage an den Serverprozeß weiterleitet. Die vom BOA erzeugten Objektreferenzen enthalten somit auch die Adreßinformationen des Daemon-Prozesses als Endpunkt-Information.

Der zweite Teil der BOA-Implementierung steckt in jedem Serverprozeß. Dieser nimmt die Anfragen für die registrierten Objekte entgegen und führt die Aufrufe auf dem Skeleton aus. Die Schnittstelle, die der IDL-Compiler für die Skeletons generiert, bzw. die gesamte Skeleton-Architektur bleibt der ORB-Implementierung überlassen. Gegenüber der Anwendung tritt der BOA als Pseudo-Objekt auf und stellt die standardisierte Schnittstelle für die Verwaltung der Objekte bereit. Beide Teile des BOAs müssen zur Aktualisierung der Informationen im ImR und zur Erzeugung von Objektreferenzen kooperieren.

3.2. Evaluation

Die BOA-Spezifikation weist viele Mängel auf. Allgemein kann formuliert werden, daß wichtige Funktionen unterspezifiziert bzw. gar nicht erst vorhanden sind. Die Spezifikationslücken mußten durch die ORB-Hersteller ausgefüllt werden, was zu herstellerspezifischen Lösungen führte und die Portabilität von BOA-basierten Serveranwendungen zwischen ORB-Umgebungen verschiedener Hersteller verhinderte. Die Probleme lassen sich grob in zwei Kategorien aufteilen. Die erste Kategorie umfaßt die Probleme, die Folge der Unterspezifizierung grundlegender BOA-Funktionen sind. Die ORB-Hersteller mußten mit eigenen Lösungen aushelfen, um eine BOA-Implementierung überhaupt realisieren zu können. Ich werde nachfolgend einige wesentliche Mängel aufführen, andere verlangen einen tieferen Einstieg in die ORB-Implementierung [15]. In die zweite Kategorie fallen die wünschenswerten Erweiterungen, die die ORB-Hersteller ebenfalls veranlaßt haben, proprietäre Lösungen hinzuzufügen.

Spezifikationsmängel in der Basisfunktionalität

- Die Skeleton-Architektur ist nicht spezifiziert. Es gibt keine Namenskonventionen für die generierten Skeletons, was es unmöglich macht, Skeletons mit Objekt-Implementierungen in einer standardisierten Weise zu assoziieren.

- Die Schnittstellen zur Erzeugung und die Zusammensetzung eines `ImplementationDef`-Objektes sind unspezifiziert. Damit ist eine portable Erzeugung von Objekten unmöglich (`create`). Gleiches gilt für die Registrierung der Objekt-Implementierungen beim BOA, damit dieser Aufrufe weiterleiten kann. Eine einheitliche Registrierungsschnittstelle ist nicht vorhanden.

- Für eine Anwendung gibt es keinen portablen Weg, um die Objektreferenz des Zielobjektes zu ermitteln, das die Aktivierung eines Servers bewirkt hat, um auf dieser `get_implementation()` aufzurufen (notwendig zur Benachrichtigung des BOAs, daß eine Implementierung bereit ist, Anfragen entgegen zu nehmen).

- Bei einem *Unshared Server* kann es zu *race conditions* beim Schreiben und Lesen eines persistenten Objekt-Zustandes kommen, da schon der Aufruf von `deactivate_impl()` den Start eines neuen Servers bewirkt, wenn eine erneute Anfrage an das Objekt eintrifft. Dieser Server liest eventuell einen noch nicht vollständig geschriebenen Zustand.

- Es gibt keine portable Möglichkeit, um einen geregelten Server-Shutdown zu initiieren. Vor einer Terminierung des Serverprozesses sollte es einen ordentlichen Weg geben, bei dem alle laufenden Anfragen beendet und neue Anfragen in dieser Zeit gepuffert werden, um dann alle Objekte zu deaktivieren, die ihrerseits ihren persistenten Zustand speichern können. Dies erfordert eine Kooperation zwischen dem BOA und den Objekt-Implementierungen.

Wünschenswerte Erweiterungen

- Obwohl die BOA-Spezifikation eine klare Unterscheidung zwischen Objekt- und Server-Aktivierung macht, stellt sie die Server-Aktivierung in den Vordergrund und vernachlässigt die Aktivierung einzelner Objekte. Damit ist nicht klar definiert, wie dynamisches Binden implementiert werden kann, wie also Objekte nach Bedarf aktiviert werden können. Es fehlt eine Instanz mit einem standardisierten Interface, das vom Entwickler implementiert und beim BOA registriert wird, um Objekt-Implementierungen dynamisch bereitzustellen.

- Es gibt auch keine Möglichkeit, eine ganze Gruppe von Objekten gemeinsam zu aktivieren, wenn auf ein einzelnes Objekt aus dieser Gruppe zugegriffen wird. Die Speicherung des persistenten Zustand für jedes einzelne Objekt in einer Datenbank ist oft zu feinkörnig. In diesen Fällen wird der Zustand für meherer Objekte zusammen gespeichert.

- Die BOA-Spezifikation trifft keine Aussagen darüber, wie auf der Serverseite die Zuweisung von Thread-Ressourcen konfiguriert werden kann. CORBA-Implementierungen, vor allem auf der Serverseite, sind inhärent *multi-threaded*. Bei nicht thread-sicheren Anwendungen muß der Entwickler selbst für die Synchronisation sorgen. Eine feinkörnige Festlegung durch eine ORB-Policy ist nicht möglich.

- Eine Objekt-Implementierung sollte die Umleitung einer Anfrage an ein anderes Objekt initiieren können. Die BOA-Spezifikation stellt aber kein standardisiertes API bereit, so daß eine Objekt-Implementierung den Location-Forward-Mechanismus nicht portabel nutzen kann.

- Die BOA-Spezifikation ist sehr schwach in den Fragen der Umsetzung von Lokalitätstransparenz. Anfragen an ein lokales CORBA-Objekt werden üblicherweise als normale Objekt-Aufrufe auf der Objekt-Implementierung realisiert. Die interessante Frage ist aber, ob die Transparenz auch für alle Aspekte der Anfragebearbeitung gewährleistet ist (bsw. wenn ein Objekt deaktiviert wurde). Die ORB-Hersteller unterstützen gewöhnlich die effiziente Ausführung dieser Aufrufe durch entsprechende Stub/Skeleton-Architekturen, die Lokalitätstransparenz ist aber nicht in vollem Umfang gewährleistet.

Fazit

Eine portable Programmierung von BOA-basierten Serveranwendung ist unmöglich, da sich die existierenden ORB-Implementierungen zu sehr voneinander unterscheiden. Selbst wenn man davon ausgeht, daß eine Anwendung auf die erweiterte Funktionalität aus der zweiten Kategorie verzichtet, muß sie sich zumindest auf die proprietären Lösungen für die Probleme der ersten Kategorie verlassen. Hinzu kommt, daß in den ORB-Implementierungen der BOA-Generation die Tendenz zu beobachten war, die Benutzung des BOAs so transparent wie möglich für den Entwickler zu gestalten. Die Erzeugung von Objektreferenzen und die Registrierung der Objekt-Implementierungen wurde vollständig innerhalb des Skeleton-Codes erledigt. Zugunsten dieser Transparenz und auf Kosten der Flexibilität verzichtete man schon mal auf einige Funktionen des BOAs. Oftmals wurden nicht alle Aktivierungsmodi unterstützt. JacORB hat bsw. in der ersten Generation auf ein ImR verzichtet und ausschließlich den *Persistent Server* bereitgestellt.

Die vielen existierenden proprietären Erweiterungen des Standards zeigen, daß die Anwendungsentwickler (oder sind es die ORB-Entwickler) höhere Anforderungen an einen Standard-OA stellen, als die, die der BOA einlösen konnte. Nun ist der POA an der Reihe, diesen Anforderungen gerecht zu werden.

4. Portable Object Adapter

Nachdem die Probleme mit dem BOA erkannt wurden, hatte man nun die Möglichkeit, die Spezifikationsmängel des BOAs zu beheben oder einen komplett eigenständigen OA zu spezifizieren. Man hat sich letztendlich für die zweite Variante entschieden, einen zweiten Versuch zu starten, in den die Erfahrungen aus der BOA-Generation einfließen und der dem Wunsch der Hersteller entgegenkommt, einen "alleskönnenden" Universal-Adapter zur Verfügung zu haben, doch mit dem Unterschied, daß er nun portablen Server-Code erlaubt. Dieses ambitionierte Unterfangen bekam den Namen "Portable Object Adapter" und wurde in der *ORB Portability Joint Submission* vorgestellt [19]. In die CORBA-Version 2.2, die im Februar 1998 erschien, ging die Spezifikation als Standard ein und ersetzt den BOA als Standard-Adapter. Der BOA ist von nun an als eine herstellerspezifische Erweiterung zu betrachten, die von einem Hersteller bereitgestellt werden kann, um existierende Anwendungen zu unterstützen. Eine Referenz-Implementierung eines POAs, wie es normalerweise von der OMG gefordert ist, gab es zu diesem Zeitpunkt nicht.

In diesem Kapitel möchte ich den POA im Detail beschreiben. Die Beschreibung geht an vielen Stellen über die der Spezifikation hinaus. Es soll gerade bei den Fragen tiefer eingestiegen werden, die auf dem abstrakten Niveau der Spezifikation nur knapp beschrieben sind oder erst mit der Einbeziehung des jeweiligen Language-Mappings verständlich werden. Demzufolge werde ich an diesen Stellen das IDL/Java-Mapping heranziehen bzw. auf die entsprechenden Abschnitte des folgenden Kapitels verweisen. Diese beiden Kapitel sollen zusammen die Grundlagen für das Verständnis der Probleme bei der Implementierung eines POAs schaffen. Außerdem werde ich an gegebener Stelle die Bezüge zur Anwendungsentwicklung herstellen, so daß die Einsatzmöglichkeiten des POAs deutlich werden. In den Teilen der Beschreibung, die unabhängig vom jeweiligen Language-Mapping gelten, wird zumindest im Sprachgebrauch eine objektorientierte Begrifflichkeit benutzt.

4.1. Grundlagen

Die 64-seitige POA-Spezifikation [27] beschreibt einen vollständigen Objekt-Adapter mit einer Fülle von Eigenschaften und Fähigkeiten die es dem Entwickler erlauben, effiziente und erweiterbare Serveranwendungen zu schreiben. Er kann in vielfältiger Weise auf die verwendeten Systemressourcen Einfluß nehmen, die für die Implementierung von CORBA-Objekten benötigt werden. Dies ermöglicht ihm im Rahmen der gegebenen Ressourcen, den bestmöglichen Service für einen Client zur Verfügung zu stellen. Die Kenntnis der verschiedenen POA-Fähigkeiten und der Beziehungen zwischen ihnen, ihrer Vorteile und Nachteile für einen Anwendungsfall, ist bedeutend für die Entwicklung effizienter Serveranwendungen.

4.1.1. Objekt- und Servant-Lebenszyklus

Der POA ist am Lebenszyklus eines CORBA-Objektes beteiligt. Die Erzeugung einer Objektreferenz durch einen POA und deren Herausgabe an einen Client ist der Zeitpunkt, an dem ein CORBA-Objekt beginnt zu existieren. Nach seiner Erzeugung wechselt es zwischen den Zuständen aktiv und inaktiv. Während ein CORBA-Objekt aktiviert ist, können auf ihm Aufrufe durchgeführt werden. Die Abstraktion eines "aktiven CORBA-Objektes" setzt neben der Veröffentlichung einer Objektreferenz auch die Inkarnation des CORBA-Objektes durch einen Servant voraus, an den die Objektaufrufe vermittelt werden können. Einer Objektaktivierung geht also in jedem Fall eine Erzeugung eines Servants voraus. Dabei ist der Lebenszyklus eines Servants vollständig unabhängig vom Lebenszyklus eines CORBA-Objektes. Ein CORBA-Objekt kann zu einem Zeitpunkt nur durch einen einzelnen Servant inkarniert sein, jedoch über seinen kompletten Lebenszyklus hinweg durch viele verschiedene Servant-Instanzen. Die Trennung dieser Lebenszyklen wird auch in der Begrifflichkeit deutlich. Die Begriffe aktiv und inaktiv bzw. Aktivierung und Deaktivierung werden vorzugsweise im Zusammenhang mit dem Objekt-Lebenszyklus gebraucht. Im Gegensatz dazu wird beim Servant-Lebenszyklus von einer Inkarnation oder von einer Etherealization gesprochen. Die Inkarnation eines CORBA-Objektes durch einen Servant wird oftmals abgekürzt als Servant-Inkarnation bezeichnet. Diese Bezeichnung stellt die Servant-Sicht innerhalb des Objektaktivierungsprozesses heraus. Mit ihr wird aber keine Aussage darüber getroffen, ob für die Objektaktivierung eine schon vorhandene oder eine neu erzeugte Servant-Instanz verwendet wird. Bei einer Objektdeaktivierung wird die Verbindung zwischen einer Objektreferenz und einem Servant unterbrochen. Sie ist ebenso mit einer Servant-Etherealization verbunden. Die Servant-Etherealization bezeichnet die Beendigung einer Servant-Objektreferenz-Beziehung aus Servant-Sicht und geht nicht zwangsläufig mit einer Zerstörung des Servants einher, denn ein Servant kann zur gleichen Zeit für mehrere Objektreferenzen zuständig sein. Ein CORBA-Objekt hört auf zu existieren, wenn eine Serveranwendung bzw. der Anwender entscheidet, den Lebenszyklus eines CORBA-Objektes zu beenden. Die Diskussion zum Objekt-Lebenszyklus wird am Schluß dieses Kapitels noch einmal aufgegriffen, wenn die Funktionalität des POAs einbezogen werden kann.

4.1.2. Modell-Komponenten

Servant Ein Servant ist ein vom Anwendungsentwickler zur Verfügung gestelltes Exemplar einer Implementierungsklasse in einer konkreten Programmiersprache. Er stellt die Implementierung für ein oder mehrere abstrakte CORBA-Objekte bereit. Aufrufe auf einer Objektreferenz werden vom ORB über einen POA an den verantwortlichen Servant vermittelt.

Objektreferenz Bei einer Objektreferenz handelt es sich um ein Exemplar einer vom IDL-Compiler generierten Stub-Klasse. Sie kapselt die Verbindungsinformationen zur serverseitigen ORB-Umgebung und den Objekt-Key eines abstrakten Objektes. Aufrufe auf einer Objektreferenz werden vom ORB an das abstrakte CORBA-Objekt vermittelt. Der POA ist verantwortlich für die Erzeugung von Objektreferenzen. Eine Objektreferenz kann jeweils durch eine ORB-Funktion in einen String

(IOR) und wieder zurück konvertiert werden. Objektreferenzen werden von einer Serveranwendung an einen Client exportiert.

Objekt-Key Der Objekt-Key enthält die Identität eines POAs als vollständig qualifizierten POA-Namen und die Objekt-ID eines abstrakten CORBA-Objektes innerhalb einer Sequenz von Bytes.

Objekt-ID Eine Objekt-ID identifiziert ein abstraktes CORBA-Objekt innerhalb eines POAs eindeutig. Objekt-IDs werden entweder vom Anwendungsentwickler zur Verfügung gestellt oder vom POA generiert. Der POA verwaltet eine Objekt-ID als eine uninterpretierte Sequenz von Bytes und benutzt sie zur Erzeugung von Objektreferenzen und zur Ermittlung des verantwortlichen Servants für einen Objektaufruf.

POA Ein POA ist die Programmkomponente, die die Auslieferung einer Anfrage an ein Servant innerhalb einer Serveranwendung realisiert und den Entwickler bei der Erzeugung, Verwaltung und Zerstörung von CORBA-Objekten unterstützt. Jeder POA verwaltet einen Namensraum von Objekt-IDs für die eigenen CORBA-Objekte und einen Namensraum für weitere geschachtelte Kind-POAs. Geschachtelte POAs bilden somit einen hierarchischen Namensraum für CORBA-Objekte innerhalb einer Serveranwendung.

Policy Eine Policy ist ein Objekt, das mit einem POA assoziiert wird und eine Eigenschaft des POAs und damit für die von ihm verwalteten CORBA-Objekte festlegt. Durch die Auswahl geeigneter Policy-Werte bei der Konfiguration eines POAs kann der Anwendungsentwickler den POA an die speziellen Erfordernisse von Objekt-Implementierungen anpassen. Anzahl und Art der Policies, ihre Werte und deren Konsequenzen auf das Verhalten eines POAs sind durch die Spezifikation vorgegeben. Das Spektrum der mittels Policies konfigurierbaren POA-Eigenschaften reicht von der Festlegung einzelner Optionen, die den Ablauf der Anfragebearbeitung beeinflussen, bis hin zu verschiedenen Optionen, die die Verwaltung der CORBA-Objekte durch einen POA betreffen.

POA-Manager Ein POA-Manager kapselt den Zustand eines POAs. Es können mehrere POAs einem POA-Manager zugeordnet werden. Mit einem POA-Manager läßt sich der Strom von Anfragen steuern, der die Objekt-Implementierungen erreicht. Es kann bestimmt werden, ob die mit einem POA-Manager assoziierten POAs alle eintreffenden Anfragen bearbeiten, puffern oder zurückweisen sollen. Außerdem können POAs über ihren POA-Manager deaktiviert werden.

Servant-Manager Ein Servant-Manager ist ein vom Anwendungsentwickler zur Verfügung gestelltes und mit einem POA assoziiertes lokales CORBA-Objekt, das den POA bei der Verwaltung von Objekt-Implementierungen und der Vermittlung von Anfragen unterstützen kann. Der POA ruft Operationen auf einem Servant-Manager auf, um bei Bedarf, also im Falle einer Anfrage, den aktuellen Servant zu ermitteln bzw. zu erzeugen oder bei einer Objektdeaktivierung den Servant-Manager zu informieren. Zur Identifizierung eines Servants wird dem Servant-Manager die

Objekt-ID übergeben. Der Servant-Manager verwaltet Objekt-ID-zu-Servant-Beziehungen und entscheidet bei einem Aufruf, ob ein CORBA-Objekt existiert oder nicht. Es gibt zwei verschiedene Arten eines Servant-Managers, einen Servant-Activator bzw. einen Servant-Locator. Die zum Einsatz kommende Variante hängt von den Policies ab, mit denen ein POA konfiguriert wurde.

Adapter-Activator Ein Adapter-Activator ist ein vom Anwendungsentwickler zur Verfügung gestelltes und mit einem POA assoziiertes lokales CORBA-Objekt, das für einen POA bei Bedarf einen Kind-POA erzeugen kann, wenn der für eine Anfrage zuständige POA noch nicht existiert. Ein Adapter-Activator ist für die Erzeugung eines POAs unter dem gewünschten Namen und für dessen Initialisierung verantwortlich.

4.1.3. Modell-Architektur

Ein wichtiges Merkmal der POA-Spezifikation ist, daß es innerhalb einer serverseitigen ORB-Umgebung mehrere POA-Instanzen geben kann. Eine spezielle, immer vorhandene POA-Instanz wird als Root-POA bezeichnet und vom ORB zur Verfügung gestellt. Sie kann vom Anwendungsentwickler über die ORB-Initialisierungsschnittstelle ermittelt werden. Alle weiteren POA-Instanzen werden als Kind-POAs durch den Aufruf einer Erzeugungsoperation auf einem Eltern-POA erzeugt. Auf diese Weise läßt sich eine Baumstruktur von mehreren POA-Instanzen innerhalb einer Anwendung erstellen. Die Erzeugung mehrerer POA-Instanzen erlaubt dem Anwendungsentwickler die Aufteilung des Namensraums für CORBA-Objekte, da Objekt-IDs relativ zu einem POA interpretiert werden. Zum einen gibt es die Möglichkeit, alle benötigten Object-Adapter während des Server-Initialisierungsprozesses zu erzeugen, zum anderen kann dies aber auch einem Adapter-Activator überlassen werden. Ein Adapter-Activator wird mit einem POA assoziiert und kann die Erzeugung und Initialisierung eines Kind-POAs bei Bedarf übernehmen. Die Assoziation eines POAs mit einem Adapter-Activator ist optional. Der Root-POA besitzt standardmäßig keinen Adapter-Activator. Jede POA-Instanz ist einem POA-Manager zugeordnet. Dadurch, daß mehrere POAs mit einem einzelnen POA-Manager verbunden werden können, lassen diese sich zu einer Gruppe zusammenfassen und gemeinsam über die Schnittstelle des POA-Managers administrieren.

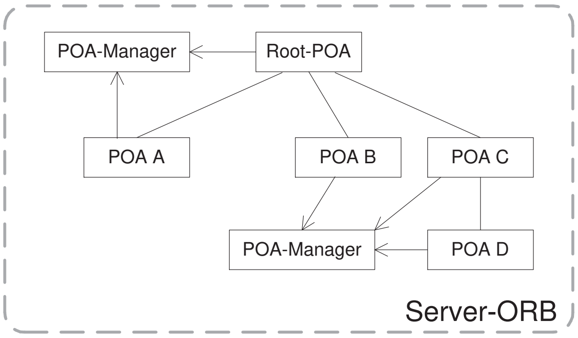


Abbildung 4.1.: POA-Hierarchie

Wird ein POA geeignet konfiguriert, so übernimmt er die Verwaltung der ihm bekannten Objekt-ID-zu-Servant-Beziehungen und speichert sie in einer *Active Object Map* (AOM). Durch einen Eintrag eines Objekt-ID-Servant-Paares in die AOM wird ein Objekt aktiviert. In der POA-Spezifikation wird also von einem "aktiven Objekt" gesprochen, wenn die AOM einen Eintrag enthält, der diese Objekt-ID mit einem Servant assoziiert. Gleiches gilt für die Bezeichnung eines "aktiven Servants". Ein Servant ist dann aktiv, wenn er durch mindestens eine Objekt-ID in der AOM referenziert wird. Objekt-IDs müssen innerhalb eines POAs eindeutig sein und dürfen somit nur einfach in der AOM auftreten.

Ein POA kann alternativ oder zusätzlich zur AOM mit einem Servant-Manager konfiguriert werden, um den aktuellen Servant für die in einer Anfrage enthaltene Objekt-ID zu ermitteln. Ein Servant-Manager verwaltet selbst die zu einer Objekt-ID zugehörigen Servants und erzeugt im Bedarfsfall einen neuen Servant, um diesen dem POA für die Durchführung einer Anfrage zur Verfügung zu stellen. Wird vom POA zusätzlich zum Servant-Manager eine AOM geführt, also eine eigene Servant-Verwaltung betrieben, so wird der vom Servant-Manager zurückgelieferte Servant in die AOM eingetragen. In diesem Fall spricht man von einer Servant-Inkarnation mit anschließender Objektaktivierung. Damit ist die Objekt-ID auch dem POA bekannt und zukünftige Anfragen kann er direkt an den registrierten Servant vermitteln. Falls keine AOM existriert, wird der vom Servant-Manager ermittelte Servant nur für die Durchführung dieser einen Anfrage verwendet und für alle weiteren Anfragen muß der Aufruf auf dem Servant-Manager wiederholt werden.

Anstelle eines Servant-Managers ist auch die Verwendung eines Default-Servants möglich. Für den Fall, daß ein POA keine AOM benutzt bzw. die in einer Anfrage enthaltende Objekt-ID des Zielobjektes nicht in der AOM verzeichnet ist, aber ein Default-Servant vorhanden ist, wird dieser für die Durchführung der Anfrage verwendet. Sind weder ein Servant-Manager noch ein Default-Servant bei einem POA registriert, kommt immer eine AOM zum Einsatz. In diesem Fall müssen Servants innerhalb der Anwendung dem POA bekannt gemacht werden. Die Abbildung 4.2 zeigt an einem konkreten Beispiel die Beziehungen zwischen den Komponenten in einer Serveranwendung mit drei POAs.

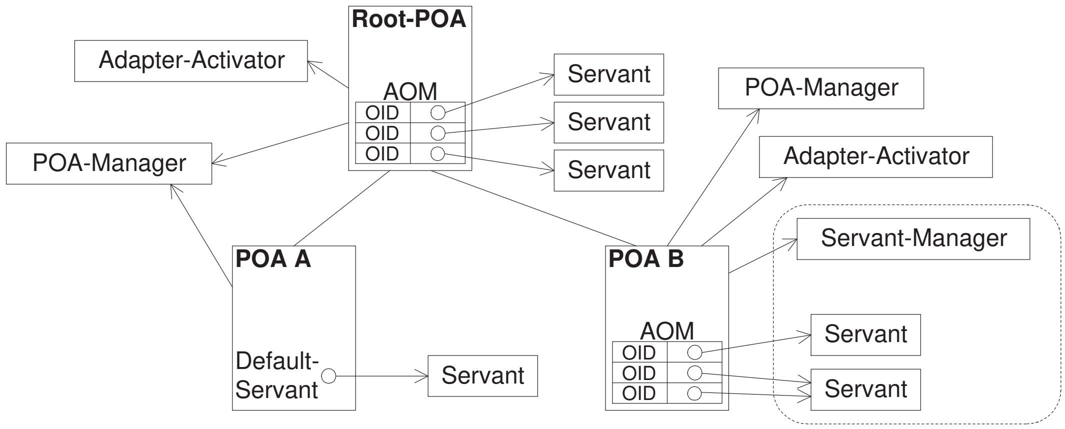


Abbildung 4.2.: POA-Architektur

Persistenz

Weder der ORB noch der POA verwalten einen persistenten POA-Zustand. Es liegt in der Verantwortung des Anwendungsentwicklers, während der Server-Initialisierung den POA-Zustand mit den von ihm verwalteten Objekten wiederherzustellen bzw. einen Adapter-Activator zu installieren, um den Initialisierungsprozeß bei Bedarf automatisch durchführen zu lassen. Die Wiederherstellung des POA-Zustandes ist vor allem für die Verwaltung persistenter Objekte von Bedeutung, die über mehrere Server-Instanzen hinweg existieren. Dies setzt die Verwendung eines Hintergrundspeichers voraus. Um ein möglichst weites Spektrum von Anwendungen optimal zu unterstützen, wurde vollständig darauf verzichtet, die Art und Weise der Speicherung des POA-Zustandes durch die Spezifikation festzulegen. Solch eine Vorgabe würde einen Anwendungsentwickler bei der Festlegung des Speichermechanismus oder bei der Auswahl einer bestimmten Datenbank unnötig einengen und wäre nicht für alle Anwendungen gleichermaßen optimal.

Um persistente Objekte zu unterstützen, die in einer vorangegangenen Server-Instanz erzeugt wurden, muß der verantwortliche POA inklusive seiner Eltern-POAs neu erzeugt werden. Die Policies, mit den der POA konfiguriert wird, sowie die POA-Namen müssen mit denen des Original-Server-Prozesses übereinstimmen. Die Verantwortung dafür liegt in den Händen des Anwendungsentwicklers. Eine portable Serveranwendung kann davon ausgehen, daß es keine Konflikte zwischen den eigenen POA-Namen und den Namen, die in anderen Anwendungen verwendet werden, gibt. Die Sicherstellung dieses Verhaltens liegt in der Verantwortung des ORB-Herstellers.

Im Unterschied zur BOA-Spezifikation wird die Aktivierung von Serverprozessen bei Bedarf komplett aus der POA-Spezifikation ausgeklammert. Um die Aktivierung einer momentan nicht verfügbaren Serveranwendung zu ermöglichen, die die Implementierung persistenter Objekte bereitstellt, kann der ORB auf weitere Dienste zurückgreifen, wie bsw. einen *Activator Daemon* und das *Implementation Repository* (ImR). Ein solcher permanent laufender Daemon-Prozeß hat die Möglichkeit, die Anfrage an ein persistentes Objekt auf einem *well-known* Serverport entgegenzunehmen und nach einer Konsultierung des ImRs den verantwortlichen Serverprozeß zu starten. Die konkrete Implementierung bleibt dem ORB-Hersteller überlassen.

Objektreferenzen, Objekt-IDs und Servants

Der Umgang mit den drei Basis-Entitäten Objektreferenz, Objekt-ID und Servant steht bei der Arbeit des POAs im Mittelpunkt. Dazu ist häufig die Abbildung dieser Einheiten in die eine oder andere Richtung gefragt. Die POA-Schnittstelle hält für diese Konvertierungsaufgaben genau jene Operationen bereit, die es einem Anwendungsentwickler ermöglichen, eine bestimmte Instanz einer Basis-Entität (also eine Objektreferenz, Objekt-ID oder einen Servant) auf eine Instanz einer anderen Basis-Entität abzubilden. Die Operationen, die eine Objektreferenz zurückgeben, werden benötigt, um die Objektreferenz an einen Client exportieren zu können. Andersherum kann ein aktueller Servant für eine gegebene Objekt-ID oder Objektreferenz ermittelt werden. Vielfältige Nutzungsmöglichkeiten dieser Konvertierungsoperationen sind denkbar. Ihre Verwendung resultiert aus dem Umstand, daß der POA an der Verwaltung der Objekt-ID-zu-Servant-Beziehungen beteiligt werden kann, und der Entwickler auf diese Informationen zurückgreifen möchte.

4.2. POA-Policies

Eine zentrale Eigenschaft der POA-Spezifikation ist die Definition mehrerer POA-bezogener Policies, mit denen eine POA-Instanz parametrisiert werden kann und die in ihrer Summe das Verhalten eines POAs bestimmen. Da ein POA Objekt-Implementierungen verwaltet, kann man diese Formulierung auch auf die Objekte selbst übertragen. Alle innerhalb eines POAs inkarnierten CORBA-Objekte sollten eine ähnliche Charakteristik aufweisen. Aus diesem Grund werden sie von von einem, speziell an ihre Bedürfnisse angepaßten POA "betreut". Diese Anforderungen werden bei der Erzeugung eines POAs durch die Angabe passender Policies spezifiziert, und ihre Umsetzung wird vom POA kontrolliert.

Alle POA-bezogenen Policies sind im Modul `PortableServer` definiert und erweitern das `CORBA::Policy`-Interface. Diese Basis-Schnittstelle stellt einige grundlegene Management-Operation zum Kopieren und Zerstören einer Policy bereit. Ein *read-only* Attribut erlaubt die Ermittlung des aktuellen abgeleiteten Policy-Typs. Der `PolicyType` für eine abgeleitete Policy ist eine Integer-Konstante, die von der OMG vergeben wird, um die Eindeutigkeit aller Policy-Typen sicherzustellen. Eine ebenfalls in diesem Modul definierte `PolicyList` erlaubt die Zusammenstellung mehrer Policies in einer Liste. Eine derartige Liste wird bei der POA-Erzeugung übergeben (siehe Abschnitt 4.3). Die abgeleiteten POA-bezogenen Policies werden durch Operationen auf einer existierenden POA-Instanz erzeugt. Für jede Policy stellt der POA eine eigene Erzeugungsoperation bereit. Policies werden nicht von einem Eltern-POA an ein Kind-POA vererbt. Nach der POA-Erzeugung können die mit einem existierenden POA assoziierten Policies nicht mehr geändert werden.

Alle abgeleiteten POA-bezogenen Policies besitzen ein weiteres *read-only* Attribut, um den Wert einer Policy auslesen zu können. Den Wertebereich für jede einzelne Policy definiert ein im `PortableServer` Modul angegebenes `enum`. An dieser Stelle soll die IDL-Definition der `ThreadPolicy`, deren mögliche Werte und ihre Erzeugungsoperation beispielhaft für alle POA-bezogenen Policies dargestellt werden.

```
module PortableServer {
    const CORBA::PolicyType THREAD_POLICY_ID = 16;
    ...
    enum ThreadPolicyValue {
       ORB_CTRL_MODEL, SINGLE_THREAD_MODEL
    };
    ...
    interface ThreadPolicy:CORBA::Policy {
       readonly attribute ThreadPolicyValue value;
    };
    ...
    interface POA {
       ThreadPolicy create_thread_policy(in ThreadPolicyValue value);
        ...
    };
};
```

Die Policy-Werte bestimmen das Verhalten eines POA. Wenn bsw. ein POA mit einem `ThreadPolicy`-Objekt mit dem Wert **SINGLE_THREAD_MODEL** initialisiert wurde, dann werde ich für diesen Sachverhalt meist eine abgekürzte Formulierung gebrauchen und von einem POA sprechen, mit dem Policy-Wert **SINGLE_THREAD_MODEL** konfiguriert wurde. Glücklicherweise kann es an dieser Stelle keine Mißverständnisse geben, da die Bezeichner der Policy-Werte auch bei Betrachtung aller Policies eindeutig sind.

In den nun folgenden Abschnitten werden die POA-bezogenen Policies vorgestellt. Falls kein bestimmtes `Policy`-Objekt bei der Erzeugung eines benutzerdefinierten POAs übergeben wird, wird ein Standardwert für dessen Policy-Wert angenommen. Die Policy-Werte für den Root-POA sind ebenfalls durch die Spezifikation festgelegt. Nicht alle möglichen Kombinationen der Policy-Werte sind für die Konfiguration eines POAs zulässig. Einige ziehen die Verwendung bestimmter Policy-Werte nach sich, oder sie schliessen sich gegenseitig aus. Andere sind zwar erlaubt, aber nicht unbedingt sinnvoll. An gegebener Stelle werde ich auf die Abhängigkeiten zwischen bestimmten Policy-Werten hinweisen. Um die Existenz bestimmter Policies zu motivieren, werde ich auch kurz auf typische Verwendungsmöglichkeiten eingehen.

4.2.1. Objekt-Lebenszyklus

CORBA unterstützt die Verwendung persistenter Objekte. Diese überleben den Serverprozeß, in dem sie erzeugt und aktiviert wurden. Die an einen Client exportierte Objektreferenz eines solchen persistenten Objektes ist konzeptionell zeitlich unbegrenzt gültig. Dem Client wird die Illusion eines ständig verfügbaren Objektes vermittelt. Wenn er eine Anfrage an ein Zielobjekt startet, obwohl für dessen Implementierung momentan gar kein Serverprozeß läuft, also weder ein POA noch Servant verfügbar bzw. aktiv ist, dann kann die ORB-Implementierung einen Prozeß starten, der die verantwortliche Serveranwendung bei Bedarf hochfährt. Nach Abschluß der Server-Initialisierung wird die Anfrage dem neu erzeugten POA zugestellt, der wiederum die Auslieferung an den nun verfügbaren Servant übernimmt. Die Entscheidung, ob ein CORBA-Objekt tatsächlich existiert, wird bei jedem Aufruf neu gefällt. Für die Realisierung dieses Konzeptes bedarf es der Unterstützung durch die ORB-Infrastruktur und dem Zusammenspiel mehrerer ORB-Komponenten. Die ORB-Implementierung muß die Verfügbarkeit der serverseitigen ORB-Umgebung sicherstellen. Dies kann durch einen permanent verfügbaren Daemon-Prozeß auf einem *well-known* Port geschehen. Die Lokalisierung und Aktivierung der Objekt-Implementierung erfolgt in Zusammenarbeit mit dem ImR. Der POA selbst trägt seinen Anteil durch die Erzeugung geeigneter Objektreferenzen bei, die die Adreßinformationen des verfügbaren Daemon-Prozesses enthalten. Ansonsten läuft der gesamte Aktivierungsprozeß für ihn transparent ab.

Der POA kann anstelle persistenter Objekte auch transiente Objekte unterstützen. Die Lebenszeit transienter Objekte ist an die Lebenszeit einer POA-Instanz gebunden. Für viele Anwendungsfälle, in denen ein Objekt zeitlich begrenzt benötigt wird, reicht die Realisierung als transientes Objekt aus. Der Vorteil besteht im geringeren Admininistrationsaufwand für eine ORB-Umgebung und dem verminderten Overhead zur Lokalisierung einer Objekt-Implementierung. Die Aktivierung eines Serverprozesses entfällt natürlich gänzlich, da eine transiente Objektreferenz während eines Serverprozesses erzeugt und exportiert wird und nach dessen Beendigung keine Gültigkeit mehr besitzt. Ein einzelner POA unterstützt entweder nur persistente oder nur transiente Objekte. Ob ein POA persistente oder transiente Objektreferenzen erzeugt, wird mit der `LifespanPolicy` festgelegt.

LifespanPolicy-Werte

- **PERSISTENT**: Der POA unterstützt ausschließlich persistente Objekte.
- **TRANSIENT**: Der POA unterstützt ausschließlich transiente Objekte. Wird als Standardwert und für den Root-POA verwendet.

4.2.2. Objekt-IDs

Eine Objekt-ID ist als IDL-Typ `ObjectId` im Modul `PortableServer` als eine Sequenz von `octet` definiert. Diese Definition ist so allgemein gehalten, daß Daten jeglicher Art für die Erzeugung einer Objekt-ID verwendet werden können. Sollte der Zustand der Objekte innerhalb einer Datenbank verwaltet werden, bietet sich die Verwendung des Datensatz-Schlüssels an. Oftmals werden Strings bevorzugt, aber auch Zahlen, wie bsw. eine Personalnummer, oder andere Datentypen können ebenso in einer Sequenz von `octet` kodiert werden. Die einzige Bedingung, die gestellt wird, ist die, daß eine Objekt-ID ein CORBA-Objekt innerhalb eines POAs eindeutig bezeichnen muß.

```
module PortableServer {
    typedef sequence<octet> ObjectId;
};
```

Für viele Anwendungen ist es vorteilhaft, die verwendeten Objekt-IDs von der Anwendung festlegen zu lassen, da die zur Verfügung gestellten Objekte schon einen eindeutigen Bezeichner besitzen (z.B. das Schlüsselattribut aus einer Datenbank). Die Objekt-ID kann somit gleichzeitig genutzt werden, um auf den in einer Datenbasis gespeicherten Zustand zuzugreifen. Für andere Anwendungen, die bsw. transiente Objekte verwalten, wäre eventuell eine Festlegung der Objekt-ID durch die Anwendung ohne jeden Nutzen. Dann ist es hilfreich, wenn der POA die Generierung einer Objekt-ID übernehmen könnte und damit gleichzeitig deren Eindeutigkeit sicherstellt. Beide Anwendungsfälle werden durch den POA unterstützt. Der Entwickler muß sich bei der Verwendung eines POAs entscheiden, ob er Objekt-IDs generieren lassen oder er sie selbst erzeugen und dem POA zur Verfügung stellen will. Im letzteren Fall muß der Entwickler selbst die Eindeutigkeit der Objekt-IDs für diesen POA sicherstellen. Die Verantwortlichkeit für die Zuweisung einer Objekt-ID wird mit Hilfe der `IdAssignmentPolicy` konfiguriert.

IdAssignmentPolicy-Werte

- **USER_ID**: Objekt-IDs werden vom Entwickler zugewiesen.
- **SYSTEM_ID**: Objekt-IDs werden vom POA generiert. Wird als Standardwert und für den Root-POA verwendet.
- **Abhängigkeit**: Der Policy-Wert **USER_ID** erfordert die gleichzeitige Verwendung des **NO_IMPLICIT_ACTIVATION** Wertes für die `Implicit-ActivationPolicy` (beschrieben im folgenden Abschnitt).

4.2.3. Objektaktivierung

Die Aktivierung eines Objektes, also die Registrierung eines Servants für eine bestimmte Objekt-ID bei einem POA, kann *explizit* durch Aufruf spezieller POA-Operationen mit der Übergabe des zu registrierenden Servants erfolgen (siehe Abschnitt 4.4.2) oder *implizit* als Folge von POA-Operationen, die eine Assoziation zwischen einer Objekt-ID und einem Servant voraussetzen. Die implizite Objektaktivierung bietet dem Entwickler eine komfortable Möglichkeit, innerhalb einer Serveranwendung die Objektreferenz für eine Servant-Instanz zu erhalten und die dafür notwendige Objektaktivierung bei einem POA transparent im Hintergrund durchführen zu lassen. Voraussetzung ist die Verwendung systemgenerierter Objekt-IDs und die Verwaltung der aktivierten Objekte durch den POA. Alle Operationen, die eine implizite Objektaktivierung nach sich ziehen können, sind im Abschnitt 4.4.3 aufgeführt. Unter diesen befindet sich auch die sehr nützliche Servant-Operation `_this`, die im Java-Mapping definiert ist und die die Generierung einer Objekt-ID, die Registrierung des Servants unter der generierten Objekt-ID bei dem zuständigen POA und die Rückgabe einer Objektreferenz in einer Operation vereinigt (siehe Abschnitt 5.1.4).

Der Entwickler hat aber auch die Möglichkeit, die implizite Objektaktivierung für einen POA auszuschalten. Der Grund für einen Verzicht auf diese Funktionalität kann darin liegen, daß der Entwickler die Registrierung eines Servants bei einem bestimmten POA unter eigener Kontrolle behalten möchte, um versehentliche Aktivierungen bei einer anderen POA-Instanz auszuschliessen. Die kontextabhängige Ermittlung des zuständigen POAs ist dabei Quelle für gelegentliche Irritationen, da es entscheidend ist, ob die Operation, die eine implizite Objektaktivierung zur Folge hat, innerhalb des Kontextes eines Aufrufes auf dem Servant erfolgt. Das kontextabhängige Verfahren zur Ermittlung des verantwortlichen POAs wird ebenfalls im Abschnitt 5.1.4 beschrieben. Ein weiterer Grund für den Verzicht auf eine implizite Objektaktivierung könnte der Wunsch nach der Verwendung benutzerdefinierter Objekt-IDs sein. Die Anwendung der impliziten Objektaktivierung wird für einen POA mit Hilfe der `ImplicitActivationPolicy` konfiguriert.

ImplicitActivationPolicy-Werte

- **IMPLICIT_ACTIVATION**: Die implizite Objektaktivierung ist eingeschaltet. Wird für den Root-POA verwendet.[1]
- **NO_IMPLICIT_ACTIVATION**: Es ist ausschließlich explizite Objektaktivierung möglich. Wird als Standardwert verwendet.
- **Abhängigkeit**: Der Wert **IMPLICIT_ACTIVATION** erfordert die gleichzeitige Verwendung des **SYSTEM_ID** Policy-Wertes und des **RETAIN** Wertes für die im folgenden Abschnitt beschriebene `ServantRetentionPolicy`.

4.2.4. Objekt-ID-zu-Servant-Beziehungen

Die Verwaltung der Objekt-ID-zu-Servant-Beziehungen ist ein zentraler Aspekt bei der Konfiguration eines POAs. Neben der Entscheidung, welche Objekte einem

[1] Damit handelt es sich bei der `ImplicitActivationPolicy` um die einzige Policy, deren Wert für den Root-POA vom Standardwert abweicht.

POA zugeordnet werden, muß der Entwickler die Festlegung treffen, welche Programmkomponenten bei der Abbildung einer Anfrage auf eine Implementierung Verantwortung übernehmen. Die in diesem Zusammenhang relevanten Policies, deren Werte und die möglichen Kombinationen dieser Werte geben dem Entwickler mehrere Alternativen in die Hand, um eine Anwendung an die speziellen Anforderungen der Objekt-Implementierungen anzupassen. Welche Werte letztendlich zum Einsatz kommen, hängt zum einen von der Anzahl der Objekte und zum anderen von der Charakteristik der Objekt-Implementierungen ab. Das Spektrum der denkbaren Anwendungen ist breit gefächert und reicht von der Bereitstellung einzelner Objekt-Implementierungen, die ihren Zustand selbst verwalten, bis hin zu vielen tausend datenorientierten Objekten, deren Zustand in einer Datenbank gespeichert sein kann.

Der POA kann die Verwaltung der Objekt-ID-zu-Servant-Beziehungen übernehmen, indem er die Assoziationen in der *Active Object Map* speichert. Die explizite bzw. implizite Aktivierung eines Objektes mit der Bereitstellung eines Objekt-ID-Servant-Paares innerhalb der Initialisierungsphase der Serveranwendung stellt eine Möglichkeit dar, Einträge in die AOM einzufügen. Dieser statische Ansatz eignet sich nur für eine begrenzte Anzahl von Objekten. Das andere Extrem besteht in dem vollständigen Verzicht auf eine AOM oder einen Default-Servant und damit auf die Servant-Verwaltung durch den POA. In diesem Fall muß ein Servant-Manager beim POA registriert werden. Der vom Entwickler bereitgestellte Servant-Manager ermittelt anhand der Objekt-ID den für eine Anfrage verantwortlichen Servant und stellt ihn dem POA für die Durchführung des Aufrufes zur Verfügung. Die Verantwortlichkeit für die Servant-Verwaltung geht vollständig auf den Servant-Manager über. Neben dem Effekt der verzögerten und bedarfsgerechten Servant-Erzeugung wird somit ein Höchstmaß an Flexibilität bei der Abbildung von Objekt-IDs auf Servants erreicht. Diese Flexibilität muß der Entwickler natürlich mit einem erhöhtem Aufwand für die Servant-Verwaltung bezahlen.

Diese beiden gegensätzlichen Ansätze der Servant-Verwaltung können aber auch ebenso kombiniert werden. In Ergänzung zur Verwendung einer AOM kann ein Servant-Manager für den POA die Ermittlung eines noch unbekannten Servants übernehmen. Dieser wird dann mit der entsprechenden Objekt-ID in die AOM eingetragen, so daß der POA bei zukünftigen Anfragen den Servant selbst herausfinden kann. Ein alternativer Ansatz wäre, mehrere Objekte von einem Servant implementieren zu lassen, der dann für den POA transparent ein von der jeweilige Objekt-ID der aktuellen Anfrage abhängiges Verhalten implementiert.

Die Abbildung mehrerer Objekt-IDs auf einen Servant kann auf unterschiedliche Art erreicht werden. Zum einen kann der POA dies direkt unterstützen, indem er Einträge in die AOM erlaubt, die auf den selben Servant verweisen, oder der Entwickler stellt einen Default-Servant zur Verfügung, der für alle Anfragen benutzt wird, die nicht über die AOM vermittelt werden können. Bei der Bearbeitung einer Anfrage an ein bestimmtes Zielobjekt schaut der POA immer zuerst in einer vorhandenen AOM, um den mit der Objekt-ID des Zielobjektes assoziierten Servant für den Aufruf zu ermitteln. Wenn dem POA keine AOM zur Verfügung steht bzw. kein Eintrag unter dieser Objekt-ID in der AOM existiert, kann anstelle dessen ein eventuell vorhandener Default-Servant benutzt werden. Alternativ läßt sich dieses Verhalten wiederum mit einem Servant-Manager realisieren.

Die ausschließlich explizite bzw. implizite Aktivierung von Objekten während der Server-Initialisierung ist für bestimmte Anwendungsfälle durchaus ausreichend, würde den Entwickler aber in anderen Fällen einschränken. Wenn die zu unterstützenden Objekte beim Server-Start noch nicht feststehen oder wenn ihre Anzahl zu groß ist, dann leisten die zusätzlichen Komponenten wie Servant-Manager oder Default-Servant wertvolle Dienste.

Ob sich ein POA an der Verwaltung der Objekt-ID-zu-Servant-Beziehungen beteiligt, kann durch die `ServantRetentionPolicy` festgelegt werden.

ServantRetentionPolicy-Werte

- **NON_RETAIN**: Der POA verwendet keine AOM.
- **RETAIN**: Der POA verwendet eine AOM. Wird als Standardwert und für den Root-POA verwendet.
- **Abhängigkeiten**: Der Wert **NON_RETAIN** erfordert die gleichzeitige Verwendung des **NO_IMPLICIT_ACTIVATION** Policy-Wertes und die Verwendung des **USE_DEFAULT_SERVANT** bzw. **USE_SERVANT_MANAGER** Wertes für die `RequestProcessingPolicy` (siehe unten).

Die `RequestProcessingPolicy` bestimmt die Komponente eines POAs, die zur Ermittlung des verantwortlichen Servants herangezogen wird, um eine Anfrage zu bearbeiten.

RequestProcessingPolicy-Werte

- **USE_SERVANT_MANAGER**: Der POA benutzt einen Servant-Manager.
- **USE_DEFAULT_SERVANT**: Der POA benutzt einen Default-Servant.
- **USE_ACTIVE_OBJECT_MAP_ONLY**: Der POA benutzt ausschließlich die AOM zur Abbildung einer Objekt-ID auf einen Servant. Wird als Standardwert und für den Root-POA verwendet.
- **Abhängigkeiten**: Der Wert **USE_ACTIVE_OBJECT_MAP_ONLY** erfordert die gleichzeitige Verwendung des **RETAIN** Policy-Wertes. Der Wert **USE_DEFAULT_SERVANT** setzt den **NON_RETAIN** Policy-Wert oder den **MULTIPLE_ID** Wert für die `IdUniquenessPolicy` voraus (siehe unten).

Die Eindeutigkeit der Objekt-ID-zu-Servant-Beziehungen innerhalb einer AOM läßt sich mit Hilfe der `IdUniquenessPolicy` konfigurieren.

IdUniquenessPolicy-Werte

- **MULTIPLE_ID**: Ein Servant kann mit mehreren Objekt-IDs assoziiert sein.
- **UNIQUE_ID**: Der POA garantiert die eindeutige Zuordnung eines Servants zu einer Objekt-ID. Wird als Standardwert und für den Root-POA verwendet.

4.2.5. Thread-Modell

Wenn eine Server-Umgebung den Ablauf der Anwendung in mehreren Threads unterstützt, ist es auch möglich, die von einem POA zu bearbeitenden Anfragen und somit die Aufrufe auf den Servants innerhalb von eigenen Threads nebenläufig durchzuführen. Neben der notwendigen Thread-Unterstützung durch das darunterliegende Betriebssystem und die Entwicklungsumgebung hängt die Entscheidung, ob Threads für die Bearbeitung von Anfragen eingesetzt werden können, von der Robustheit der Objekt-Implementierungen gegenüber dem Einsatz in einer *multi-threaded* Umgebung ab. Für die Konfiguration eines POAs stehen zwei verschiedene Thread-Modelle zur Auswahl. Für den Fall, daß eine Anwendung selbst keine geeigneten Synchronisationsmaßnamen trifft, da sie nicht für eine *multi-threaded* Umgebung entwickelt wurde, kann ein POA mit dem *Single Thread Model* konfiguriert werden, so daß alle Aufrufe auf den vom Entwickler zur Verfügung gestellte Komponenten (Servant, Servant-Manager, Adapter-Activator) serialisiert werden. Die Integration existierender Alt-Systeme kann somit vereinfacht werden.

Die andere Möglichkeit besteht darin, die Zuweisung von Threads für die Anfragebearbeitung der ORB-Implementierung zu überlassen. Bei der Verwendung des *ORB Controlled Models* ist es Sache der ORB-Entwickler, die Möglichkeiten des Betriebssystems bzw. der Entwicklungsumgebung für den Einsatz von Threads auszunutzen. Durch die Spezifikation ist lediglich festgelegt, daß sich ein Entwickler beim *ORB Controlled Model,* im Gegensatz zum *Single Thread Model,* der möglichen Verwendung seiner Anwendung in einer *multi-threaded* Umgebung bewußt sein und er selbst für die notwendige Serialisierung innerhalb der Anwendung Sorge tragen muß. Dieses Modell muß aber ebenso in Umgebungen angeboten werden, die keine Threads unterstützen und demzufolge auch keine verwenden können. Aus diesem Grund ist die Verwendung des *ORB Controlled Models* nicht zwangsläufig für jede ORB-Implementierung mit der Verwendung von Threads für die Anfragebearbeitung gleichzusetzen.

Innerhalb einer Serveranwendung können durchaus unterschiedliche Thread-Modelle für einzelne POA-Instanzen zum Einsatz kommen. Die Verwendung des *Single Thread Models* für einen POA ist prinzipiell unabhängig von der Möglichkeit, eine Anwendung innerhalb mehrerer Threads laufen zu lassen. Da sich mehrere *single-threaded* POAs einen einzigen Thread für Servant-Aufrufe teilen, ist auch in dem Fall, daß ein Servant bei mehreren *single-threaded* POAs registriert wurde, die Serialisierung in einer solchen *multi-threaded* Serveranwendung gewährleistet. Das Thread-Modell für einen POA wird mit der `ThreadPolicy` konfiguriert.

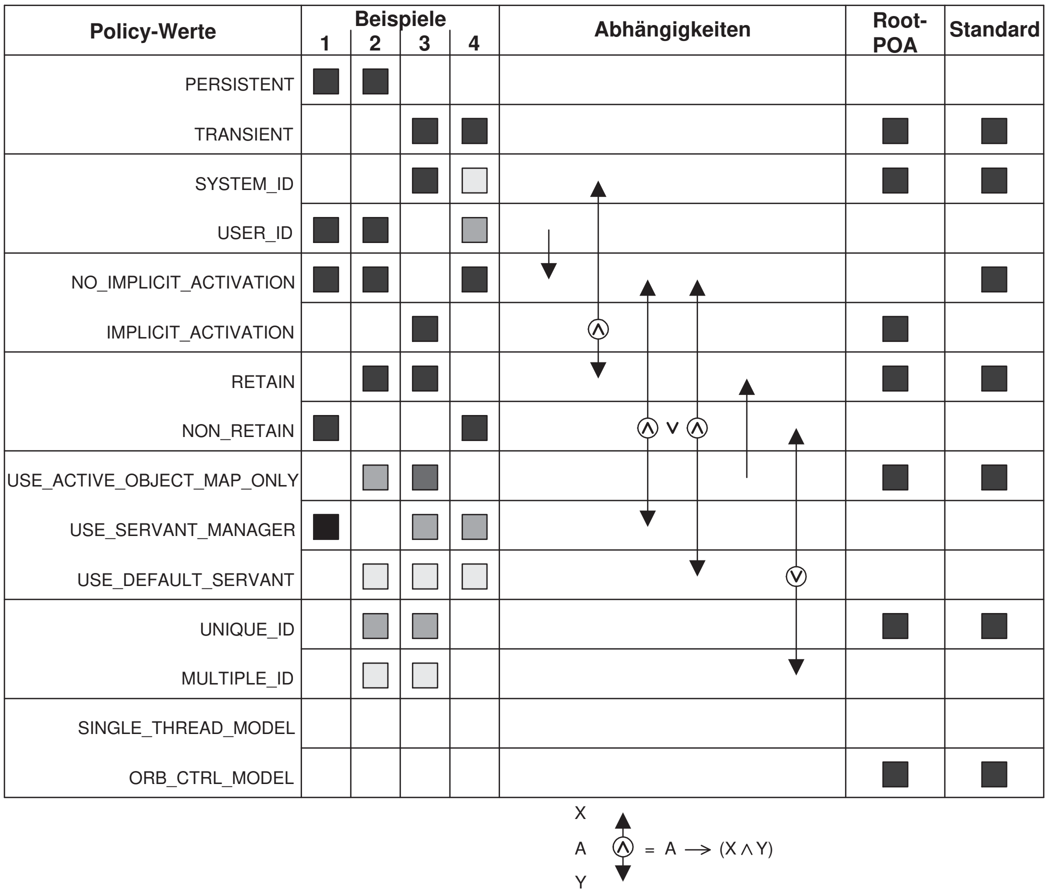
ThreadPolicy-Werte

- **SINGLE_THREAD_MODEL**: Der serialisierte Ablauf von Anwendungscode wird erzwungen (Servant, Servant-Manager, Adapter-Activator).

- **ORB_CTRL_MODEL**: Der eventuell nebenläufige Ablauf von Anwendungscode innerhalb von Threads wird von der ORB-Implementierung festgelegt. Wird als Standardwert und für den Root-POA verwendet.

4.2.6. Einsatz der Policy-Werte

Zum Abschluß der Beschreibung aller Policies möchte ich die verschiedenen Policy-Werte in einer Gesamtübersicht darstellen. In der Abbildung 4.3 sind neben den Abhängigkeiten zwischen den Policy-Werten, dem Wert für den Root-POA und dem Standardwert einige typische POA-Konfigurationen anhand von vier Beispielen aufgeführt. Die Beispiele sind zwar sehr allgemein formuliert, sie geben jedoch einen Überblick über die Einsatzmöglichkeiten des POAs.



Beispiel 1: Große Anzahl persistenter Objekte, deren Zustand in einem Hintergrundspeicher verwaltet wird (Daten-Objekte).
Beispiel 2: Kleine Anzahl persistenter Objekte (bzw. ein Objekt), die öffentliche Dienste bereitstellen (Service-Objekte).
Beispiel 3: Kleine bis mittlere Anzahl transienter Objekte, die Hilfsfunktionalität bereitstellen.
Beispiel 4: Unbekannte bzw. potentiell große Anzahl transienter Objekte, deren Rolle an den Serverprozeß gebunden ist.

Abbildung 4.3.: Einsatz der Policy-Werte

Die vorgeschlagenen Policy-Werte sind kein Dogma, im Rahmen der zulässigen Möglichkeiten sind auch abweichende POA-Konfigurationen denkbar. An einigen Stellen sind auch Alternativen angegeben. In der Übersicht sind diese durch einen helleren Grauton gekennzeichnet. Wenn kein Wert für eine bestimmte Policy gekennzeichnet wurde, dann ist dieser für das Problem nicht relevant, bzw. er unterliegt spezielleren Entwurfskriterien.

Zu den wichtigsten Entwurfskriterien gehören die erwartete Anzahl der von einem POA "betreuten" Objekte und die Entscheidung, ob es sich um transiente oder persistente Objekte handelt. Bei einer potentiell großen bzw. unbekannten Anzahl von Objekten kommen immer die dynamischen Fähigkeiten des POAs ins Spiel, Servants erst bei Bedarf zu aktivieren, oder im Extremfall einen Servant-Manager oder einen Default-Servant bereitzustellen, der als Vermittler zwischen der Anwendung und dem POA fungiert. Die Abbildung der verwalteten Objekte auf ihre Implementierungen wird dann für jeden Aufruf neu vorgenommen. Diese Art der Entkopplung zwischen POA und Anwendung ist auch für existierende Alt-Systeme interessant, denen mit geringem Aufwand CORBA-Funktionalität beigebracht werden soll.

Das andere Extrem bildet eine POA-Konfiguarion, die nur wenige Objekte verwaltet. Diese kommen meist mit den Standardfähigkeiten des POAs aus, wie der Verwaltung der Objeke in der AOM oder, bei einem einzelnen Objekt, durch den Default-Servant. Bei der Suche nach der bestmöglichen POA-Konfiguarion muß jedesmal ein Kompromiß zwischen Speicherverbrauch, Laufzeiteffizienz und der gewünschten Komfortabilität gefunden werden. Die dynamische Ermittlung einer Objekt-Implementierung und Zugriffe auf einen Hintergrundspeicher, um einen Objektzustand wiederherzustellen, kosten Zeit, sparen aber Speicherplatz durch die Einsparung der AOM.

Andere Kriterien hängen sehr stark von den speziellen Anforderungen der Anwendung ab oder sind orthogonal zueinander. Demzufolge lassen sich kaum allgemeingültige Aussagen treffen. Bsw. hängt das einzustellende Thread-Modell oftmals nur von der Multi-Threading-Fähigkeit der Implementierung ab. Ob eine Anwendung die implizite Objektaktivierung gestattet, ist meist, sofern diese nicht mit anderen Policy-Werten kollidiert, ein Implementierungsdetail. Die Generierung von Objekt-IDs durch den POA wird typischerweise im transienten Fall eingesetzt. Für persistente Objekte bietet sich gewöhnlich ein logischer Bezeichner oder ein Datensatzschlüssel an.

Eine tiefere Behandlung von Entwurfsmustern POA-basierter Serveranwendungen bietet [8].

4.3. POA-Erzeugung

Um einen benutzerdefinierten POA zu erzeugen, muß die `create_POA()` Operation auf einer vorhandenen POA-Instanz aufgerufen werden. Jede server-seitige ORB-Umgebung besitzt einen vom ORB verwalteten Root-POA, der als Startpunkt für die Erzeugung weiterer `POA`-Instanzen dient. Der Root-POA kann über die ORB-Initialisierungsschnittstelle durch Aufruf der `resolve_initial_references()` Operation mit dem Parameter "RootPOA" ermittelt werden. Jeder neu erzeugte POA ordnet sich als Kind-POA des erzeugenden Eltern-POAs in die POA-Hierarchie ein. Der mit der `create_POA()` Operation übergebene `adapter_name` muß den Kind-POA innerhalb des Eltern-POAs eindeutig bezeichnen. Sollte schon ein Kind-POA unter diesem Namen existieren, wird eine **AdapterAlreadyExists** User-Exception geworfen. Als zweiter Parameter kann ein POA-Manager übergeben werden, mit dem der POA assoziiert werden soll. Wird anstelle dessen der Wert **null** übergeben, muß eine neue `POAManager`-Instanz erzeugt werden. Die Funktionalität des POA-Managers wird im Abschnitt 4.11 beschrieben. Das dritte Argument der Operation kann eine `PolicyList`, also eine Sequenz von `Policy`-Objekten bzw. den Wert **null** enthalten. Für die Policies, die bei der POA-Erzeugung nicht übergeben werden, werden die Standardwerte angenommen. Im Fall sich widersprechender Kombinationen der Policy-Werte, bei der Verwendung eines unbekannten Policy-Typs oder falls durch die ORB-Implementierung festgelegte Voraussetzungen nicht erfüllt sind, wird eine **InvalidPolicy** User-Exception geworfen, die den Index der ersten unzulässigen Policy aus der Liste enthält (siehe Abhängigkeiten im Abschnitt 4.2). Übergebene Policies werden für die Assoziation mit dem POA kopiert. Ein neu erzeugter POA hält also keine Referenzen auf die ihm übergebenen `Policy`-Objekte.

```
module PortableServer {
    exception AdapterAlreadyExists {};
    exception InvalidPolicy {
        unsigned short index;
    };
    interface POA {
        POA create _POA(
                in string adapter_name,
                in POAManager poa_manager,
                in CORBA::PolicyList policies
            ) raises(AdapterAlreadyExists, InvalidPolicy);
    };
};
```

Die für einen neu erzeugten POA geltenden Policies sind vollständig unabhängig von den Policies des Eltern-POAs. Sie werden also nicht vererbt, sondern alle von den Standardwerten abweichenden Policies müssen bei der POA-Erzeugung angegeben und können im nachhinein nicht mehr geändert werden. Die Zuweisung eines Apapter-Activators, eines Servant-Managers oder eines Default-Servants ist Teil der POA-Initialisierung, sie wird aber erst im Anschluß an die POA-Erzeugung vorgenommen. Die POA-Schnittstelle stellt dafür entsprechende Operationen bereit.

4.4. Objekterzeugung und -aktivierung

4.4.1. Objekterzeugung

Die `POA`-Schnittstelle hält zwei Operationen bereit, um eine Objektreferenz zu erzeugen, ohne aber einen Servant beim POA zu registrieren. Die `create_reference()` Operation ist für die Verwendung mit einem POA gedacht, der mit dem **SYSTEM_ID** Policy-Wert konfiguriert wurde. Ihr wird eine *Interface Repository Id* als String übergeben. Dieser bezeichnet das *most derived* IDL-Interface, das das neue Objekt unterstützt. Erweitert solch ein spezialisiertes IDL-Interface irgendwelche Basis-Interfaces, dann werden diese von dem neuen Objekt ebenso unterstützt. Die für die Objektreferenz-Erzeugung notwendige Objekt-ID wird vom POA generiert. Sollte für den POA nicht der **SYSTEM_ID** Policy-Wert gelten, wird eine **WrongPolicy** User-Exception geworfen.

```
module CORBA {
    typedef string RepositoryId;
};

module PortableServer {
    interface POA {
        Object create_reference(
                    in CORBA::RepositoryID intf_rep_id
               ) raises(WrongPolicy);

        Object create_reference_with_id(
                    in ObjectId oid,
                    in CORBA::RepositoryID intf_rep_id
               );
    };
};
```

Für den anderen Fall, daß neben der Repository- auch die Objekt-ID durch die Anwendung übergeben werden soll, kann die `create_reference_with_id()` Operation benutzt werden. Die bevorzugte Anwendung liegt sicherlich bei der Verwendung mit einem **USER_ID** POA, wenn also Objekt-IDs benutzt werden sollen, die von der Anwendung bereitgestellt werden. Dieser Anwendungsfall stellt aber trotzdem keine Bedingung dar. Die Operation kann theoretisch genauso auf einem **SYSTEM_ID** POA aufgerufen werden. In diesem Fall muß die Anwendung dafür sorgen, daß die übergebene Objekt-ID, je nach geltender `LifespanPolicy`, von dieser oder einer vorangegangenen POA-Instanz erzeugt wurde. Der ORB selbst ist nicht zur Sicherstellung dieser Forderung verpflichtet. Wenn er aber solch eine Fehlersituation erkennt, hat diese Operation eine **BAD_PARAM** System-Exception zur Folge.

Der mehrmalige Aufruf der Operationen `create_reference_with_id()` mit der gleichen Objekt- und Repository-ID ist legal, kann aber abhängig von der ORB-Implementierung zu unterschiedlichen Ergebnissen führen. Einige ORBs werden einen Caching-Mechanismus implementieren, so daß jeder Aufruf die einmal erzeugte Objektreferenz zurückliefert. Andere könnten für jeden Aufruf eine neue Instanz erzeugen. Außer den Konsequenzen, die den Speicherbedarf und den Vergleich von Objektreferenzen betreffen, ist natürlich auch im letzteren Fall sichergestellt, daß Aufrufe auf unterschiedlichen Kopien von Objektreferenzen auf den gleichen Servant abgebildet werden, da sie alle die gleiche Objekt-ID enthalten und eine Abbildung auf einen Servant anhand der Objekt-ID vorgenommen wird. Es gibt keine Möglichkeit, den mehrmaligen Aufruf mit einer gleichen Objekt-ID, aber mit völlig voneinander unabhängigen Repository-IDs, zu verhindern. Der POA wird keine Exception werfen. Das Erkennen solch einer Fehlersituation würde unverhältnismäßig hohe Verwaltungskosten verursachen. Im Falle eines transienten POAs ist es eventuell noch vertretbar, für einen persistenten POA müßten alle jemals erzeugten Objektreferenzen einbezogen werden. Im Gegensatz dazu führt der mehrmalige Aufruf der `create_reference()` Operation zu dem Ergebnis, daß für jeden neuen Aufruf, bei gleicher Repository-ID, eine neue Objekt-ID generiert und für die Objektreferenz-Erzeugung verwendet wird. Aufrufe auf diesen Objektreferenzen führen in der Regel nicht zum gleichen Servant.

Die Möglichkeit, eine Objektreferenz zu erzeugen und zu exportieren, ohne einen Servant bei einem POA zu registrieren, ist die Voraussetzung dafür, Objekt-Implementierungen erst bei Bedarf bereitstellen zu können. Dieser Umstand zeigt deutlich, daß der Lebenszyklus eines CORBA-Objektes vollständig unabhängig von dem eines Servants ist. Wenn bsw. sehr viele Objekte von einem POA verwaltet werden sollen, wäre es nicht sinnvoll alle Servants im Vorfeld beim POA anzumelden, wenn für einen Großteil gar keine Anfragen zu erwarten sind. In Verbindung mit einem Servant-Manager kann dieses Problem gelöst werden (siehe Abschnitt 4.4.4).

4.4.2. Explizite Objektaktivierung

Der direkteste Weg, ein Objekt bei einem POA zu aktivieren, ist explizit für eine Objekt-ID einen Servant zu registrieren. Analog zur Objektreferenz-Erzeugung stellt der POA zwei Varianten der Aktivierungsoperation bereit. Bei der Operation `activate_object()` wird die zur Registrierung benötigte Objekt-ID vom POA generiert und als Ergebnis zurückgeliefert. Mit der `activate_object_with_id()` Operation kann die Objekt-ID übergeben werden. Beiden Operationen wird der zu registrierende Servant übergeben. Die Konfiguration des POAs mit dem **RETAIN** Policy-Wert ist Voraussetzung für die explizite Objektaktivierung. Für die Verwendung der `activate_object()` Operation wird zusätzlich noch der **SYSTEM_ID** Policy-Wert gefordert. Sollte der POA nicht mit diesen Policies konfiguriert worden sein, wird eine **WrongPolicy** User-Exception geworfen.

```
module PortableServer {
    native Servant;
    interface POA {
        ObjectId activate_object(
                   in Servant servant
               ) raises(ServantAlreadyActive, WrongPolicy);

        void activate_object_with_id(
                   in ObjectId oid,
                   in Servant servant
               ) raises(ObjectAlreadyActive, ServantAlreadyActive,
                        WrongPolicy);
    };
};
```

Wenn bei einem **UNIQUE_ID** POA der übergebene Servant schon in der AOM enthalten sein sollte, wird bei beiden Operationen eine **ServantAlreadyActive** User-Exception generiert. Die Eindeutigkeit der Objekt-IDs in der AOM muß in jedem Fall gewährleistet bleiben. Generiert der POA selbst die Objekt-ID, stellt er damit sicher, daß die neu generierte Objekt-ID niemals zuvor erzeugt und damit auch nicht aktiviert wurde. Wird die Objekt-ID von der Anwendung bereitgestellt, so muß geprüft werden, ob die übergebene Objekt-ID schon in der AOM verzeichnet ist. Wenn sie schon enthalten ist, wird eine **ObjectAlreadyActive** User-Exception geworfen.

Bei einem **MULTIPLE_ID** POA führt ein mehrmaliger Aufruf der Operation `activate_object` mit demselben Servant als Parameter auch zu mehreren Objektaktivierungen. Aufgrund der Tatsache, daß ein Servant unter mehreren Objekt-IDs registriert sein kann, wird jedesmal eine neue Objekt-ID generiert und diese zusammen mit dem Servant in die AOM eingeragen. Analog zur Operation `create_reference_with_id` wird die Operation `activate_object_with_id` vorzugsweise im Zusammenhang mit einem **USER_ID** POA genutzt werden. Trotzdem ist es gestattet, diese Operationen auch auf einem **SYSTEM_ID** POA aufzurufen. Ein POA hat aber in diesem Fall die Möglichkeit, eine Objekt-ID mit einer **BAD_PARAM** System-Exception zurückzuweisen, die nicht von ihm selbst oder einer früheren Instanz (eines persistenten POAs) erstellt wurde. Die bei der Objektaktivierung zurückgegebene- bzw. übergebene Objekt-ID kann später innerhalb der Serveranwendung genutzt werden, um eine Objektreferenz zu erzeugen, auf den registrierten Servant zuzugreifen oder um das Objekt wieder zu deaktivieren. Die dafür vorgesehenen Operationen werden in den Abschnitten 4.9 und 4.10 beschrieben.

4.4.3. Implizite Objektaktivierung

Wenn ein POA mit den Policy-Werten **RETAIN** und **SYSTEM_ID** konfiguriert wurde, reicht die Übergabe eines Servants aus, um eine Objektaktivierung durchzuführen. Der Objektaktivierung folgt üblicherweise die Erzeugung einer Objektreferenz mit dem Ziel, diese einem Client für den Aufruf von Operationen zur Verfügung zu stellen. Ein POA kann mit dem Policy-Wert **IMPLICIT_ACTIVATION** auch so konfiguriert werden, daß er eine Objektaktivierung automatisch durchführt und zwar immer dann, wenn der Entwickler eine Operation aufruft, der eine Servant-Instanz übergibt und die eine logische Verbindung zwischen einer Objekt-ID und einem Servant voraussetzt, diese aber noch nicht existiert. Zwei der im Abschnitt 4.9 beschriebenen Konvertierungsoperationen erlauben die Abbildung eines Servants auf eine Objekt-ID oder eine Objektreferenz. Wenn der übergebene Servant noch nicht registriert wurde, kann ein POA die Generierung einer Objekt-ID und die Eintragung in die AOM automatisch vornehmen, um dann die Objekt-ID bzw. die Objektreferenz zurückzugeben. Die Möglichkeit der impliziten Objektaktivierung bietet dem Entwickler einen höheren Komfort bei der Anwendungsentwicklung, sofern die Kombination von **RETAIN** und **SYSTEM_ID** Policy-Werten für den Anwendungsfall ausreichend ist. Folgende Operationen unterstützen die implizite Objektaktivierung:

- `POA::servant_to_reference()`, die einen Servant auf eine Referenz abbildet,
- `POA::servant_to_id()`, die einen Servant auf eine Objekt-ID abbildet
- und im Language-Mapping definierte Operationen, die eine Objektreferenz oder eine Objekt-ID für einen Servant ermitteln. Bsw. wird im Java-Mapping die `Servant`-Operation `_this` definiert, die es erlaubt, über einen Servant eine lokale Objektreferenz des CORBA-Objektes zu erhalten, das durch den Servant inkarniert wird (siehe Abschnitt 5.1.4).

4.4.4. Servant-Manager

Für einige Anwendungsfälle ist die Registrierung aller Servants beim POA sicher nicht die beste Wahl. Wenn sehr viele Objekte von einer Serveranwendung verwaltet werden sollen, dann würde solch ein Vorgehen einen unvertretbar hohen Verbrauch an Ressourcen für die Speicherung der Assoziationen zwischen Object-IDs und Servants in der AOM bedeuten. Die Verwendung eines Servant-Managers ermöglicht es, die Verwaltung dieser Beziehungen dynamischer zu gestalten und einen Servant erst bei einer konkreten Anfrage bereitzustellen. Servant-Manager sind Callback-Objekte, die der Anwendungsentwickler zur Verfügung stellt und die den POA aktiv bei der Verwaltung der Object-ID-zu-Servent-Beziehungen unterstützen können. Wenn ein POA eine Anfrage an ein bestimmtes Zielobjekt erhält, kann er mit der Hilfe des Servant-Managers den verantwortlichen Servant zur Bearbeitung der Anfrage ermitteln.

Der POA besitzt zwei Operationen, mit denen sich der Servant-Manager setzen bzw. ermitteln läßt. Voraussetzung ist die Konfiguration des POAs mit dem **USE_SERVANT_MANAGER** Wert für die `RequestProcessingPolicy`. In der Spezifikation werden zwei Versionen eines Servant-Managers definiert. Je nach Wert für die `ServantRetentionPolicy` muß der verwendete Servant-Manager ein bestimmtes IDL-Interface aus dem `PortableServer`-Modul implementieren. Gilt der **RETAIN** Wert, muß er das `ServantActivator`-Interface unterstützen, im Fall von **NON_RETAIN** ist es das `ServantLocator`-Interface. Das `ServantManager`-Interface ist leer und dient als Basis-Interface für diese beiden spezialisierten Interfaces und ermöglicht lediglich die einheitliche Verwendung beider Servant-Manager-Typen innerhalb von POA-Operationen. Wenn der POA nicht mit dem Policy-Wert **USE_SERVANT_MANAGER** konfiguriert sein sollte, erzeugen die Operationen zum Setzen und Lesen eines `ServantManager`-Objektes eine **WrongPolicy** User-Exception. Die Operation zum Setzen des Servant-Managers darf nur einmal nach der Erzeugung des POAs aufgerufen werden. Jeder Versuch einen einmal gesetzten Servant-Manager zu ändern, hat eine **BAD_INV_ORDER** System-Exception zur Folge.

```
module PortableServer {
    interface ServantManager {};

    interface POA {
        void set_servant_manager(
                 in ServantManager servant_manager
             ) raises(WrongPolicy);

        ServantManager get_servant_manager()
               raises(WrongPolicy);
   };
};
```

Servant-Activator

Wenn ein POA, der mit einem Servant-Activator assoziiert wurde, eine Anfrage erhält, deren Objekt-ID er nicht anhand der AOM auflösen kann, dann ruft er die `incarnate()` Operation auf dem `ServantActivator`-Objekt auf und übergibt die in der Anfrage enthaltene Objekt-ID und eine Referenz auf sich selbst. Die vom Entwickler bereitgestellte Implementierung der `incarnate()` Operation liefert eine Instanz eines Servants zurück, die der POA benutzen kann, um den Objekt-Aufruf auszuführen. Der POA trägt den zurückgelieferten Servant zusammen mit der Objekt-

ID in die AOM ein, so daß er für zukünftige Anfragen an das selbe Zielobjekt den Servant-Manager nicht mehr zu konsultieren braucht. Der Servant-Manager bildet seinerseits die übergebene Objekt-ID auf einen Servant ab, bzw. er lokalisiert oder erzeugt ggf. einen Servant für diese Objekt-ID. Der POA kann den Servant aber nur für einen Aufruf verwenden, wenn auch die der Servant-Inkarnation folgende Objektaktivierung erfolgreich durchgeführt werden kann. Wenn ein Servant zurückgeliefert wird, der bereits für eine andere Objekt-ID aktiviert ist und der **UNIQUE_ID** Policy-Wert für diesen POA gilt, wird eine **OBJ_ADAPTER** System-Exception generiert. Die Implementierung der `incarnate()` Operation kann selbst ebenfalls eine CORBA-System-Exception werfen, wie z.B. **OBJECT_NOT_EXIST**, was in jedem Fall zu einem Abbruch der Bearbeitung der Anfrage und zu einer Rücksendung der Exception an den Client führt.

Neben der verzögerten Bereitstellung eines Servants im Bedarfsfall kann mit Hilfe eines Servant-Managers die Umleitung einer Anfrage an ein anderes CORBA-Objekt implementiert werden. Zu diesem Zweck wird im `PortableServer`-Modul eine **ForwardRequest** User-Exception definiert, die von der `incarnate()` Operation geworfen werden kann, um die Anfrage an ein anderes Objekt weiterzuleiten. Die in der Exception enthaltene Objektreferenz wird vom ORB innerhalb des **LOCATION_FORWARD** Reply-Status an den Client zurückgesendet, der die Anfrage nun an das neue Objekt schicken kann.

```
module PortableServer {
    exception ForwardRequest {
        Object forward_reference;
    };
    interface ServantActivator : ServantManager {
        Servant incarnate(
                    in ObjectId oid,
                    in POA adapter
                ) raises (ForwardRequest);

        void    etherealize(
                    in ObjectId oid,
                    in POA adapter,
                    in Servant servant,
                    in boolean cleanup_in_progress,
                    in boolean remaining_activations
                );
    };
};
```

Das Gegenstück zu `incarnate()` ist die `etherialize()` Operation. Mit ihr wird der Servant-Manager über die Deaktivierung eines Objektes benachrichtigt. Dieser hat damit die Möglichkeit, Aufräumungsarbeiten auf dem vormals verantwortlichen Servant durchzuführen. Der POA übergibt dem Servant-Manager die Objekt-ID des deaktivierten Objektes, eine Referenz auf sich selbst, den Servant und zwei Flags, die den Kontext des `etherealize()` Aufrufes beschreiben. Wird der POA selbst deaktiviert oder zerstört, wobei für alle aktivierten Objekte die `etherealize()` Operation ausgeführt werden soll, wird der `cleanup_in_progress` Parameter auf den Wert

TRUE gesetzt. Der `remaining_activations` Parameter hat den Wert **TRUE**, wenn der übergebene Servant noch für weitere Objekt-ID-zu-Servant-Beziehungen genutzt wird (bei einem **MULTIPLE_ID** POA). Erzeugt die Implementierung von `etherealize()` irgendeine Exception, wird sie vom POA ignoriert. Zusammenfassend lassen sich folgende Situationen angeben, in denen die `etherealize()` Operation aufgerufen wird:

- während der expliziten Objektdeaktivierung durch den Aufruf von `POA::deactivate_object()` (siehe Abschnitt 4.10),
- wenn der POA selbst deaktiviert wird durch `POAManager::deactivate()` (siehe Abschnitt 4.11) oder er zerstört wird durch `POA::destroy()` (siehe Abschnitt 4.12),
- wenn ORB oder POA intern feststellen sollten, daß ein Objekt deaktiviert werden muß. Dies kann z.B. der Fall sein, wenn ein Limit der AOM-Implementierung erreicht wurde, um Platz zu schaffen für neue Objektaktivierungen.

Für *multi-threaded* Serveranwendungen werden von der POA-Implementierung bestimmte Garantien übernommen, um eine ungewollte mehrfache Servant-Inkarnation bzw. -Etherealization oder deren gegenseitige Beeinflussung innerhalb nebenläufiger Threads zu verhindern.

- Alle Aufrufe von `incarnate()` und `etherealize()` auf einem Servant-Manager werden vom POA serialisiert und schließen sich gegenseitig aus.
- Aufrufe von `incarnate()` mit der selben Objekt-ID können nicht mehrfach durchgeführt werden, d.h. eine Servant-Inkarnation unter dieser Objekt-ID kann erst wieder erfolgen, wenn zwischenzeitlich eine Servant-Etherealization stattfand. Neue Anfragen an ein im Deaktivierungsprozeß befindliches Objekt werden gepuffert, bis der Aufruf von `etherealize()` beendet ist.

Mehrere `POA`-Instanzen sind nicht verpflichtet in irgendeiner Weise zu kooperieren, um diese Garantien auch in dem Fall abgeben zu können, wenn ein Servant-Activator mit mehreren `POA`-Instanzen assoziiert wird. Um ein daraus resultierendes Fehlverhalten auszuschliessen, sollte die Assoziation eines Servant-Activators mit mehreren `POA`-Instanzen vermieden werden.

Servant-Locator

Ein mit einem Servant-Locator assoziierter POA verwaltet selbst keine Servants in einer AOM. Zur Auslieferung einer Anfrage hat er die Möglichkeit, über den Servant-Locator durch Aufruf der `preinvoke()` Operation den zuständigen Servant in Erfahrung zu bringen. Der ermittelte Servant ist nur für die Bearbeitung dieser einen Anfrage gültig. Der Aufruf von `preinvoke()` muß für jede Anfrage erneut durchgeführt werden. Nachdem der eigentliche Servant-Aufruf ausgeführt wurde, ruft der POA für dieses Zielobjekt die Operation `postinvoke()` auf dem Servant-Locator auf. Dieser hat damit die Möglichkeit, Aufräumungsarbeiten, wie z.B. das Zurückspeichern eines Objektzustandes in eine Datenbank, durchzuführen. Der Servant-Aufruf wird also von den beiden `ServantLocator`-Operationen eingerahmt.

```
module CORBA {
    typedef string Identifier;
};
module PortableServer {
    interface ServantLocator : ServantManager {
        native Cookie;

        Servant preinvoke(
                    in ObjectId oid,
                    in POA adapter,
                    in CORBA::Identifier operation,
                    out Cookie the_cookie
                ) raises (ForwardRequest);

        void    postinvoke(
                    in ObjectId oid,
                    in POA adapter,
                    in CORBA::Identifier operation,
                    in Cookie the_cookie,
                    in Servant servant
                );
    };
};
```

Der POA übergibt der `preinvoke()` Operation die Objekt-ID des Zielobjektes, eine Referenz auf sich selbst, einen Identifier, der den Namen der Operation als String bezeichnet, und ein `Cookie`-Objekt. Ein `Cookie` ist als *native type* im jeweiligen Language-Mapping definiert. Für das Java-Mapping wird ein `Cookie` auf ein `java.lang.Object` abgebildet. Bei der `preinvoke()` Operation ist der `Cookie`-Parameter als **out**-Parameter angegeben. Dies hat bsw. für das Java-Mapping zur Folge, daß an dieser Stelle ein Exemplar einer generierten `CookieHolder`-Klasse übergeben wird, die als einziges Attribut ein `java.lang.Object` enthält. Darin kann der Entwickler ein beliebiges Programmiersprachenobjekt speichern. Die Implementierung von `preinvoke()` hat nun die Möglichkeit, dieses Objekt in irgendeiner Weise zu manipulieren oder zu setzen. Damit lassen sich beliebige Informationen speichern, auf die später innerhalb der `postinvoke()` Operation wieder zugegriffen werden kann. Der POA ist verpflichtet, das im `CookieHolder`-Objekt gespeicherte `Cookie`-Objekt beim Aufruf von `postinvoke()` wieder zu übergeben. Wenn die Implementierung des Servant-Locators den `Cookie` nicht benötigt, wird er einfach ignoriert. Der POA stellt der `postinvoke()` Operation zusätzlich den vorher ermittelten Servant bereit. Die weiteren Parameter entsprechen denen von `preinvoke()`.

Analog zur `incarnate()` Operation des `ServantActivator`-Interfaces kann die Implementierung von `preinvoke()` eine System-Exception oder eine **ForwardRequest** User-Exception werfen. Der Servant-Aufruf wird in diesem Fall nicht durchgeführt. Bevor aber eine Rückübertragung der Exception an den Client bzw. eine Rücksendung der Objektreferenz innerhalb des **LOCATION_FORWARD** Reply-Status stattfindet, muß in jedem Fall noch der Aufruf der `postinvoke()` Operation erfolgen. Im Gegensatz zu den Operationen des Servant-Activators, sind die Aufrufe von `preinvoke()` und `postinvoke()` immer paarweise auszuführen, da beide Operationen Teil der Anfragebearbeitung sind (im Unterschied zu `etherealize()`). Wenn innerhalb von `postinvoke()` irgendeine Exception auftritt, muß ein eventuell vorhandenes Ergebnis des Servant-Aufrufes zurückgesetzt werden, denn ein Servant-Aufruf ist nur als erfolgreich zu betrachten, wenn auch die `postinvoke()` Operation erfolgreich durchgeführt werden konnte. Der POA gibt für die Aufrufe auf einem Servant-Locator bestimmte Garantien, die für die Portabilität von Serveranwendungen von Bedeutung sind:

- Ein von `preinvoke()` zurückgelieferter Servant wird nur im Kontext dieser Anfrage für einen Servant-Aufruf benutzt. Der POA übergibt genau diesen Servant an den Aufruf von `postinvoke()`.
- Der Aufruf von `preinvoke()`, der Servant-Aufruf und der Aufruf von `postinvoke()` werden für eine einzelne Anfrage innerhalb eines Threads durchgeführt.
- Ein POA, der mit dem *ORB Controlled Thread Model* konfiguriert wurde und dessen Implementierung *multi-threaded* ist, muß keinesfalls Einschränkungen bezüglich der Nebenläufigkeit von `preinvoke()` und `postinvoke()` Operationen implementieren, da die vom Servant-Locator zur Verfügung gestellten Servants nicht zusätzlich vom POA in einer AOM verwaltet werden. Dies hat aber auch zur Folge, daß durch mehrere nebenläufige Aufrufe von `preinvoke()` mit der selben Objekt-ID zu einer Zeit durchaus mehrere Servants für ein Objekt existieren können.
- Wenn der POA mit dem *Single Thread Model* konfiguriert wurde, müssen alle Aufrufe auf einem Servant-Locator serialisiert werden. Dies trifft auch für einen Servant-Locator zu, der bei mehreren *single-threaded* POAs registriert wurde.

4.4.5. Default-Servant

Eine weitere Möglichkeit, einen Servant bei einem POA zu registrieren, besteht darin, einem POA einen Default-Servant zuzuweisen. Dieser wird immer dann für eine Anfrage verwendet, wenn der POA keinen anderen Servant für die Objekt-ID des Zielobjektes registriert hat, er also keinen Eintrag in der AOM findet (**RETAIN**) bzw. keine AOM verwendet (**NON_RETAIN**).

```
module PortableServer {
    interface POA {
        void    set_servant(in Servant default_servant)
                    raises(WrongPolicy);
        Servant get_servant()
                    raises(NoServant, WrongPolicy);
    };
};
```

Die POA-Schnittstelle stellt zwei Operationen bereit, um einen Servant als Default-Servant zu setzen oder einen registrierten Default-Servant zu erfragen. Beide Operationen erfordern den **USE_DEFAULT_SERVANT** Policy-Wert. Sollte dieser nicht gelten, wird eine **WrongPolicy** User-Exception geworfen. Die Operation `set_servant` darf nur einmal auf einer `POA`-Instanz aufgerufen werden. Jeder weitere Versuch hat eine **BAD_INV_ORDER** System-Exception zur Folge. Die `get_servant()` Operation wirft eine **NoServant** User-Exception, wenn vorher kein Default-Servant mit dem POA assoziiert wurde.

Jedes Objekt, das durch den Default-Servant inkarniert wird, muß dasselbe Interface bzw. ein Basis-Interface unterstützen. Insofern wird ein Default-Servant oft eingesetzt, wenn es in einer Serveranwendung mehrere Objekte mit der selben Schnittstelle gibt. Da ein Default-Servant mehrere dieser Objekte inkarniert, sollten diese keinen eigenen Zustand besitzen. Ein Servant hat aber auch die Möglichkeit, während der Bearbeitung einer Anfrage auf die Objekt-ID des Zielobjektes zuzugreifen, was zum einen die Implementierung eines objektspezifischen Verhaltens und zum anderen die Verwaltung eines objektspezifischen Zustandes in einer externen Datenbasis ermöglicht. Der Zugriff auf derartige Informationen des Aufrufkontextes wird im Abschnitt 4.7 beschrieben.

4.5. POA-Aktivierung

Ähnlich wie Servants können auch `POA`-Instanzen erst bei Bedarf erzeugt werden. Wenn vom ORB eine Anfrage an einen POA ausgeliefert werden soll, der noch nicht existiert, kann dieser durch einen Aufruf auf einem beim Eltern-POA registrierten Adapter-Activator erzeugt werden. Dem neu erzeugten POA kann dann die Anfrage übergeben werden. Wenn jedoch die Anfrage nicht für diesen, sondern für einen weiteren Kind-POA des eben erzeugten POAs bestimmt ist und der eben erzeugte POA wiederum mit einem Adapter-Activator initialisiert wurde, kann auch dieser benutzt werden, um einen weiteren Kind-POA zu erzeugen. Dieser Mechanismus läßt sich bis zu einer erfolgreichen Erzeugung des verantwortlichen POAs fortführen. Die `AdapterActivator`-Schnittstelle besitzt eine einzige Operation und ist ebenfalls im `PortableServer`-Modul definiert.

```
module PortableServer {
    interface AdapterActivator {
        boolean unknown_adapter(in POA parent, in string name);
    };
};
```

Die `unknown_adapter()` Operation bekommt den existierenden Eltern-POA und den Namen des neu zu erzeugenden Kind-POAs übergeben. Die Implementierung eines Adapter-Activators verwaltet Name-zu-POA-Beziehungen und erzeugt einen Kind-POA bei dem übergebenen Eltern-POA unter dem gewünschten Namen. Bei einer erfolgreichen Erzeugung und Initalisierung wird der Wert **TRUE** zurückgegeben. In diesem Fall kann der ORB mit der Auslieferung der Anfrage fortfahren, in der Annahme, daß der gerade erzeugte Kind-POA nun existiert. Während der Ausführung der Operation auftretende System-Exceptions werden vom ORB aufgefangen, und dem Client wird eine **OBJ_ADAPTER** System-Exception übermittelt. Diese weist auf eine fehlerhafte POA-Konfiguration hin. Wird jedoch **FALSE** zurückgegeben, wird dem Client die **OBJECT_NOT_EXIST** System-Exception zurückgeliefert. Die Gründe für die Zurückgabe des Wertes **FALSE** können verschiedener Natur sein. Letztendlich entscheidet die Server-Implementierung, ob ein CORBA-Objekt existiert. Die Erzeugung des verantwortlichen POAs wäre dafür die Voraussetzung.

Ein Adapter-Activator übernimmt üblicherweise auch die vollständige Initialisierung des neu erzeugten POAs, bsw. mit einem Adapter-Activator, einem Default-Servant, einem Servant-Manager oder mit der initialen Aktivierung einer Anzahl von Objekten. Da während des Aufrufes von `unknown_adapter()` alle Anfragen an den Kind-POA vom ORB gepuffert werden, ist die Verwendung eines Adapter-Activators zur Erzeugung benutzerkonfigurierter POAs auch eine Maßnahme, um *Race Conditions* bei der Erzeugung von Objekt-Adaptern mit gleichzeitig eintreffenden Anfragen zu vermeiden. Für die Initialisierung eines POAs mit einem Adapter-Activator stellt die `POA`-Schnittstelle eine Operation zum Setzen des `the_activator` POA-Attributes bereit.

Eine weitere Möglichkeit einen POA erst bei Bedarf zu erzeugen, bietet die `find_POA()` Operation der `POA`-Schnittstelle. Sie kann von einer Serveranwendung verwendet werden, um die Erzeugung eines Kind-POAs hinauszuschieben. Gleichzeitig liefert sie, aufgerufen auf einem Eltern-POA, einen bereits vorhanden Kind-POA mit dem übergebenen Adapternamen zurück.

```
module PortableServer {
    exception AdapterNonExistent {};
    interface POA {
        POA find_POA(in string adapter_name, in boolean activate_it)
                   raises(AdapterNonExistent);
    };
};
```

Wenn als `activate_it` Parameter der Wert **TRUE** übergeben wurde, so wird auf einem vorhandenen Adapter-Activator die `unknown_adapter()` Operation aufgerufen, falls kein Kind-POA unter diesem Namen registriert ist. Ist schon ein POA registriert, oder kann er mit Hilfe eines Adapter-Activators erfolgreich erzeugt werden, wird der ermittelte POA zurückgegeben. Anderfalls wird eine **AdapterNonExistent** User-Exception generiert.

Da die `create_POA()` Operation der `POA`-Schnittstelle die einzige Möglichkeit für eine Serveranwendung darstellt, eine neue POA-Instanz zu erzeugen, wird die vom Entwickler bereitgestellte Implementierung eines Adapter-Activators ebenso diese Operation benutzen müssen. Damit ist der Code für die Erzeugung eines POAs in dieser einen Operation gekapselt. Nebenläufige Aufrufe dieser Operationen, die eine POA-Erzeugung mit gleichem Adapternamen nach sich ziehen würden, sind durchaus möglich. Ob nun die ORB-Implementierung infolge einer nicht zustellbaren Anfrage die `unknown_adapter()` Operation aufruft, der Entwickler in einer *multi-threaded* Serveranwendung `find_POA()` bzw. `create_POA()` verwendet, oder diese Operationen innerhalb einer Objekt-Implementierung verwendet werden, in jedem Fall muß die Implementierung die potentiellen Fehler einer manuellen oder automatischen POA-Erzeugung behandeln. Speziell für den Implementierer der `unknown_adapter()` Operation bedeutet dies die geeignete Behandlung der möglichen Exceptions, die von der `create_POA()` Operation erzeugt werden können (siehe Abschnitt 4.3). Eine ebensolche Quelle für potentielle *Race Conditions* sind nebenläufige Zerstörungsprozesse eines POAs mit gleichem Namen. Auf diese wird im Abschnitt 4.12 eingegangen.

4.6. Anfragebearbeitung

Dieser Abschnitt soll den Ablauf vom Empfang einer Anfrage durch den POA bis zur Durchführung des Aufrufes auf einem Servant übersichtsartig zusammenstellen. Vieles von dem, was hier in einen Gesamtzusammenhang gebracht wird, wurde schon in den vorherigen Abschnitten erwähnt, um die einzelnen Policies und Systemkomponenten einzuführen. Die programmiersprachenspezifischen Schnittstellen für die Durchführung des Servant-Aufrufes werden an dieser Stelle nur angerissen und erst im Kapitel 5 für das Java-Mapping vertieft. Dieser Abschnitt dient also auch als Wegweiser in andere Teile dieser Arbeit.

Ermittlung des Servants

Die Vorgehensweise des POAs bei der Ermittlung des zuständigen Servants hängt von dem Policy-Wert für die `ServantRetentionPolicy` ab. Gilt der Wert **RETAIN** und der POA kann mit Hilfe der AOM einen aktiven Servant für diese Objekt-ID ermitteln, wird dieser für den Aufruf der Operation genutzt. Wenn kein Servant für diese Objekt-ID in der AOM registriert bzw. keine AOM verwendet wurde, wird der Wert für die `RequestProcessingPolicy` herangezogen. Diese kann drei verschiedene Werte besitzen, deren Konsequenzen im folgenden dargestellt werden.

- Gilt der Wert **USE_DEFAULT_SERVANT**, wird der Default-Servant zum Aufruf der Operation verwendet (siehe Abschnitt 4.4.5). Sollte aber festgestellt werden, daß kein Servant als Default-Servant beim POA registriert wurde, wird die Bearbeitung der Anfrage durch den POA beendet und dem Client eine **OBJ_ADAPTER** System-Exception übermittelt.

- Gilt der Wert **USE_SERVANT_MANAGER**, wird ein beim POA registrierter Servant-Manager herangezogen und auf diesem entweder die Operation `incarnate()` oder `preinvoke()` aufgerufen, um einen Servant zum Aufruf der Operation zu ermitteln (siehe Abschnitt 4.4.4). Sollte aber festgestellt werden, daß kein Servant-Manager beim POA registriert wurde, wird die Bearbeitung der Anfrage durch den POA beendet und dem Client eine **OBJ_ADAPTER** System-Exception übermittelt. Die Implementierung der `incarnate()` bzw. `preinvoke()` Operation kann alternativ zur Rückgabe eines Servants eine **ForwardRequest** User-Exception erzeugen. Auftretende System-Exceptions werden dem Client zurückgesendet. Falls kein Servant zurückgegeben werden konnte, ist die Bearbeitung der Anfrage durch den POA beendet. Es liegt in Verantwortung des Anwendungsentwicklers, Fehlerzustände bei der Inkarnation eines Servants zu behandeln und das beschriebene Verhalten dieser Operationen sicherzustellen.

- Gilt der Wert **USE_ACTIVE_OBJECT_MAP_ONLY** wird dem Client eine **OBJECT_NOT_EXIST** System-Exception gesendet. Die Bearbeitung der Anfrage ist damit für den POA beendet.

Aufruf auf dem Servant

Nach der Ermittlung des verantwortlichen Servants kann auf diesem der eigentliche Objekt-Aufruf durchgeführt werden. Der POA stellt die Ressourcen bereit, die für die weitere Bearbeitung benötigt werden, z.B. einen eigenen Thread als Ablaufumgebung für den Aufruf. Für den Aufruf auf dem Servant und das *Dispatching* der Anfrage auf die korrekte Servant-Operation definiert das jeweilige Language-Mapping eine Skeleton-Architektur. Im Java-Mapping werden zwei APIs definiert, die ein Servant unterstützen kann, ein stream-basiertes und ein auf dem *Dynamic Skeleton Interface* basierendes API. Beide APIs definieren eine einzelne Operation, die vom POA aufgerufen wird, um dem Servant die nötigen Anfrageparameter zu übergeben und die Ergebnisse der Ausführung des Aufrufes wieder in Empfang zu nehmen. Eine standardkonforme ORB-Implementierung muß beide möglichen APIs verwenden können. Der Servant kann sich seinerseits der ebenfalls im Language-Mapping definierten Schnittstellen zum *Unmarshaling* der Parameter, zum *Marshaling* und zur Rückgabe der Ergebnisse bedienen. Im Abschnitt 5.1 sind die Skeleton-Architektur und die erwähnten APIs ausführlich beschrieben.

4.7. Zugriff auf den Aufrufkontext

In den bisher beschriebenen Szenarien wird ein Servant einem POA als ein Callback-Objekt zur Verfügung gestellt, auf dem der POA den Objekt-Aufruf durchführen kann. Für eine Servant-Implementierung ist es aber häufig auch notwendig, den POA zu ermitteln, der den Aufruf auf dem Servant vermittelt hat, um Operationen der POA-Schnittstelle aufrufen zu können, z.B. für die Erzeugung von Objektreferenzen. Wenn ein Servant mehrere Objekte inkarniert, könnte es ein Anliegen sein, die Objekt-ID des Zielobjektes in Erfahrung zu bringen, um somit ein objektspezifisches Verhalten implementieren zu können. Dieser Ansatz wird häufig in Verbindung mit einem Default-Servant verwendet. Um diese Funktionalität zu realisieren, verwaltet die ORB-Implementierung den Kontext für jeden Servant-Aufruf, und die POA-Spezifikation definiert eine Möglichkeit, während der Ausführung eines Servant-Aufrufes auf Kontext-Informationen des eigenen Aufrufthreads zuzugreifen. Dafür steht dem Entwickler eine weitere Schnittstelle aus dem `PortableServer`-Modul zur Verfügung.

```
module PortableServer {
    interface Current : CORBA::Current {
        exception NoContext {};

        POA      get_POA()       raises (NoContext);
        ObjectId get_object_id() raises (NoContext);
    };
};
```

Wie jeder ORB- bzw. CORBA-Service, der einen eigenen Kontext bereitstellt, erweitert das `PortableServer::Current`-Interface das leere `CORBA::Current`-Interface. All diesen abgeleiteten `Current`-Interfaces ist gemeinsam, daß sie den Zugriff auf Informationen ermöglichen, die mit dem Thread verknüpft sind, in dem ihre Operationen aufgerufen werden. Bsw. besitzen der *Security Service* und der *Transaction Service* ein eigenes `Current`-Interface. Ein POA-Current-Objekt kann innerhalb einer Anwendung über die ORB-Initialisierungschnittstelle ermittelt werden (`resolve_initial_references("POACurrent")`).

POA-Current-Objekte sind sowohl *locality-* als auch *thread-constrained*. Ein `Current`-Objekt kann zwar von verschiedenen Threads einer Anwendung genutzt werden, es gestattet aber nur den Zugriff auf die Informationen, die mit dem aufrufenden Thread verknüpft sind. Natürlich ist der Zugriff auf ein `Current`-Objekt auch in *single-threaded* Umgebungen bzw. Anwendungen möglich. Ein `Current`-Objekt liefert mit dem Aufruf von `get_POA()` den verantwortlichen POA und mit dem Aufruf von `get_object_id()` die Objekt-ID, die das Zielobjekt eines Aufrufes identifiziert. Beide Operationen werfen eine **NoContext** User-Exception, falls ihr Aufruf nicht während eines vom POA vermittelten Aufrufes auf einem Servant erfolgt.

Zusätzlich zu der eben beschriebenen Möglichkeit des expliziten Zugriffs auf Kontext-Informationen ist deren Verwaltung durch die ORB-Implementierung auch für einige Operationen der `POA`- und `POAManager`-Schnittstelle von Bedeutung. Einige dieser Operationen weisen ein kontextabhänges Verhalten auf. Wenn ihr Aufruf im Kontext eines Servant-Aufrufes erfolgt, können diese Operationen die Informationen des Aufrufkontextes als zusätzliche Informationsquelle benutzen. Es stellt sich auch als hilfreich heraus, neben dem verantwortlichen POA und der Objekt-ID des Zielobjektes den Servant, auf dem der Aufruf stattfindet, innerhalb der Kontext-Informationen zu verwalten, um je nach Definition für den Begriff "Aufrufkontext" die nötigen Informationen bereitstellen zu können. Bsw. muß die POA-Operation `servant_to_reference()` im Kontext eines Aufrufes auf dem übergebenen Servant verwendet werden, damit die Kontext-Informationen herangezogen werden (siehe Abschnitt 4.9). In anderen Fällen, bsw. bei den Operationen des POA-Managers, reicht es aus, einen Aufrufkontext für einen beliebigen Servant innerhalb der vom POA-Manager verwalteten POA-Instanzen zu erkennen, um eine Verklemmungssituation zu vermeiden (siehe Abschnitt 4.11).

4.8. Lokale Objekte

In den bisherigen Ausführungen zum POA wurde immer davon ausgegangen, daß eine Anfrage an ein Objekt über ein Netzwerk vermittelt wird. Objektreferenzen können natürlich genausogut im selben Prozeß wie ihre Objekt-Implementierungen residieren. Im Abschnitt 2.3.5 wurde schon darauf hingeweisen, daß Aufrufe auf einem lokalen Objekt mit einer höheren Effizienz ablaufen können. Das Java-Mapping definiert dafür eine Stub-Architektur, die diese Optimierungen berücksichtigt. Die definierten Stub-Klassen enthalten Code, der speziell bei lokalen Aufrufen zur Ausführung kommt. Die dabei verwendeten APIs und der Ansatz, mit dem ein Servant an eine Objektreferenz vermittelt wird, damit diese den Aufruf direkt auf dem Servant durchführen kann, werden im Java-Mapping-Kapitel im Abschnitt 5.2 beschrieben. Trotz dieser Optimierung muß das "Prinzip der Lokalitätstransparenz" gewährleistet sein.

Die in diesem Kapitel beschriebene POA-Funktionalität gilt also auch für den Fall, daß sich ein Client im selben Prozeß befindet. Es darf also keinen Unterschied machen, ob ein Aufruf auf einem lokalen oder einem entfernten Objekt durchgeführt wird. Die verschiedenen POA-Zustände regeln den Anfragefluß, den eine Objekt-Implementierung erreicht, in genau der gleichen Weise, wie es für Fernaufrufe der Fall ist (siehe Abschnitt 4.11). Wenn für einen lokalen Aufruf noch kein Servant bei einem POA registriert wurde, kann ebenfalls ein Servant-Manager bemüht werden, um einen Servant bei Bedarf zu erzeugen. Diese Dinge seien nur als Beispiele genannt; bei der Beschreibung der POA-Funktionalität wird nicht gesondert darauf hingewiesen. Insbesondere sind die Informationen des Aufrufkontextes ebenso für lokale Aufrufe über das `Current`-Objekt verfügbar. Diese Vorgaben legen nahe, lokale und entfernte Anfragen in einer ORB- bzw. POA-Implementierung auf ähnliche Weise zu behandeln. Dies betrifft vor allem die Auslieferung einer Anfrage durch den ORB, die Abbildung der Anfrage innerhalb eines POAs auf einen Servant und die Verwaltung des Aufrufkontextes. Durch das jeweilige Language-Mapping muß sichergestellt werden, daß die Semantik der Parameter-Übergabe und Ergebnis-Rückgabe bei allen möglichen Parameter-Typen und Übergabe-Modi auch für lokale Aufrufe gewährleistet ist.

Entsprechend der Strategie der OMG, selbst Kernkomponenten konsequent in IDL zu definieren, handelt es sich bei den POA-bezogenen Schnittstellen ebenfalls um echte IDL-Interface-Definitionen. Ich möchte diesen Abschnitt auch nutzen, um die im POA-Kapitel angesprochenen Systemkomponenten nach ihren Objekt-Eigenschaften einzuordnen. Es lassen sich zwei verschiedene Gruppen von IDL-Interface-Definitionen ausmachen, für die sich das "Prinzip der Lokalitätstransparenz" nicht aufrechterhalten läßt.

4.8.1. Locality and Invocation Constrained Objects

Die POA-bezogenen Objekte werden überwiegend von der ORB- bzw. POA-Implementierung bereitgestellt. Diese Objekte einer konkreten Programmiersprache realisieren ein Framework, mit dem der Entwickler einer Serveranwendung die von ihm zur Verfügung gestellten Servants mit dem ORB verbinden kann. Der Lebenszyklus dieser Objekte ist gekoppelt an den Serverprozeß, eine Reaktivierung nach Beendigung des Serverprozesses ist nicht sinnvoll, genausowenig wie ein externer Zugriff auf diese Objekte. Damit sind die wesentlichen Vorzüge einer Implementierung als CORBA-Objekte verpufft. Außerdem implementieren diese Objekte gerade die serverseitige Kernfunktionalität, die für die Realisierung eines CORBA-Objektes benötigt wird. Damit sich "die Ratte nicht in den Schwanz beißt", müssen diese Objekte gesondert behandelt werden. Die Aktivierung eines POA-Objektes bei einem POA-Objekt wäre ein unlösbares Problem.

In früheren Versionen der CORBA-Spezifikation hätte man diese Objekte in PIDL definiert (wie bsw. den BOA). Nun ist man dazu übergegangen, diese Objekte mit echten IDL-Schnittstellen zu versehen, mit allen Konsequenzen für die zu implementierenden Schnittstellen einer CORBA-Objektreferenz. Für die Generierung der sprachspezifischen Schnittstellen und Klassen durch den IDL-Compiler werden somit die normalen Regeln des Language-Mappings verwendet. Um nun trotzdem eine effiziente Implementierung als Programmiersprachen-Objekt zu ermöglichen, werden im Spezifikationstext Einschränkungen für die Verwendung dieser Objektreferenzen und die Konsequenzen für eine Fehlanwendung definiert. Zu den Komponenten, die auf diese Weise spezifiziert sind, gehören neben dem **POA** alle **POA-Policies**, das **POA-Current** und der **POA-Manager**. In der Spezifikation findet sich, am Beispiel des POAs, folgender Text.

> A `POA` object must not be exported to other processes, or externalized with `ORB::object_to_string`. If any attempt is made to do so, the offending ope-

> ration will raise a **MARSHAL** system exception. An attempt to use a `POA` object with the DII may raise the **NO_IMPLEMENT** exception.

Eine Implementierung für die Basis-Schnittstellen eines CORBA-Objektes muß natürlich bereitgestellt werden. Diese hat aber in der definierten Weise zu erfolgen, die verhindert, daß eine derartige Objektreferenz die Prozeßgrenzen verläßt oder das DII für einen CORBA-Aufruf verwendet wird. Alle anderen Operationen der speziellen IDL-Schnittstelle können direkt, ohne den Umweg über einen Servant, implementiert werden. Ein wichtiger Effekt dieses Vorgehens ist, daß diese lokalen Programmiersprach-Objekte an den Stellen verwendet werden können, an denen eine CORBA-Objektreferenz verlangt wird. Damit lassen sich einheitliche Schnittstellen für den Zugriff auf lokale bzw. entfernte Dienste realisieren, wie bsw. `ORB::resolve_initial_reference()`.

4.8.2. Locality Constrained Objects

Die Implementierung eines Adapter-Activators oder eines Servant-Managers, und damit wären alle IDL-Schnittstellen des `PortableServer`-Moduls erwähnt, könnte auf den ersten Blick auf die zuvor geschilderte Art und Weise erfolgen. Die Verwendung dieser Objekte ist ebenso an einen Serverprozeß gebunden. Und doch gibt es einen feinen Unterschied. Ihre Implementierung wird durch den Anwendungsentwickler bereitgestellt. Der Entwickler muß diese Objekte in einer portablen Weise implementieren und erzeugen können. Im Gegensatz zu den im vorherigen Abschnitt beschriebenen Objekten gibt es keine Operationen der ORB- bzw. POA-Schnittstelle, die eine Erzeugung übernehmen. Damit ist eine direkte Implementierung als Objektreferenz ausgeschlossen, denn die Implementierung der Basis-Schnittstellen eines CORBA-Objektes müßte der Entwickler beisteuern. Frei in seiner Implementierung, wäre er nicht gezwungen, die in der Spezifikation angegebenen Einschränkungen hinsichtlich der Verwendung der Objekte zu berücksichtigen. Demzufolge sind diese in der Spezifikation auch wesentlich schwächer formuliert.

> A `AdapterActivator` object must be local to the process containing the POA objects it is registered with.

Diese Formulierung legt die Implementierung dieser Objekte als Servants und die Registrierung bei einem POA fest. Demnach muß eine Implementierung die generierten Skeleton-Klassen erweitern bzw. benutzen (siehe Abschnitt 5.1). Durch die Aktivierung dieser Objekte bei einem geeigneten POA (z.B. dem Root-POA) ist die Objektreferenz erhältlich, die dann in dem gewünschten POA, bsw. als `AdapterActivator`-Objekt, gesetzt werden kann. Somit sind alle Aufrufe auf diesen Objekten ganz normale, lokale CORBA-Aufrufe, die über den ORB vermittelt und an den zuständigen POA (z.B. den Root-POA) ausgeliefert werden, der dann seinerseits den Aufruf an die bereitgestellte Implementierung weiterleitet.

Die Lokalität dieser Objekte ist durch die spezifizierte Funktionalität sichergestellt, denn ihre Aufgabe ist gerade die Bereitstellung lokaler Komponenten, wie POA und Servant. Der teilweise Gebrauch von *native types* und *locality and invocation constrained objects* als Schnittstellen-Parameter verlangt ihre lokale Verwendung, denn ein *Marshaling* und *Unmarshaling* dieser Parameter-Typen ist nicht möglich (siehe auch Abschnitt 5.2).

Die geforderte Implementierung dieser Komponenten als lokale CORBA-Objekte ist ein deutlicher Hinweis auf das Anliegen der OMG, keine weiteren Pseudo-Objekte zu definieren. Mit dieser Lösung sind aber auch Nachteile hinsichtlich der Effizienz eines Aufrufes auf solch einem Objekt verbunden. Die Alternative wäre eine Definition innerhalb des Language-Mappings gewesen, mit den beschriebenen Nachteilen in puncto Klarheit und Vollständigkeit der Spezifikation, ob nun als Pseudo-Objekt oder als *native type.* Nach diesen eher generellen Ausführungen zur POA-Spezifikation möchte ich mich nun wieder der POA-Funktionalität zuwenden.

4.9. Konvertierungsoperationen

Objektreferenzen, Objekt-IDs und Servants sind die drei Basis-Entitäten, die ein POA verwaltet. Der POA stellt dem Anwendungsentwickler über seine IDL-Schnittstelle sechs Operationen zur Verfügung, die ihm die Abbildung eines Elements aus einer Menge auf ein Element einer anderen Menge ermöglichen. Es gibt somit für jede Entität zwei Operationen, denen ein Element aus dieser Menge übergeben werden kann und die ein Element einer anderen Entitätsmenge zurückliefert. Für eine Abbildung zwischen Objekt-IDs und Servants kann der POA auf seinen aktuellen Zustand, also auf die Menge der registrierten Objekt-ID-zu-Servant-Beziehungen zurückgreifen. Wenn er nicht in die Verwaltung dieser Assoziationen einbezogen wird, für ihn also weder der Policy-Wert **RETAIN** noch **USE_DEFAULT_SERVANT** gilt, kann er selbst keine dieser Abbildungen vornehmen, denn ein eventuell vorhandener Servant-Manager, der einen Teil bzw. die komplette Servant-Verwaltung eines POAs übernehmen kann, wird nicht einbezogen. Eine weitere Möglichkeit besteht in dem Zugriff auf den aktuellen Aufrufkontext, falls eine der Operationen während der Bearbeitung einer Anfrage auf einem Servant aufgerufen wird (siehe Abschnitt 4.7).

Auf einer anderen Ebene liegen die Abbildungen zwischen Objekt-IDs und Objektreferenzen. Eine Objekt-ID ist Teil einer Objektreferenz. Sie läßt sich somit aus ihr extrahieren. Andersherum kann eine Objektreferenz jederzeit aus einer Objekt-ID, angereichert mit weiteren Informationen, genannt sei hier nur die *Interface Repository Id,* erzeugt werden. Ob ein POA die einmal erzeugten Objektreferenzen verwaltet, um sie bei Bedarf wiederverwenden zu können, bleibt der ORB-Implementierung überlassen.

Die Operationen, die zwischen Objektreferenzen und Servants abbilden, müssen nun eine Kombination der bisher vorgestellten Wege durchführen. Sie beinhalten zum einen die Abbildung zwischen Objekt-ID und Objektreferenz in eine der beiden Richtungen und zum anderen die Ermittlung einer Objekt-ID oder eines Servants durch den Zugriff auf die AOM, den Default-Servant oder den Aufrufkontext.

Neben der Abbildungsfunktionalität können die Operationen, die eine Verbindung zwischen einer Objekt-ID und einem Servant voraussetzen und denen ein Servant als Parameter übergeben wird, bei geeigneter POA-Konfiguration die implizite Aktivierung eines neuen Objektes übernehmen (siehe Abschnitt 4.4.3). Immer dann, wenn der POA nicht mit den geforderten Policy-Werten konfiguriert ist, wird eine **WrongPolicy** User-Exception generiert. Wenn eine übergebene Objekt-ID bzw. Objektreferenz nicht vom POA abgebildet werden kann, da das Objekt nicht aktiv ist, wird eine **ObjectNotActive** User-Exception erzeugt. Ähnliches gilt für einen Servant. Ist dieser nicht aktiv bzw. kann er auch nicht implizit aktiviert werden, wird eine **ServantNotActive** User-Exception erzeugt. Wenn ein Default-Servant für die Abbildung genutzt werden soll, da für den POA der **USE_DEFAULT_SERVANT** Policy-Wert gilt, aber kein Default-Servant beim POA registriert wurde, handelt es sich um eine fehlerhafte POA-Konfiguration, die zu einer **WrongAdapter** User-Exception führt. In den Fällen, in denen eine Objekt-ID bzw. Objektreferenz übergeben wurde, die der der POA nicht selbst erzeugt hat, wird ebenso eine **WrongAdapter** User-Exception generiert.

```
module PortableServer {
    interface POA {
        ObjectId servant_to_id(in Servant servant)
                     raises(ServantNotActive, WrongPolicy);
        Object   servant_to_reference(in Servant servant)
                     raises(ServantNotActive, WrongPolicy);

        Servant  reference_to_servant(in Object reference)
                     raises(ObjectNotActive, WrongAdapter,
                            WrongPolicy);
        ObjectId reference_to_id(in Object reference)
                     raises(WrongAdapter, WrongPolicy);

        Servant  id_to_servant(in ObjectId oid)
                     raises(ObjectNotActive, WrongPolicy);
        Object   id_to_reference(in ObjectId oid)
                     raises(ObjectNotActive, WrongPolicy);
    };
};
```

Im folgenden werde ich die Abbildungsfunktionen einzeln beschreiben.

- Die Operation `servant_to_id` bildet einen Servant auf eine Objekt-ID ab. Voraussetzung ist der **RETAIN** Policy-Wert in Verbindung mit den Werten **UNIQUE_ID** oder **IMPLICIT_ACTIVATION**. Alternativ kann ein Default-Servant gesetzt sein. Ein **UNIQUE_ID** POA gibt die mit dem Servant assoziierte Objekt-ID zurück. Falls der Servant nicht aktiv ist oder der Wert **MULTIPLE_ID** gilt, wird bei gesetztem **IMPLICIT_ACTIVATION** Policy-Wert eine implizite Objektaktivierung vorgenommen. Dazu generiert der POA eine Objekt-ID, die er mit dem übergebenen Servant assoziiert. Die generierte Objekt-ID wird zurückgegeben. Falls die bisher aufgeführten Bedingungen nicht zutreffen, der übergebene Servant aber dem registrierten Default-Servant entspricht und die Operation im Kontext eines Aufrufes auf dem Default-Servant erfolgt ist, wird die Objekt-ID des Aufrufkontextes zurückgegeben.

- Die Operation `servant_to_reference` bildet einen Servant auf eine Objektreferenz ab. Voraussetzung ist der **RETAIN** Policy-Wert in Verbindung mit den Werten **UNIQUE_ID** oder **IMPLICIT_ACTIVATION**, sofern die Operation außerhalb eines Aufrufkontextes aufgerufen wurde. Wird die Operation aber während eines Aufrufes auf dem übergebenen Servant durchgeführt, wird die Objektreferenz zurückgegeben, die mit dem Aufrufkontext assoziiert ist. Ein **UNIQUE_ID** POA gibt die mit dem Servant assoziierte Objektreferenz zurück. Falls der Servant nicht aktiv ist oder der Wert **MULTIPLE_ID** gilt, wird bei gesetztem **IMPLICIT_ACTIVATION** Policy-Wert eine implizite Objektaktivierung vorgenommen. Dazu generiert der POA eine Objekt-ID, die er mit dem übergebenen Servant assoziiert. Eine dazu korrespondierende, neu erzeugte Objektreferenz wird zurückgegeben.

Sofern diese Operationen außerhalb des Kontextes eines Aufrufes verwendet werden, besitzen sie eine identische Funktionalität, wenn man von der zusätzlich vorzunehmenden Abbildung der Objekt-ID auf eine Objektreferenz während der `servant_to_reference()` Operation absieht. Werden diese Operationen aber innerhalb eines Aufrufkontextes verwendet, ändert sich dieses Bild dramatisch. Während sich die `servant_to_id()` Operation ausschließlich dazu eignet, innerhalb des Kontextes eines Aufrufes auf einem Default-Servant verwendet zu werden, läßt sich die Operation `servant_to_reference` universell einsetzen. Ihre Verwendung ist unabhängig von der konkreten POA-Konfiguration, sofern ihr Aufruf während der Bearbeitung einer Anfrage auf dem übergebenen Servant stattfindet. Ob es sich dabei um einen Default-Servant oder im Extremfall um einen von einem Servant-Locator für die Zeit eines Aufrufes bereitgestellten Servant handelt, ist unerheblich. Die `servant_to_reference()` Operation liefert eine Objektreferenz des Zielobjektes und erlaubt es einer Objekt-Implementierung, die Referenz des CORBA-Objektes zu ermitteln, das durch den Servant, zumindest für die Zeit der Bearbeitung der Anfrage, inkarniert wird. Eine ähnliche, aber nicht identische Funktionalität wird von der Servant-Operation `_this` bereitgestellt, die im Java-Mapping definiert ist. Im Abschnitt 5.1.4 werde ich auf diese Unterschiede gesondert hinweisen.

- Die Operation `reference_to_servant` bildet eine Objektreferenz auf einen Servant ab. Voraussetzung ist der **RETAIN** Policy-Wert oder die Konfiguration des POAs mit einem Default-Servant. Für die in der Objektreferenz enthaltene Objekt-ID wird der aktive Servant ermittelt. Sollte kein Servant unter dieser Objekt-ID in der AOM verzeichnet sein bzw. keine AOM verwendet werden, kann ein registrierter Default-Servant zurückgegeben werden.

- Die Operation `id_to_servant` bildet eine Objekt-ID auf einen Servant ab. Voraussetzung ist der **RETAIN** Policy-Wert oder die Konfiguration des POAs mit einem Default-Servant. Für die übergebene Objekt-ID wird der aktive Servant ermittelt. Sollte kein Servant unter dieser Objekt-ID in der AOM verzeichnet sein bzw. keine AOM verwendet werden, kann ein registrierter Default-Servant zurückgegeben werden.

Beide Operationen besitzen wiederum eine fast identische Funktionalität. Nur die Überprüfung, ob die übergebene Objektreferenz vom POA selbst erzeugt wurde, und die Extrahierung der Objekt-ID werden von der `reference_to_servant()` Operation zusätzlich durchgeführt. Der Entwickler kann mit Hilfe dieser Operationen wahlweise eine Objekt-ID oder eine Objektreferenz benutzen, um auf die aktive Inkarnation eines CORBA-Objektes zuzugreifen. Der Aufrufkontext spielt keine Rolle.

- Die Operation `reference_to_id` bildet eine Objektreferenz auf eine Objekt-ID ab. Die Verwendung der **WrongPolicy** User-Exception ist momentan nicht definiert. Ihre Angabe ist für künftige Erweiterungen der Spezifikation gedacht. Einzige Aufgabe ist die Überprüfung, ob die übergebene Objektreferenz durch den POA erzeugt wurde und die Extraktion und Rückgabe der Objekt-ID.

- Die Operation `id_to_reference` bildet eine Objekt-ID auf einen Objektreferenz ab. Voraussetzung ist der **RETAIN** Policy-Wert. Es wird überprüft, ob die Objekt-ID in der AOM verzeichnet ist. Ist das bezeichnete Objekt aktiv, wird eine korrespondierende Objektreferenz zurückgegeben.

Die Besonderheit der `reference_to_id()` Operation besteht darin, daß sie die Objekt-ID auch dann aus einer Objektreferenz extrahiert, wenn das bezeichnete Objekt nicht aktiv ist. Damit ist sie unabhängig von der POA-Konfiguration einsetzbar. Im Gegensatz dazu wird bei der `id_to_reference()` Operation auf eine bestehende Aktivierung des Objektes geachtet. Ein eventuell vorhandener Default-Servant hat keinen Einfluß. Für den Fall, daß ein POA mit dem Policy-Wert **USE_ACTIVE_OBJECT_MAP_ONLY** ausgestattet wurde und die Operation zum ersten Mal für eine bestimmte Objekt-ID zum Einsatz kommt, ist ihre Funktionalität identisch mit der eines `create_reference_with_id()` Aufrufes. Für beide Operationen spielt der Aufrufkontext keine Rolle.

4.10. Objektdeaktivierung

Im den Abschnitten 4.4.2 und 4.4.3 wurde beschrieben, wie ein Objekt bei einem POA aktiviert werden kann und daß die Aktivierung eines Objektes die Verwaltung einer Objekt-ID-zu-Servant-Beziehung durch den POA voraussetzt. Eine solche Verbindung kann auch wieder getrennt werden. Durch eine Objektdeaktivierung, also die Austragung eines Objekt-ID-Servant-Paares aus der AOM, können Anfragen an ein CORBA-Objekt nicht mehr an einen Servant vermittelt werden, sofern der verantwortliche POA nicht mit einem Servant-Manager bzw. einem Default-Servant konfiguriert wurde. Eine Serveranwendung kann Objekte, die nicht mehr benötigt werden und zerstört werden sollen, durch den Aufruf der `deactivate_object()` Operation auf dem verantwortlichen POA deaktivieren. Erneute Anfragen an ein deaktiviertes Objekt führen entweder zu einer **OBJECT_NOT_EXIST** System-Exception oder einer Reinkarnation durch einen Servant-Manager bzw. werden von einem eventuell vorhandenen Default-Servant bearbeitet.

```
module PortableServer {
    interface POA {
        void deactivate_object(in ObjectId oid)
                  raises(WrongPolicy, ObjectNotActive);
    };
};
```

Voraussetzung ist die Konfiguration des POAs mit dem **RETAIN** Policy-Wert, anderenfalls wird eine **WrongPolicy** User-Exception erzeugt. Sollte die übergebene Objekt-ID nicht in der AOM verzeichnet sein, wird eine **ObjectNotActive** User-Exception generiert. Die Austragung aus der AOM erfolgt erst, wenn alle aktiven Anfragen an das Objekt beendet sind. Solange dieser Zustand nicht erreicht ist, können weitere Anfragen an das Objekt gestartet werden. Dies führt bei einem fortlaufenden Strom von Anfragen zu dem Effekt, daß eine Objektdeaktivierung nicht wirksam werden kann. Der Grund für dieses Vorgehen liegt in der Ermöglichung rekursiver Aufrufe, die durch eine Objektdeaktivierung nicht unterbrochen werden sollen.

Nach der Austragung aus der AOM wird, wenn der POA mit einem Servant-Activator konfiguriert wurde, auf diesem die `etherealize()` Operation aufgerufen. Damit hat ein Servant-Activator die Möglichkeit, Aufräumungsarbeiten auf dem Servant durchzuführen, bsw. den Zustand eines persistenten Objektes in eine Datenbasis zu schreiben (siehe Abschnitt 4.4.4). Wenn ein Servant mehrere Objekte inkarniert, kann bei einer Deaktivierung dieser Objekte die `etherealize()` Operation für einen Servant mehrfach aufgerufen werden. Der Servant-Activator bekommt mit dem `remaining_activations` Flag als Operationsparameter angezeigt, ob es noch weitere Objekte innerhalb des POAs gibt, die mit diesem Servant assoziiert sind. Für Servants, die bei verschiedenen POAs aktiviert sind, können keine Garantien übernommen werden, da weder Objekt-Adapter noch Servant-Activators in irgendeiner Weise zusammenarbeiten. Somit liegt es in der Verantwortung des Entwicklers, eine Zerstörung eines Servants zu verhindern, der an anderer Stelle noch aktiviert ist.

Wenn während der Ausführung der `etherealize()` Operation eine Anfrage an das deaktivierte Objekt eintrifft, kann keine Assoziation mit einem Servant in der AOM gefunden werden, was ein Aufruf von `incarnate()` auf dem Servant-Activator nach sich zieht. Eine Reaktivierung des Objektes blockiert solange, bis die Etherealization des Servants abgeschlossen ist. Dies gilt für die Reinkarnation des Objektes durch den Servant-Activator ebenso wie für eine explizite bzw. implizite Objektaktivierung. Die `deactivate_object()` Operation kehrt sofort wieder zurück, ohne auf die Beendigung der Anfragen an das Objekt oder eine eventuelle Etherealization des Servants zu warten, um eine Verklemmungssituation auszuschliessen, bei der eine Objekt-Implementierung während der Ausführung einer Anfrage die Deaktivierung des Zielobjektes durchführen will.

Die Deaktivierung eines Objektes und eine spätere Reaktivierung mit Hilfe eines Servant-Activators ist ein übliches Verfahren, um bei Anwendungen mit vielen Objekten Speicherressourcen zu schonen. Die Anwendung sollte in diesem Fall garantieren, daß ein Anfrage im Zusammenspiel mit dem Servant-Activator immer ausgeliefert werden kann. Wurde aber ein Objekt deaktiviert und eine folgende Anfrage zog eine **OBJECT_NOT_EXIST** System-Exception nach sich, sollte damit der Lebenszyklus des Objektes endgültig beendet sein. Eine spätere Auferstehung des Objektes würde nur Verwirrung stiften und steht nicht im Einklang mit dem CORBA-Objekt-Modell.

4.11. Kontrolle des Anfrageflusses

Durch die Zuordnung von POA-Instanzen zu einem POA-Manager, können mehrere POAs und damit die von ihnen verwalteten Objekte gruppiert und der Strom von Anfragen, der die Objekt-Implementierungen erreicht, über die Schnittstelle des POA-Managers kontrolliert werden. Für die Kontrolle des *Request Flows* werden vier POA-Zustände eingeführt. Ein POA-Manager kapselt den Zustand der mit ihm assoziierten POAs. In den bisherigen Betrachtungen wurde stets angenommen, daß ein POA die Auslieferung einer Anfrage an einen Servant unverzüglich vornimmt, sobald er sie vom ORB übergeben bekommen hat. Dieses Verhalten wird mit dem POA-Zustand **ACTIVE** modelliert. Zur Synchronisation des POA-Initialisierungs- bzw. -zerstörungsprozesses, zur Realisierung von Load-Balancing-Verfahren und zum Schutz eines POAs vor einer Überflutung mit Anfragen werden drei weitere Zustände verwendet, in denen sich die mit einem POA-Manager assoziierten POAs befinden können.

HOLDING Es werden keine Anfragen an Servants ausgeliefert. Eingehende Anfragen werden gepuffert. Nach dem Verlassen des Zustandes werden die gepufferten Anfragen entsprechend dem neuen Zustand bearbeitet. Diesen Zustand kann eine Objekt-Implementierung nutzen, um einen temporären Fehlerzustand anzuzeigen bzw. um sich vor einer Überflutung mit Anfragen zu schützen. Er wird aber auch für Synchronisationszwecke verwendet, um Verwaltungs- bzw. Initialisierungsaufgaben ungestört durchführen zu können. Wenn das von einer ORB-Implementierung festgelegte Ressourcenlimit erreicht ist, wird für alle neu eintreffenden Anfragen eine **TRANSIENT** System-Exception an den Client verschickt, die eine temporäre Unerreichbarkeit des Objektes signalisiert. Der Client kann seine Anfrage zu einem späteren Zeitpunkt wiederholen.

ACTIVE Die Bearbeitung gepufferter bzw. eingehender Anfragen durch einen POA wird fortgesetzt. Auch in diesem Zustand kann eine Pufferung von Anfragen, die ein POA nicht sofort verarbeiten kann, nötig sein. Die Anzahl der Anfragen, die gepuffert werden können, ist von der ORB-Implementierung abhängig. Ist das Ressourcen-Limit erreicht, wird ebenfalls eine **TRANSIENT** System-Exception zurückgegeben.

DISCARDING Alle Anfragen werden vom POA mit einer **TRANSIENT** System Exception zurückgewiesen. Diesen Zustand kann eine Objekt-Implementierung ebenfalls für die Kontrolle des *Request Flows* nutzen. Im Gegensatz zum Zustand **HOLDING** wird auf eine Pufferung verzichtet, alle eingehenden Anfragen werden sofort zurückgewiesen. Damit eignet sich dieser Zustand eher für zeitkritische Anwendungen, bei denen die verzögerte Auslieferung im Zustand **HOLDING** unakzeptabel ist. Der Client kann seine Anfrage zu einem späteren Zeitpunkt wiederholen, da erwartet wird, daß der POA nach der Behebung des Fehlers wieder in den Zustand **ACTIVE** wechselt.

INACTIVE Dieser Zustand wird verwendet, um einen POA herunterzufahren. Im Gegensatz zu den Zuständen **HOLDING** und **DISCARDING** ist er nicht temporär, es wird davon ausgegangen, daß ein POA daraufhin zerstört wird. Ein POA hat damit die Möglichkeit, in Zusammenarbeit mit einem Servant-Activator eine Etherealization aller aktivierten Objekte durchzuführen, also bsw. ihren persistenten Zustand zu speichern. Alle eingehenden Anfragen werden vom POA zurückgewiesen. Der Mechanismus zur Zurückweisung der Anfragen bleibt der ORB-Implementierung überlassen. Es gibt z.B. die Möglichkeit, den *GIOP Location Forward Mechanismus* oder eine *Close Connection Message* einzusetzen. Für Anfragen lokaler Objekte wird eine **OBJ_ADAPTER** System-Exception geworfen. Ein POA, der sich im Zustand **INACTIVE** befindet, muß erst zerstört werden, um ihn unter gleichem Namen in einer Serveranwendung neu erzeugen zu können (siehe Abschnitt 4.3 und 4.12).

Wenn ein POA-Manager neu erzeugt wird, befindet er sich im Zustand **HOLDING**. Damit werden alle Anfragen an einen assoziierten POA von diesem in einer Queue gepuffert. Für den Start der Anfragebearbeitung muß der POA-Manager explizit in den Zustand **ACTIVE** versetzt werden. Für Zustandsänderungen stellt die `POAManager`-Schnittstelle vier Operationen bereit. Eine weitere Operation dient der Abfrage des aktuellen Zustandes. Das `POAManager`-Interface ist im Modul `PortableServer` definiert.

```
module PortableServer {
    interface POAManager {
        exception AdapterInactive {};
        enum State { ACTIVE, HOLDING, DISCARDING, INACTIVE };
        State get_state();

        void  activate() raises (AdapterInactive);
        void  hold_requests(in boolean wait_for_completion)
                          raises (AdapterInactive);
        void  discard_requests(in boolean wait_for_completion)
                          raises (AdapterInactive);
        void  deactivate(in boolean etherealize_objects,
                          in boolean wait_for_completion)
                          raises (AdapterInactive);
};
```

Für jede Zustandsänderung steht eine Operation zur Verfügung. Der Wechsel zwischen den Zuständen **HOLDING**, **ACTIVE** und **DISCARDING** ist beliebig oft und in beliebiger Reihenfolge möglich. Ein mehrmaliger Aufruf der gleichen Operation hat dabei keinen Effekt. Der Aufruf der `deactivate()` Operation versetzt den POA-Manager und damit alle assoziierten POAs in den Zustand **INACTIVE**. Dieser ist endgültig und kann nicht mehr verlassen werden. Ein anschliessender Aufruf einer Zustandswechseloperation hat eine **AdapterInactive** User-Exception zur Folge.

Die Operation `activate` kehrt nach dem Zustandswechsel sofort wieder zurück. Alle anderen Zustandswechseloperationen können so parametrisiert werden, daß sie erst zurückkehren, wenn alle aktiven Anfragen innerhalb aller assoziierten POAs beendet sind. Dafür muß der `wait_for_completion` Parameter auf den Wert **TRUE** gesetzt sein. Wird eine solche Operation bei gesetztem `wait_for_completion` Flag innerhalb des Kontextes eines Servant-Aufrufes verwendet, wird, um eine Verklemmungssituation zu vermeiden, eine **BAD_INV_ORDER** System-Exception geworfen und der Zustand nicht geändert. Das Warten der Operationen `hold_requests` und `discard_requests` auf die Beendigung der aktiven Anfragen wird durch den Aufruf einer beliebigen anderen Zustandswechseloperation innerhalb eines nebenläufigen Threads beendet.

Die Operation `deactivate` besitzt einen weiteren Parameter, mit dem die Durchführung der Etherealization aller aktiven Objekte innerhalb der assoziierten POAs

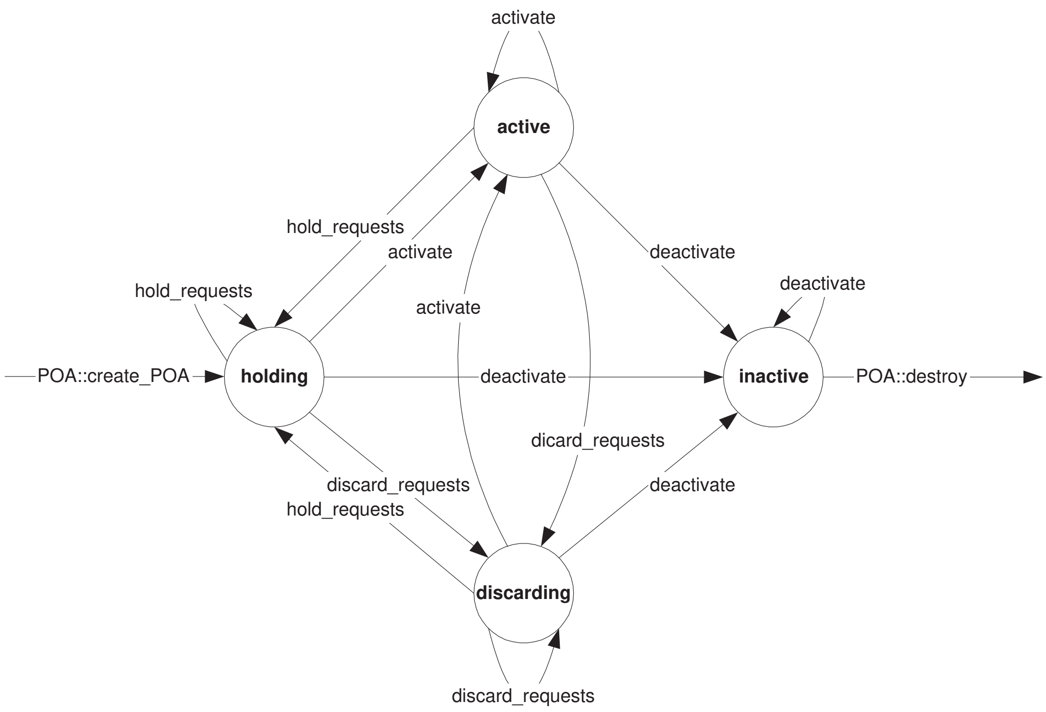


Abbildung 4.4.: POA-Manager-Zustände und Zustandswechseloperationen

bestimmt werden kann. Bei gesetztem `etherealize_objects` Flag wird die `etherealize()` Operation auf einem vorhandenen Servant-Activator für jedes aktivierte Objekt aufgerufen (siehe Abschnitt 4.4.4). Die Etherealization kann erst im Anschluß an die Beendigung aller aktiven Anfragen durchgeführt werden, um keine aktiven Anfragen durch die Zerstörung von Servants in Mitleidenschaft zu ziehen. Wenn eine Etherealization durchgeführt werden muß, kehrt die Operation bei gesetztem `wait_for_completion` Flag erst nach Fertigstellung der Objekt-Etherealization zurück.

Befindet sich ein POA im Zustand **ACTIVE**, so liegt es in der Verantwortung des POAs, eingehende Anfragen an einen Servant zu vermitteln. Mit der Ausnutzung der POA-Zustände **HOLDING** und **DISCARDING** kann eine Objekt-Implementierung die Steuerung des *Request Flows* übernehmen. Der POA dient in diesem Fall als Schutzschild für eine Objekt-Implementierung, bis sie für die Bearbeitung neuer Anfragen bereit ist. Um den *Request Flow* für einzelne Objekte zu steuern, muß deren Zordnung zu `POA`-Instanzen und die Einrichtung der Assoziationen mit einem POA-Manager gezielt vorgenommen werden, da die Zustandswechseloperationen über die `POAManager`-Schnittstelle erfolgen. Die Assoziation eines POAs mit einer exklusiven `POAManager`-Instanz kann der Entwickler bei der Erzeugung eines POAs erzwingen (siehe Abschnitt 4.4.4).

Die Möglichkeit, die normale Auslieferungstätigkeit eines POAs zu unterbinden, ist auch der Schlüssel für die Lösung eines Synchronisationsproblems, das während der POA-Initialisierungsphase auftreten kann. Die Assoziation eines POAs mit einem POA-Manager, der sich schon im Zustand **ACTIVE** befindet, schaltet die

Anfragebearbeitung durch diesen POA von Beginn an frei. Ist dieser POA mit dem **USE_SERVANT_MANAGER** Policy-Wert konfiguriert und trifft eine Anfrage ein, obwohl noch kein Servant-Activator mit dem POA assoziiert ist, handelt es sich um eine fehlerhafte POA-Konfiguration, die eine **OBJ_ADAPTER** System-Exception zur Folge hat. Um diese *Race Conditions* zu vermeiden, wurde der Initialzustand eines POA-Managers auf **HOLDING** festgelegt. Der Entwickler kann die nötigen Initalisierungsschritte nacheinander durchführen. Erst danach sollte die Auslieferungstätigkeit des POAs durch die Aktivierung des POA-Managers freigeschaltet werden. Bei der Verwendung eines POA-Managers für mehrere `POA`-Instanzen kann der Entwickler einen schon aktivierten POA-Manager für die Zeit einer POA-Erzeugung kurzzeitig in den Zustand **HOLDING** versetzen. Den gleichen Synchronisationseffekt kann der Entwickler durch die Verwendung eines Adapter-Activators erreichen (siehe Abschnitt 4.5).

Ein vorhandener Adapter-Activator wird **nicht** aufgerufen, wenn sich der existierende POA in den Zuständen **HOLDING**, **DISCARDING** oder **INACTIVE** befindet. In diesen Fällen wird mit einer Anfrage an einen der noch nicht existierenden Kind-POAs genauso verfahren, wie im vorherigen Abschnitt beschrieben. Im Zustand **HOLDING** wird eine Anfrage vom elterlichen POA solange gepuffert, bis ein Zustandswechsel eingetreten ist. Nach der Wiederaufnahme der Anfragebearbeitung kann je nach Folgezustand die weitere Auslieferung der Anfrage erneut versucht bzw. die Anfrage zurückgewiesen werden. Die Erzeugung eines POAs bei Bedarf setzt also die Aktivierung aller übergeordneten POAs voraus.

4.12. POA-Zerstörung

Die Zerstörung eines POAs wird normalerweise bei einem bevorstehenden ORB-Shutdown durchgeführt (`ORB::shutdown()`). Aber nicht nur in den Fällen, in denen die Serveranwendung beendet werden soll, macht die Zerstörung eines POAs Sinn. Ähnlich wie man Objekte deaktiviert, um sie bei einem späteren Gebrauch mit einem Servant-Activator wieder aktivieren zu können, ist es auch denkbar, einen POA herunterzufahren und ihn zu zerstören, um Ressourcen zu schonen und ihn bei späteren Anfragen wieder mit Hilfe eines Adapter-Activators zu aktivieren. Die Zerstörung eines POAs schließt die Zerstörung aller untergeordneten Kind-POAs mit ein. Die `POA`-Schnittstelle stellt dafür die `destroy()` Operation zur Verfügung.

```
module PortableServer {
    interface POA {
        void destroy(in boolean etherealize_objects,
                     in boolean wait_for_completion);
    };
};
```

Zuerst werden alle untergeordneten Kind-POAs zerstört, also deren `destroy()` Operation aufgerufen. Nach Abschluß der vollständigen Zerstörung aller Kind-POAs wird gewartet, bis alle aktiven, von diesem POA vermittelten Anfragen zum Ende gekommen sind. Erst danach wird die Zerstörung eines POAs nach außen "sichtbar". Für neu eintreffende Anfragen kann nun vom ORB ein eventuell vorhandener Adapter-Activator des Eltern-POAs aufgerufen werden, um einen POA unter gleichem Namen erneut zu erzeugen. Im Gegensatz dazu ist eine Reaktivierung im POA-Zustand **INACTIVE** nicht möglich (siehe Abbildung 4.4). Ein inaktiver POA muß erst zerstört werden, um ihn unter gleichem Namen, ob manuell oder automatisch, erneut erzeugen zu können (siehe Abschnitt 4.3 und 4.5). Die POA-Erzeugungsoperation blockiert aber solange, bis die POA-Zerstörung komplett abgeschlossen ist.

Die Markierung des POAs als "zerstört" führt dazu, daß alle Anfragen an diesen POA, die nicht schon durch den Aufruf eines Adapter-Activators des Eltern-POAs vom ORB gepuffert werden oder sich noch unbearbeitet in der Queue befinden, mit einer **OBJECT_NOT_EXIST** System-Exception zurückgewiesen werden. Alle Aufrufe auf der `POA`-Schnittstelle, abgesehen von der `destroy()` Operation selbst, haben ebenfalls eine **OBJECT_NOT_EXIST** System-Exception zur Folge. Damit ist der POA von außen nicht mehr ansprechbar, alle aktiven Anfragen sind beendet, und er kann einige Aufräumungsarbeiten "in Ruhe" vornehmen. Ähnlich wie beim Zustandswechsel zu **INACTIVE** kann mit dem `etherealize_objects` Parameter festgelegt werden, ob eine Objekt-Etherealization für jede in der AOM enthaltene Objekt-ID durchgeführt werden soll. Dieser Parameter ist natürlich nur relevant, wenn der POA auch mit einem Servant-Activator konfiguriert wurde (siehe Abschnitt 4.4.4). An dieser Stelle sei erinnert, daß bei der Zerstörung der Servants keine Aufrufe auf der `POA`-Schnittstelle mehr möglich sind, da der POA offiziell schon zerstört ist. Der POA kann nun seinerseits nicht mehr benötigte Systemressourcen freigeben (z.B. die Leerung eines Thread-Pools).

Mit dem `wait_for_completion` Parameter kann der Entwickler festlegen, ob er auf die Beendigung aller aktiven Anfragen und eine eventuelle Objekt-Etherealization warten möchte, oder ob die Operation gleich nach der Zerstörung der untergeordneten POAs und der Markierung des POAs als "zerstört" zurückkehren soll. Die `destroy()` Operation ist die einzige POA-Operation, die auch nach der sichtbaren Zerstörung des POAs aufgerufen werden kann. Damit haben mehrere Threads einer Anwendung die Möglichkeit, auf die Zerstörung eines POAs gemeinsam zu warten. Aber nur der Wert des `etherealize_objects` Parameters des ersten Aufrufes ist relevant für die Festlegung, ob eine Etherealization durchgeführt werden soll. Bisher unerwähnt blieb, daß bei einem Wert **TRUE** des `wait_for_completion` Parameters gleich zu Beginn der `destroy()` Operation geprüft wird, ob sie im Kontext eines Aufrufes auf einem beliebigen Servant dieses ORBs verwendet wurde. Wenn ja, wird, um eine Verklemmungssituation zu vermeiden, eine **BAD_INV_ORDER** System-Exception geworfen.

Die rekursive Zerstörung einer ganzen POA-Hierarchie, von der Blättern bis zur Wurzel des Baumes, kann Probleme aufwerfen, wenn die POA-Hierachie nicht sorgsam entwickelt wurde. Es sollte immer darauf geachtet werden, daß Objekte die für die Funktionsweise von POA-Instanzen benötigt werden, wie z.B. ein Servant-Manager oder ein Adapter-Activator bei POA-Instanzen aktiviert werden, die in der Hierarchie höher angesiedelt sind. Damit wird eine versehentliche Deaktivierung eines solchen Objektes ausgeschlossen.

4.13. Der Lebenszyklus eines CORBA-Objektes

Nach der Beschreibung der POA-Funktionalität möchte ich die Diskussion um die Existenz eines CORBA-Objektes noch einmal aufgreifen. Ein CORBA-Objekt existiert natürlich in keiner sichtbaren Form, es ist eine virtuelle Entität, eine Abstraktion bzw. ein Konzept eines Objektes. Erst die Inkarnation eines CORBA-Objektes durch einen Servant für die Dauer einer Anfrage verleiht dieser Abstraktion die Substanz.

Ein CORBA-Objekt kann existieren, auch wenn es keine Assoziation zwischen der Objektreferenz und einem Servant in der AOM gibt. Selbst wenn der Serverprozeß und mit ihm der POA zerstört wurde, können persistente Objekte weiterhin existieren. Eine Entscheidung, ob ein CORBA-Objekt existiert, wird bei jedem Objektaufruf neu gefällt. Wenn der Aufruf erfolgreich war, dann existiert das Objekt, wenn eine **OBJECT_NOT_EXIST** System-Exception gesendet wurde, dann nicht. Die Unterscheidung zwischen einem aktivierten und einem deaktivierten Objekt ist für den Client nicht relevant. Egal ob ein Adapter-Activator erst den erforderlichen POA erzeugt, oder ob ein Servant-Manager den Servant bereitstellt, da der POA keinen persistenten Zustand besitzt, liegt die Entscheidung, ob ein CORBA-Objekt zu einem bestimmten Zeitpunkt existiert, letztendlich beim Anwendungsentwickler, denn die Anwendung stellt die Inkarnation eines Objektes durch einen Servant zur Verfügung.

Ein Server kontrolliert nur die Erzeugung einer Objektreferenz, aber nicht deren Zerstörung. Wenn erst einmal eine Objektreferenz erzeugt und an den Client weitergegeben wurde, ist sie außerhalb der Kontrolle des Servers. Dies zeigt deutlich, das auch Objektreferenzen einen Lebenszyklus besitzen, der sich von dem Lebenszyklus des CORBA-Objektes unterscheidet, das sie referenzieren. Wenn sich ein Server bzw. der Anwender entschieden hat, den Lebenzyklus des CORBA-Objektes zu beenden, können Objektreferenzen zwar weiterhin existieren, sie besitzen aber keinen Wert mehr. Die Zerstörung eines transienten Objektes geschieht gewöhnlich durch den Aufruf der `destroy()` Operation auf einer POA-Instanz. Es ist natürlich auch denkbar, daß ein Server noch innerhalb seiner Lebenszeit entscheidet, ein transientes Objekt dauerhaft zu deaktivieren, und damit zu zerstören. Im persistenten Fall ist es gewöhnlich der Anwender, der sich entscheidet, ein persistentes Objekt nicht mehr zur Verfügung zu stellen.

4.14. Evaluation

Der größte Unterschied zwischen dem BOA und dem POA besteht darin, daß der BOA sein Hauptaugenmerk auf die Verwaltung von Serverprozessen legt, wohingegen beim POA dieses Thema vollständig ausgespart wurde. Das Konzept persistenter Objekte wird zwar vom POA unterstützt, die Aktivierung von Serverprozessen ist aber an andere ORB-Dienste delegiert (Activator-Daemon, ImR). Der POA selbst leistet seinen Beitrag durch die Generierung passender Objekt-Keys. Außerdem besitzt der Entwickler mit der Verwendung eines Servant-Managers eine flexible Möglichkeit, den persistenten Zustand von Objekten vor der Aktivierung des Servants zu laden und bei einer Deaktivierung auf einen Hintergrundspeicher zu schreiben.

Der POA fokussiert die Verwaltung von Servants. Die dynamischen Fähigkeiten des POAs, Servants erst bei Bedarf bereitzustellen, übertreffen die Möglichkeiten des BOAs (spätes Binden). Der Entwickler kann sehr flexibel mit der Bereitstellung eines Servant-Managers auf die Servant-Verwaltung Einfluß nehmen. Typische POA-Konfiguration wurden im Abschnitt 4.2.6 dargestellt.

Im Unterschied zum BOA ist der POA keine einzelne Komponente. Innerhalb eines Servers können mehrere POA-Instanzen in einer POA-Hierarchie existieren. Die dynamischen Möglichkeiten einer Serveranwendung bleiben nicht nur auf einen einzelnen POA beschränkt, mit dem Konzept des Adapter-Activators lassen sich POA-Instanzen mit den von ihnen "betreuten" Objekten dynamisch verwalten.

Der Anfragefluß, der die Objekt-Implementierungen erreicht, kann durch das Konzept des POA-Managers, der POA-Zustände und durch eine Gruppierung von POA-Instanzen sehr flexibel gesteuert werden. Durch die Möglichkeit der Pufferung von Anfragen (Zustand **HOLDING** bzw. bei der POA-Erzeugung durch einen Adapter-Activator) lassen sich Fehler im Zusammenhang mit potentiell auftretenden *race conditions* bei Aktivierungs- bzw. Deaktivierungsprozessen und gleichzeitig eintreffenden Anfragen wirksam vermeiden.

Ein häufig genannter Kritikpunkt am POA ist der, daß die einstellbare Thread-Policy nur die Unterscheidung zwischen einem *single-threaded* POA und einem vom ORB kontrollierten Thread-Management erlaubt. Letzteres läuft üblicherweise auf die generell nebenläufige Abarbeitung von Anwendungscode (Servant, Servant-Manager, Adapter-Activator) innerhalb von Threads hinaus.[2] Für eine feinkörnige Konfiguration der Threading-Eigenschaften muß der Entwickler die Objekt-Implementierungen, die nicht thread-sicher sind, bei einem *single-threaded* POA gruppieren (evtl. entgegen anderen Entwurfskriterien). Der POA könnte eine Möglichkeit bieten, ähnlich einem "einsteckbaren" Servant-Manager, das Thread-Management von einem Thread-Manager übernehmen zu lassen. Der Entwickler könnte somit ein optimal auf die Erfordernisse der Anwendung eingestelltes Verfahren für die Zuweisung von Threads implementieren. Die Thread-Policy bekäme bsw. einen zusätzlichen Wert **USE_THREAD_MANAGER**, und der Entwickler müßte ein spezifiziertes Thread-Manager-Interface implementieren und dem POA zur Verfügung stellen. Auf diesem Weg liessen sich ausgefeilte Thread-Management-Strategien auf jeder ORB-Umgebung verfügbar machen.

Um die Kritik am POA zu vervollständigen, sei noch auf die Realisierung des Servant-Managers und Adapter-Activators als echte (lokale) CORBA-Objekte hingewiesen. Die Definition dieser Schnittstellen in IDL ist zwar konsequent, bringt aber Probleme bei der Generierung von Stubs/Skeletons für diese Schnittstellen, da sie als Parametertypen *native types* verwenden, für die kein fehlerfreier Marshaling-Code generiert werden kann. Diese Probleme hätten vermieden werden können, wenn diese Komponenten selbst als *native types* im Language-Mapping definiert worden wären. Außerdem muß man bedenken, daß der Entwickler einen Servant für diese Komponenten bei einem anderen POA aktivieren muß, um eine lokale Objektreferenz zu erzeugen, die er bei einem POA setzen kann. Jeder Aufruf auf diesen Komponenten ist also ein vom ORB vermittelter (lokaler) Objektaufruf.

Aufrufe auf lokalen Objekten werden von einem ORB der POA-Generation in

[2] mit den spezifizierten Einschränkungen der nebenläufigen Ausführung der Operationen des Servant-Managers

der Regel optimiert durchgeführt. Letzendlich muß die ORB-Implementierung diese Optimierung unterstützen. Im Blick auf das aktuelle Java-Mapping sind die Voraussetzungen durch eine entsprechend vorbereitete Skeleton-Architektur gegeben (siehe Abschnitt 5.1).

Der POA unterstützt das Konzept der Anfrageumleitung. Eine Implementierung eines Servant-Managers kann auf portable Weise den Location-Forwarding-Mechanismus benutzen und somit eine Objektreferenz an eine anderes Objekt binden.

Der Plan der OMG, einen "alleskönnenden" Standard-Adapter zu definieren, scheint im Großen und Ganzen aufgegangen zu sein, wenn man von den immer vorhandenen Wünschen nach noch mehr Konfigurierbarkeit absieht. Die POA-Spezifikation läßt einem ORB-Implementierer keine Spielräume für Interpretationen, die die Portabilität von Serveranwendungen einschränken könnten. In diesem Sinne hat die POA-Spezifikation ihr Soll erfüllt. Die Entlastung des POAs von der Verantwortlichkeit hinsichtlich der Server-Aktivierung kam dem Anspruch auf Vollständigkeit der Spezifikation sicherlich entgegen.

Dieser "Mangel" an Spezifikation sollte die Portabilität von Server-Code aber in keiner Weise einschränken, denn die administrativen Schnittstellen einer ORB-Implementierung zur Konfiguration des Server-Aktivierungsmechanismus brauchen nicht im Server–Code verwendet werden. Diese werden sich dann natürlich, je nach ORB-Implementierung, unterscheiden. Wenn man unterstellt, daß ORB-Hersteller auf proprietäre Erweiterungen des Standards verzichten, wie bsw. einen expliziten Aufruf zur Server-Registrierung im ImR, sollten POA-basierte Serverprogramme portabel sein.

Ob die OMG die POA-Spezifikation in Zukunft erweitert, ist schwierig einzuschätzen. Im Prinzip besitzt sie einen entgültigen Charakter. Die Einführung einer neuen Policy ist nicht ohne weiteres möglich, um nicht die Portabilität von Serverprogrammen zu verlieren, die eine neu definierte Policy einsetzen wollen. Durch die Verwendung der generischen Struktur bei der POA-Erzeugung ist zumindest eine Voraussetzung dafür erfüllt. Eine POA-Implementierung, die sich der Existenz dieser Policy nicht bewußt ist, würde aber eine Exception bei der POA-Erzeugung generieren. Selbst wenn man unterstellt, daß ein POA eine ihm unbekannten Policy ignoriert, hängt es stark von der Semantik der neuen Policy ab, ob sie das Verhalten des POAs in einer Weise beeinflußt, die eine Wechselwirkung mit den anderen Konzepten des POAs beinhaltet. Im Falle einer orthogonal zu den anderen Einstellungen des POAs wirkenen Policy, wie bsw. einer Thread-Management-Policy, wäre eine nachträgliche Definition denkbar.

Die Serveranwendungsentwicklung ist mit der Einführung des POAs komplexer geworden. Eine Anwendung, die alle Möglichkeiten des POAs nutzen möchte, muß als verteilte Anwendung entworfen und entwickelt werden. Verteilungstransparenz auf der Serverseite, wie sie noch in verschiedenen BOA-Implementierungen erreicht wurde (durch proprietäre Erweiterungen), ist mit dem POA nicht in dem Maße möglich. Der POA versucht an dieser Stelle einen Kompromiß zu finden, zwischen einer einfachen Handhabung bei einfachen Anwendungsbeispielen und der Möglichkeit, komplexe Serverkonfigurationen zu realisieren.

5. Java Mapping

Dieses Kapitel ist als Ergänzung zum vorherigen Kapitel gedacht und soll die Java-spezifischen Aspekte bei der Integration des POAs in einen ORB und der Entwicklung von Serveranwendungen beschreiben. Das *Java Mapping* definiert neben der Abbildung der IDL-Sprachkonstrukte und aller IDL-Parametertypen eine Stub- und Skeleton-Architektur für portable Java-Client- bzw. Serveranwendungen. Die Anforderungen an Portabilität und Interoperabilität generierter Stub- und Skeleton-Klassen sind bei der Verwendung der Programmiersprache Java von besonderer Bedeutung. Da Java-Klassen über das Netz geladen werden können, sollte ihr Einsatz mit einer anderen Java-ORB-Implementierung als mit der, deren IDL-Compiler für ihre Generierung genutzt wurde, möglich sein. Ein zusätzliches Laden der kompletten ORB-Runtime-Umgebung wäre damit vermeidbar.

Das Java-Mapping enthält ein API von Java-Klassen und -Interfaces, die in den Paketen `org.omg.CORBA.portable` und `org.omg.PortableServer.portable` zusammengefaßt sind und von einer CORBA-konformen ORB-Implementierung verwendet werden. Genaugenommen sind dort zwei verschiedene APIs für Stubs/Skeletons definiert, eines basiert auf dem DII/DSI, das andere verwendet eine von der Programmiersprache Java her bestens bekannte stream-basierte Lösung. Ein IDL-Compiler muß eine der beiden Varianten unterstützen und entsprechend ausgestattete Stubs/Skeletons generieren, wohingegen eine CORBA-konforme ORB-Runtime-Umgebung mit beiden APIs zurechtkommen muß, um beliebige Stubs/Skeletons verarbeiten zu können.

In diesem Kapitel sollen die serverseitige Skeleton-Architektur, die Realisierung lokaler Objektaufrufe und die Verwendung dieser Klassen und Interfaces in POA-basierten Java-Serveranwendungen im Mittelpunkt stehen. Für das Mapping der IDL-Sprachkonstrukte und für alle Fragen der clientseitigen Stub-Architektur, soweit sie nicht die lokalen Aufrufe betreffen, sei auf das offizielle OMG-Dokument zum Java-Mapping verwiesen [26]. Für den Serveranwendungsentwickler ist das Mapping des *native* IDL-Typs `PortableServer::Servant` von zentraler Bedeutung.

5.1. Skeleton-Architektur

Um die Implementierung eines CORBA-Objektes zur Verfügung zu stellen, muß der Entwickler einer Java-Serveranwendung eine Implementierungsklasse schreiben. Die von ihm bereitgestellte Implementierung der Operationen korrespondiert mit den spezifizierten Operationen des IDL-Interfaces, die nach den Regeln des Language-Mappings auf abstrakte Operationen einer Programmiersprache abgebildet werden. Der IDL-Compiler generiert aus einem IDL-Interface `Foo` zwei verschiedene Skeleton-Klassen und ein Operations-Interface. Das `FooOperations`-Interface kapselt die Definition aller vom IDL-Interface abgeleiteten Operationen. Das Java-

Mapping spezifiziert zwei alternativ verwendbare Skeleton-Klassen, die sich in ihrer Beziehung zur Implementierungsklasse unterscheiden. Die abstrakte Klasse `FooPOA` stellt die grundsätzliche Skeleton-Funktionalität zur Verfügung und erweitert diese um die abstrakten Operationen des `FooOperations`-Interfaces. Der Entwickler, der den vererbungsbasierten Ansatz wählt, kann seine Implementierungsklasse, bsw. `FooImpl`, durch Vererbung um die Skeleton-Funktionalität erweitern. Da Java keine multiple Vererbung unterstützt, gibt es damit keine Möglichkeit mehr, Anwendungsfunktionalität hinzuzuerben. Da innerhalb von Anwendungsentwürfen die Verteilungsfunktionalität oftmals außen vor gelassen wird und ebenso Vererbungsbeziehungen zwischen den Anwendungsklassen bestehen können, ist ein alternativ einsetzbarer, delegationsbasierter Ansatz vorgesehen. Die dafür generierte konkrete Tie-Klasse `FooPOATie` erbt die komplette Skeleton-Funktionalität, implementiert aber das Operations-Interface auf eine Weise, daß jeder Operationsaufruf an ein sogenanntes Delegate-Objekt weitergegeben wird. Der Entwickler stellt dieses Delegate-Objekt als eine Instanz seiner Implementierungsklasse, bsw. `FooDelegate`, bei der Erzeugung einer `FooPOATie`-Instanz zur Verfügung. Die Implementierungsklasse muß also in beiden Fällen das Operations-Interface implementieren, mit dem Unterschied, daß beim delegationsbasierten Ansatz die Möglichkeit des Erbens von Anwendungsfunktionalität erhalten bleibt.

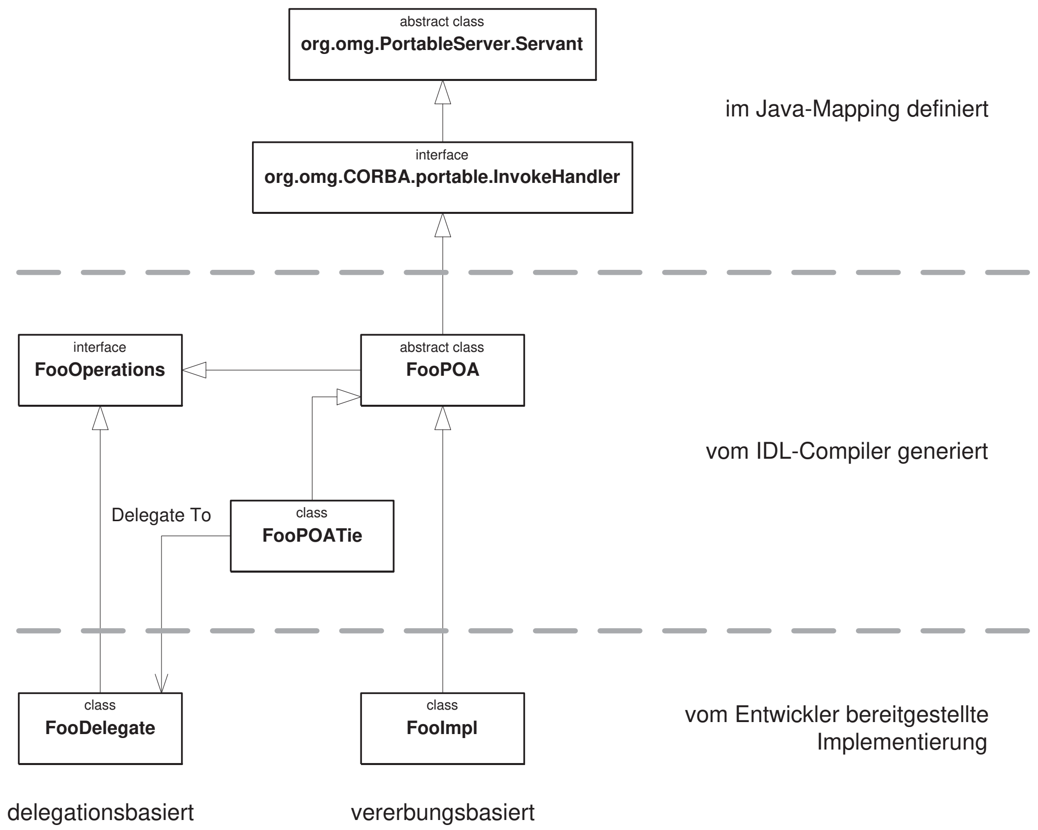


Abbildung 5.1.: Die stream-basierte Skeleton-Architektur

Damit ein Objekt-Adapter eine Anfrage an eine Objekt-Implementierung ausliefern kann, muß der Servant eines der bereits erwähnten APIs unterstützen. Der IDL-Compiler generiert die entsprechende Aufrufoperation in die Skeleton-Klasse hinein. Diese übernimmt anhand des Operationsnamens das *Dispatching* eines Aufrufes auf die vom Entwickler bereitgestellte Implementierung der Operation und kapselt das *Unmarshaling* der Eingabeparameter und das *Marshaling* des Rückgabewertes bzw. einer auftretenden User-Exception. Das Java-Mapping definiert die Code-Generierung der Skeleton-Klassen als eine *Binary Portability Layer*, um die Binärkompatibilität von Servant-Klassen bei ihrer Verwendung in ORB-Umgebungen verschiedener Herrsteller zu garantieren. Auf der anderen Seite stellt die ORB-Implementierung im Java-Mapping festgelegte Schnittstellen bereit, um die Kommunikation zwischen Servant und ORB bzw. Objekt-Adapter, die Übermittlung der Aufrufparameter und die Rückgabe des Ergebnisses zu ermöglichen.

5.1.1. Stream-basiertes API

Werden vom IDL-Compiler stream-basierte Skeleton-Klassen generiert, so implementieren diese das Java-Interface `org.omg.CORBA.portable.InvokeHandler`. Der Object-Adapter ruft bei einer Anfrage an einen Servant dessen `_invoke()` Operation auf und übergibt dieser den Namen der Operation als String, einen Input-Stream, von dem die Eingabeparameter gelesen werden können und einen Response-Handler, von dem ein Output-Stream bezogen werden kann. Nach dem Lesen der Eingabeparameter und dem Aufruf der eigentlichen Operation wird das Ergebnis auf den Output-Stream geschrieben. Im folgenden sei für ein einfaches IDL-Interface `Foo` ein Ausschnitt der generierten stream-basierten Skeleton-Klasse angegeben.

```
interface Foo {
    long operation(in string parameter);
};

abstract public class FooPOA extends org.omg.PortableServer.Servant
                     implements org.omg.CORBA.portable.InvokeHandler,
                                FooOperations {
    ...
    public org.omg.CORBA.portable.OutputStream _invoke(
                   String                                 op_name,
                   org.omg.CORBA.portable.InputStream     is,
                   org.omg.CORBA.portable.ResponseHandler handler) {
        org.omg.CORBA.portable.OutputStream os = null;
        if (''operation''.equals(op_name)) {
            java.lang.String s = is.read_string();
            int result = this.operation(s);
            os = handler.createReply();
            os.write_long(result);
            return os;
        } else {
            throw new org.omg.CORBA.BAD_OPERATION();
        }
```

```
    }
    ...
}
```

Eine Implementierung der abstrakten Java-Klasse `org.omg.CORBA.portable.InputStream` stellt eine Menge von Operationen bereit, mit denen sich Werte bzw. Instanzen aller in IDL vorhandenen Parametertypen auslesen lassen. Diese Lese-Operationen, wie z.B. `read_string`, liefern den gewünschten Wert bzw. die Instanz des Parametertyps zurück, führen also intern das Unmarshaling der Parameter aus. Die Reihenfolge, mit der die Parameter aus dem Input-Stream gelesen werden und die Lese-Operationen, die zur Anwendung kommen, werden vom IDL-Compiler aus der IDL-Interface-Definition statisch abgeleitet. Analog dazu stellt eine Implementierung der abstrakten Klasse `org.omg.CORBA.portable.OutputStream` die Operationen zum Schreiben des Rückgabewertes auf den Output-Stream zur Verfügung (Marshaling), wie z.B. `write_long(result)`. Eine Implementierung des Interfaces `org.omg.CORBA.portable.ResponseHandler` hat nur die Aufgabe, einen Output-Stream bereitzustellen, der den Rückgabewert bzw. eine auftretende User-Exception aufnehmen kann. Dafür bietet ein Response-Handler zwei getrennte Operationen an, `createReply` bzw. `createExceptionReply`.

Durch die Benutzung von Streams und die Einbettung in das in Java vorhandene Stream-Konzept kann eine Java-ORB-Implementierung auf sehr komfortable Art und Weise die Verbindung zwischen dem Serverport, auf dem eine Anfrage entgegengenommen bzw. beantwortet wird, und der Objekt-Implementierung realisieren. Von einer notwendigen Pufferung der Daten und den Details einer offenen Netzwerkverbindung kann ein Java-ORB-Implementierer schon sehr frühzeitig abstrahieren.

5.1.2. DSI-basiertes API

Das *Dynamic Skeleton Interface* bietet die Möglichkeit, eine Objekt-Implementierung bereitzustellen, obwohl das implementierte Interface zur Übersetzungszeit noch nicht feststeht. Für die dynamische Behandlung von Servant-Aufrufen wird vom ORB bzw. vom Objekt-Adapter die *Dynamic Implementation Routine* (DIR) der Skeleton-Klasse aufgerufen, die für jede Objekt-Implementierung unabhängig der implementierten Interfaces dieselbe Signatur besitzt. Der DIR wird ein Request-Objekt übergeben, mit dessen Hilfe die gefragte Operation und die aktuellen Parameter ermittelt werden können. Durch die Einbeziehung des *Interface Repositories* (IR) kann eine Anfrage dynamisch aufgebaut werden. Bei der Generierung von DSI-basierten Skeleton-Klassen werden aber selbstverständlich die statischen Typ-Informationen des IDL-Interfaces genutzt, um eine optimierte und somit spezialisierte Version einer Skeleton-Klasse zu erhalten.

DSI-basierte Skeleton-Klassen implementieren das `org.omg.CORBA.PortableServer.DynamicImplementation` Interface. Der Objekt-Adapter ruft bei einer Anfrage an einen Servant die `invoke()` Operation (DIR) auf und übergibt dieser ein `ServerRequest`-Objekt. Das Pseudo-Interface (PIDL) `CORBA::ServerRequest` wird im Java-Mapping auf die abstrakte Klasse `org.omg.CORBA.ServerRequest` abgebildet. Eine Implementierung dieser abstrakten Klasse gestattet den Zugriff auf die Request-Informationen und die Rückgabe der Ergebnisse bzw. einer auftretenden Exception. Für die Übernahme der Operationsparameter bzw. die Speicherung von "inout" oder "out" Parametern wird die vom DII bekannte `NVList`-Struktur verwendet. Als Behälter dienen `Any`-Objekte. Der ORB stellt Serviceroutinen für die Erzeugung von `NVList`-, und `Any`-Instanzen zur Verfügung. `Any`-Objekte erlauben die Speicherung der Werte beliebiger IDL-Typen zusammen mit den `TypeCode`-Informationen. Für jeden IDL-Typ hält ein `Any`-Objekt eine Ein- und Ausleseroutine bereit. Das Unmarshaling der Aufrufparameter erfolgt also intern durch den ORB und die Bereitstellung der Werte innerhalb von `Any`-Objekten und die anschliessende Extrahierung dieser Werte. Für das *Dispatching* der Anfrage auf die auszuführende Servant-Operation wird der im `ServerRequest`-Objekt enthaltene Operationsname herangezogen. Für die Speicherung des Rückgabewertes bzw. einer auftretenden Exception wird ebenfalls ein `Any`-Objekt verwendet. Dieses wird nach Ausführung des Servant-Aufrufes dem `ServerRequest`-Objekt übergeben. Der ORB übernimmt das Marshaling der Ergebnisse und die Rückübertragung an den Client. Analog zum Beispiel des vorherigen Abschnittes sei hier ein Ausschnitt der DSI-basierten Skeleton-Klasse aufgeführt.

```
abstract public class FooPOA extends org.omg.PortableServer.Servant
             implements org.omg.PortableServer.DynamicImplementation,
                        FooOperations {
    ...
    public void invoke(org.omg.CORBA.ServerRequest request) {
        String op_name = request.operation();
        if (''operation''.equals(op_name)) {
            org.omg.CORBA.NVList params = _orb().create_list(1);
            org.omg.CORBA.Any s_any = _orb().create_any();
            s_any.type(_orb().get_primitive_tc(
                                  org.omg.CORBA.TCKind.tk_string));
            params.add_value(''s'',s_any,org.omg.CORBA.ARG_IN.value);
            request.arguments(params);
            String s = s_any.extract_string();
            int result = this.operation();
            org.omg.CORBA.ANY result_any = _orb.create_any();
            result_any.insert_long(result);
            request.set_result(result_any);
        } else {
            throw new org.omg.CORBA.BAD_OPERATION();
        }
    }
    ...
}
```

Obwohl DSI-basierte Skeleton-Klassen hinsichtlich des implementierten IDL-Interfaces optimiert werden können und somit der Zugriff auf ein *Interface Repository* entfällt, scheint der betriebene Aufwand zur Bereitstellung der Operationsparameter und Rückgabewerte größer zu sein als der, der bei stream-basierten Skeleton-Klassen anfällt. Die separate Erzeugung eines `Any`-Objektes für jeden einzelnen Wert wird bei einer Java-Implementierung schwerer ins Gewicht fallen als die in Java gut integrierte Stream-Lösung. Damit dürfte die Wahl des bevorzugten Skeleton-Typs für die IDL-Compiler-Implementierung eines Java-ORBs aus Performance-Gründen für die stream-basierte Variante ausfallen. Die ORB-Implementierung muß natürlich mit beiden Varianten von Skeleton-Klassen zusammenarbeiten können.

5.1.3. Servant-API

Die bisher beschriebene Skeleton-Funktionalitiät eines Servants ist vor allem für die ORB-Implementierung einschließlich dem IDL-Compiler von Bedeutung. Neben der Implementierung eines oder mehrerer IDL-Interfaces stellt eine Servant-Klasse die Funktionalität der abstrakten Klasse `org.omg.PortableServer.Servant` bereit. Sie bildet den *native* IDL-Typ `Servant` ab. Die `Servant`-Schnittstelle wird vom ORB für die Verwaltung von Objekt-Implementierungen, den Zugriff auf Typinformationen der unterstützten IDL-Interfaces und die Erzeugung lokaler Objektreferenzen genutzt. Dem Anwendungsentwickler werden Hilfsmethoden bereitgestellt, die ihm die implizite Aktivierung von Objekten und einen bequemen Zugriff auf lokale Objektreferenzen eines aktivierten Servants oder auf Informationen des eigenen Aufrufkontextes erlauben. Diese Hilfsmethoden bieten dem Entwickler eine alternative Zugriffsmöglichkeit auf häufig verwendete ORB- bzw. POA-Funktionalität, wobei oft mehrere Funktionen zusammengefaßt und über die `Servant`-Schnittstelle verfügbar gemacht werden. Das Language-Mapping definiert für fast alle Operationen ein Default-Verhalten. Durch das Überschreiben von Operationen hat der Entwickler die Möglichkeit, diese an die Erfordernisse der Objekt-Implementierung anzupassen.

In der neuesten Version des Java-Mappings (CORBA V2.3 Stand: Juni 1999) wurde, ähnlich wie auf der Clientseite, ein simpler Delegate-Mechanismus eingeführt. Die Implementierung der Klasse `org.omg.PortableServer.Servant` ist somit vollständig im Java-Mapping definiert, wobei eine Instanz dieser Klasse nun ein Delegate-Objekt benutzt, an das die Operationsaufrufe weitergeleitet werden. Die eigentliche Implementierung der Servant-Funktionen wird durch eine Implementierung des Interfaces `org.omg.PortableServer.portable.Delegate` geleistet. Damit haben die ORB-Implementierer die Möglichkeit, ORB-spezifische Schnittstellen innerhalb der Implementierung zu verwenden.

```
package org.omg.PortableServer;
import org.omg.CORBA.ORB;
import org.omg.PortableServer.portable.Delegate;

abstract public class Servant {
    private transient Delegate _delegate = null;

    final public Delegate _get_delegate() {
        if (_delegate == null) {
            throw new org.omg.CORBA.BAD_INV_ORDER(
      ''The servant has not been associated with an ORBinstance'');
        }
        return _delegate;
    }
    final public void _set_delegate(Delegate delegate) {
        _delegate = delegate;
```

```
    }
    public POA _default_POA() {
        return _get_delegate().default_POA(this);
    }
    public boolean _is_a(String repId) {...}
    public boolean _non_existent() {...}
    public org.omg.CORBA.InterfaceDef _get_interface() {...}

    final public org.omg.CORBA.Object _this_object() {...}
    final public org.omg.CORBA.Object _this_object(ORB orb) {
        try {
            ((org.omg.CORBA_2_3.ORB)orb).set_delegate(this);
        } catch (ClassCastException e) {
            throw new org.omg.CORBA.BAD_PARAM(
      ''POA Servant requires an instance of org.omg.CORBA_2_3.ORB'');
        }
        return _this_object();
    }
    final public org.omg.CORBA.ORB _orb() {...}
    final public POA _poa() {...}
    final public byte[] _object_id {...}

    abstract public String[] _all_interfaces(POA poa, byte[] oid);
}
```

Mit Ausnahme der Operationen `_this_object(ORB orb)` und `_all_interfaces()` dürfen die Methoden auf einem Servant nur aufgerufen werden, wenn dieser vorher mit einem `Delegate`-Objekt assoziiert wurde. Alle anderen Operationen werfen eine **BAD_INV_ORDER** System-Exception, wenn keine Assoziation mit einem Delegate eingerichtet wurde. Das `Delegate`-Objekt wird von der `ORB`-Instanz zur Verfügung gestellt, in dessen POA-Hierarchie der Servant verwendet wird. Voraussetzung ist die Verwendung einer Instanz einer ORB-Klasse, die die abstrakte Klasse `org.omg.CORBA_2_3.ORB` erweitert. Diese stellt u.a. die Operation `set_delegate` zur Verfügung, der eine `Servant`-Instanz übergeben werden kann und die das Setzen des Delegates übernimmt. Mit dem Einrichten dieser Assoziation wird gleichzeitig das Delegate mit der `ORB`-Instanz verbunden. Die Assoziation mit dem ORB bildet die Voraussetzung für die Funktionsweise der Operationen, die ein kontextabhängiges Verhalten aufweisen, da sie auf die ORB-Initialisierungsschnittstelle zur Ermittlung des Aufrufkontextes zugreifen müssen.

Das Einrichten der Assoziation zwischen Servant und Delegate und damit der Verbindung zwischen Servant und ORB wird normalerweise implizit als Folge einer Registrierung eines Servants bei einem POA durchgeführt. Dies gilt für die explizite Objektaktivierung, also den Aufruf der Operationen `activate_object` bzw. `activate_object_with_id`, die Registrierung eines Default-Servants und die Bereitstellung eines Servants durch einen Servant-Manager als Folge des Aufrufes von `incarnate()` bzw. `preinvoke()`. Es ist aber auch möglich, diese Assoziation in einer Serveranwendung explizit herzustellen, indem die Operation `set_delegate` auf einer `ORB`-Instanz, oder die Operation `_this_object(ORB orb)` auf einem Servant aufgerufen wird. Die Operation `_this_object(ORB orb)` wirft eine **BAD_PARAM** System-Exception, wenn es sich nicht um eine `ORB`-Instanz handelt, die ein Servant-Delegate setzen kann (Instanz von `org.omg.CORBA_2_3.ORB`).

Verwaltungsoperationen für die ORB-Implementierung

Die `_default_POA()` Operation liefert den Default-POA zurück, der außerhalb des Kontextes eines Servant-Aufrufes genutzt werden soll, um eine implizite Objektaktivierung zu ermöglichen. Standardmäßig wird der Root-POA zurückgegeben. Die Default-Implementierungen der Operationen `_get_interface` und `_is_a` nutzen die Typinformationen, die über die Implementierung von `_all_interfaces` erhältlich sind, um entweder eine Objektreferenz eines `InterfaceDef`-Objektes zurückzuliefern, das die Erforschung und Manipulation eines korrespondierenden Eintrages im IR erlaubt bzw. einen Vergleich mit einer übergebenen Repository-ID anstellt. Die Default-Implementierung von `_non_existent()` liefert **FALSE** zurück. Alle diese Operationen können vom Entwickler überschrieben werden, um ein abweichendes Verhalten zu definieren. Die beim Entwickler populärste Operation zum Überschreiben ist mit Sicherheit `_default_POA()`, mit der er einen vom Root-POA abweichenden POA festlegen kann, der dann bei impliziter Objektaktivierung zum Einsatz kommt. Bei der Verwendung einer Tie-Klasse für die Servant-Implementierung ist zu beachten, daß zum Überschreiben dieser Methoden vom Entwickler eine eigene Tie-Klasse abgeleitet werden muß.

Die Implementierung der abstrakten Operation `_all_interfaces` wird vom IDL-Compiler generiert und aus der IDL-Interface-Definition abgeleitet. Sie liefert die Informationen über alle implementierten IDL-Interfaces als eine Sequenz von Repository-IDs in einem String-Array zurück. Das Basis-Interface "IDL:omg.org/CORBA/Object:1.0" ist nicht enthalten. Das *most derived* Interface, in unserem Beispiel wäre es "IDL:Foo:1.0", ist über den Index **0** des Arrays zugreifbar. Der ORB benutzt diese Information ebenfalls zur Erzeugung von Objektreferenzen. Die Operation `_all_interfaces` wird gewöhnlich nicht vom Entwickler überschrieben.

Hilfsmethoden für den Anwendungsentwickler

Die `_orb()` Methode stellt eine Convenience-Operation für den Entwickler dar, die die aktuell mit dem Servant verbundene `ORB`-Instanz zurückliefert. Die beiden Operationen `_poa` und `_object_id` sind nur innerhalb eines Kontextes eines Aufrufes auf einem CORBA-Objekt gestattet. Sollte dies nicht der Fall sein, wird eine **OBJ_ADAPTER** System-Exception geworfen. Beide Operation greifen über die ORB-Initialisierungsschnittstelle auf das POA-`Current`-Objekt zu und bieten dem Entwickler einen bequemen Zugang zu den beiden `Current`-Operationen `get_POA` und `get_object_id`. Die Operation `_poa` liefert den Kontext-POA, der die aktuelle Anfrage an einen beliebigen Servant vermittelt hat und die Operation `_objectd_id` liefert die Objekt-ID des Zielobjektes, auf dem der Aufruf durchgeführt wird.

Die wohl interessanteste `Servant`-Operation für den Anwendungsentwickler ist `_this_object()`. Sie erlaubt ihm, über die `Servant`-Instanz eine lokale Objektreferenz des CORBA-Objektes zu erhalten, das durch den Servant inkarniert wird. Innerhalb des Kontextes eines CORBA-Aufrufes, wobei das Zielobjekt durch diesen Servant inkarniert sein muß, wird die lokale Objektreferenz des Zielobjektes zurückgegeben.[1] Durch den Zugriff auf die Informationen des Aufrufkontextes (siehe Abschnitt 4.7) führt die Operation unabhängig von den geltenden Policies des vermittelnden Kontext-POAs zum Ziel, da sowohl der aktuelle Servant also auch die Objekt-ID des Zielobjektes Teil der Kontext-Informationen sind. Der Zugriff auf den Kontext-POA erfolgt durch Aufruf der `_poa()` Operation. Außerhalb des Kontextes eines Aufrufes auf diesem Servant wird anstelle des Kontext-POAs der Default-POA herangezogen. Dieser ist durch den Aufruf von `_default_POA()` erhältlich. Je nach geltenden Policies für den Default-POA kann nun nicht mehr in jedem Fall eine Objektreferenz des inkarnierten CORBA-Objektes ermittelt werden. Voraussetzung ist der **RETAIN** Policy-Wert und somit die Verwaltung der Objekt-ID-zu-Servant-Beziehungen durch den Default-POA. Außerdem darf ein Servant nicht mehrere CORBA-Objekte inkarnieren können, d.h. es muß der **UNIQUE_ID** Policy-Wert gelten. Für den Fall, daß der Servant nicht in der AOM enthalten ist, muß eine implizite Objektaktivierung möglich sein. Dafür ist wiederum der **IMPLICIT_ACTIVATION** Policy-Wert die Voraussetzung. Die registrierte Objekt-ID eines schon aktivierten Servants bzw. die neu generierte Objekt-ID wird genutzt, um eine korrespondierende Objektreferenz zurückzugeben. Sollte eine der beschriebenen Policies nicht für den Default-POA gelten, wird eine **OBJ_ADAPTER** System-Exception geworfen.

Bei der Verwendung der Operation `_this_object()` außerhalb eines Aufrufkontextes und zur impliziten Objektaktivierung ist zu beachten, daß der Servant mit einem Delegate initialisiert sein muß. Beim erstmaligen Aufruf sollte also der ORB übergeben werden, der den Servant mit einem Delegate verbinden kann. Die Operation `_this_object(ORB orb)` ergänzt die im Absatz zuvor beschriebene Funktionalität um die vorherige Einrichtung der Assoziationen mit Delegate und ORB.

In früheren Versionen des Java-Mappings war die Semantik der `_this_object()` Operation, nachdem der zuständige POA ermittelt wurde, identisch mit der POA-Operation `servant_to_reference()`. Durch das Fehlen des Servant-Delegates wurde in diesen Versionen die Funktionalität direkt von der `Servant`-Klasse implementiert, so daß nur die spezifizierte POA-Schnittstelle zur Verfügung stand, um portable Skeleton-Klassen zu generieren. Dies ist in der neuesten Version des Java-Mappings mit Einführung des Servant-Delegates und der damit möglichen Verwendung ORB-interner Schnittstellen nicht mehr der Fall. Im Unterschied zur Funktionalität der `servant_to_reference()` Operation besitzt die `_this_object()` Operation restriktivere Vorgaben hinsichtlich der zulässigen Policies für den POA (siehe Abschnitt 4.9). Eine versehentliche Erzeugung von CORBA-Objekten bei geltendem **MULTIPLE_ID** Policy-Wert durch die implizite Aktivierung immer neuer Objekte für jeden Aufruf von `_this_object()` auf dem selben Servant ist nun nicht mehr möglich.

5.1.4. Convenience-API

Die im vorherigen Abschnitt beschriebenen Hilfsmethoden zur Ermittlung einer lokalen Objektreferenz können nur Instanzen vom Typ `org.omg.CORBA.Object` zurückgeben, da ihre Implementierung in der `Servant`-Klasse erfolgt, von der alle generierten Skeleton-Klassen erben. Um ein spezielles IDL-Interface innerhalb einer Server-

[1] Ein Servant kann darüber hinaus mehrere CORBA-Objekte inkarnieren.

anwendung verwenden zu können, muß der Entwickler mit der erhaltenen Objektreferenz ein *Narrowing* durchführen. Für das *Narrowing* von Objektreferenzen wird die statische `narrow()` Operation der ebenfalls generierten Helper-Klassen benutzt. Diese Operation liefert eine Objektreferenz des *most derived* IDL-Interface-Typs als eine Instanz der entsprechenden Stub-Klasse zurück. Ob die Implementierung der `narrow()` Operation einen einfachen *Down-Cast* der übergebenen Objektreferenz versucht oder anhand der statischen Typ-Informationen deren Kompatibilität überprüft (`_is_a()` Operation) und ggf. eine neue Instanz der Stub-Klasse erzeugt, um diese erst bei Bedarf bereitzustellen, bleibt der ORB-Implementierung überlassen. Um nun von einem Servant eine Objektreferenz zu erhalten, die auch das implementierte IDL-Interface unterstützt, kann der Entwickler bsw. folgenden Code schreiben.

```
Foo ref = FooHelper.narrow(servant._this_object());
```

Diese Funktionalität wird häufig verwendet, und da die Typ-Informationen bei der Generierung der Skeleton-Klassen bekannt sind, werden diese mit zwei zusätzlichen Methoden ausgestattet, die statt einer Instanz von `org.omg.CORBA.Object` eine Objektreferenz des *most derived* IDL-Interface-Typs zurückliefern. Die Operationen `_this()` und `_this(ORB orb)` besitzen die gleiche Funktionalität wie die im vorherigen Abschnitt beschriebenen Operationen `_this_object()` bzw. `_this_object(ORB orb)`, mit dem Unterschied, daß sie zusätzlich das *Narrowing* des Ergebnisses durchführen. Der oben angeführte Beispielcode reduziert sich nun auf die einfache Verwendung der `_this()` Operation.

```
Foo ref = servant._this();

abstract public class FooPOA extends org.omg.PortableServer.Servant
             implements org.omg.PortableServer.DynamicImplementation,
                        FooOperations {
    public Foo _this() {
        return FooHelper.narrow(super._this_object());
    }
    public Foo _this(ORB orb) {
        return FooHelper.narrow(super._this_object(orb));
    }
}
```

5.2. Lokale Objekte

Im Abschnitt 4.8 der Arbeit wurde schon darauf hingewiesen, daß Aufrufe auf einem lokalen CORBA-Objekt mit einer höheren Effizienz als Fern-Aufrufe durchgeführt werden können. Der Aufruf auf einem lokalen Servant soll letztendlich als normaler Objekt-Aufruf einer Programmiersprache ausgeführt werden. Für die Vermittlung eines lokalen Servants an eine Objektreferenz zur direkten Durchführung des Servant-Aufrufes definiert das Java-Mapping ein *"Local Invocation API"*.

Ich möchte in diesem Abschnitt nicht die komplette Stub-Architektur beschreiben, sondern nur auf die Dinge eingehen, die die serverseitige ORB-Implementierung betreffen, um lokale Objekt-Aufrufe zu realisieren. Für weitergehende Fragen zur

Stub-Architektur sei auf die offizielle Spezifikation zum Java-Mapping verwiesen [26]. Die dort definierte Architektur berücksichtigt die lokale Verwendung von Objektreferenzen. Die vom IDL-Compiler generierten Stub-Klassen enthalten Code, der ausschließlich für die Ausführung bei einem lokalen Aufruf gedacht ist. In Bezug auf das schon eingeführte `Foo`-Beispiel werde ich den relevanten Ausschnitt einer generierten Stub-Klasse angeben. Im Gegensatz zu früheren Versionen des Java-Mappings werden keine extra Stub-Klassen generiert, die ausschließlich für lokale Objektreferenzen gedacht sind. Die "aktuellen" Stub-Klassen enthalten sowohl die Funktionalität für einen Fern-Aufruf, als auch diejenige, die für einen lokalen Aufruf benötigt wird.

Ähnlich wie bei einem Servant wird auch für die Implementierung einer Objektreferenz ein Delegationsmechismus benutzt, bei dem die Implementierung von `ObjectImpl` alle Operationsaufrufe von `org.omg.CORBA.Object` an eine Instanz von `org.omg.CORBA.portable.Delegate` weiterreicht. Dieses Delegate kapselt die herstellerspezifische Implementierung einer CORBA-Objektreferenz.

```
import org.omg.CORBA.portable.*;

public class _FooStub extends org.omg.CORBA.portable.ObjectImpl {
    final public static java.lang.Class opsClass =
                                        FooOperations.class;
    public int operation(String parameter) {
        while(true) {
            if (!this._is_local()) {
                ... // Remote Call Functionality
            } else {
                ServantObject so = _servant_preinvoke(''operation'',
                                                       opsClass);
                if (so == null) {
                    continue;
                }
                FooOperations self = (FooOperations)so.servant;
                try {
                    return self.operation(parameter);
                } finally {
                    _servant_postinvoke(so);
                }
            }
        }
    }
}
```

Die von `org.omg.CORBA.portable.ObjectImpl` bereitgestellte `_is_local()` Operation wird innerhalb der Implementierung einer vom IDL-Interface abgeleiteten Operation benutzt, um zwischen einem lokalen und einem Fern-Aufruf zu unterscheiden. Bei der Erzeugung einer Objektreferenz wird das bereitgestellte `Delegate`-Objekt entsprechend parametrisiert, so daß es für den lokalen Anwendungsfall den Wert **TRUE** zurückgeben kann (das Delegate und der inkarnierende Servant müssen mit der selben ORB-Instanz assoziiert sein). Standardmäßig wird von der `_is_local()` Operation der Wert **FALSE** zurückgegeben.

Eine Java-Referenz des aktuellen `Servant`-Objektes wird durch einen Aufruf von `_servant_preinvoke()` ermittelt. Genauer gesagt liefert dieser an das Delegate weitergereichte Aufruf eine Instanz von `org.omg.CORBA.portable.ServantObject` oder den Wert **null**. Ein `ServantObject` dient nur dazu, eine `Servant`-Instanz aufzunehmen, die über das einzige Attribut `servant` gesetzt und abgefragt werden kann.

```
package org.omg.CORBA.portable;

public class ServantObject {
    public org.omg.PortableServer.Servant servant;
}
```

Der `_servant_preinvoke()` Operation wird zur Ermittlung des Servants der Operationsname und das entsprechende `Class`-Objekt der Operations-Klasse übergeben. Das `Class`-Objekt repräsentiert den erwarteten Typ des Servants. Falls vom Delegate kein Servant ermittelt werden kann oder ein ermittelter Servant nicht bzw. nicht mehr lokal ist, wird der Wert **null** zurückgegeben. Wenn bsw. ein Servant-Manager, der die Anfrage letztendlich bearbeitet, eine **ForwardRequest** Exception generiert, könnte er die Anfrage an eine andere Objekt-Implementierung umleiten, die nicht mehr innerhalb dieses Prozesses residiert oder nicht von der selben ORB-Instanz verwaltet wird. Damit wird vom Stub die Auslieferung der Anfrage erneut versucht und die `_is_local()` Operation könnte nun den Wert **FALSE** ergeben. Das Delegate, das den Aufruf erneut über die "normale" stream- bzw. DII-basierte Schnittstelle empfängt, kann den Stub nun an die neue Adresse (IOR) binden.

Falls der ermittelte Servant nicht dem erwarteten Typ entspricht, wird eine **BAD_PARAM** System-Exception erzeugt. Um das im Abschnitt 4.8 angesprochene Lokalitätstransparenz-Prinzip zu gewährleisten, sollte eine ORB-Implementierung die Auslieferung der Anfrage an den zuständigen POA und dessen Ermittlung des zuständigen Servants analog zur gewöhnlichen Anfrageauslieferung implementieren und die `_servant_preinvoke()` Operation solange blockieren lassen, bis ein Servant bereitgestellt werden kann.

Wird ein `ServantObject` zurückgegeben, so sollte dieses auch eine Instanz des erwarteten Typs enthalten. Um dies sicherzustellen, kann der Stub einen *Type-Cast* durchführen und dann die Operation direkt auf dem Servant aufrufen. Für die Implementierung des Delegate ist zu beachten, daß ein `ServantObject` nur für einen einzigen Aufruf verwendet werden darf. Die Operation `_servant_prostinvoke` wird aufgerufen, wenn das `ServantObject` ermittelt werden konnte, auch für den Fall, daß innerhalb des Servant-Aufrufes eine Exception auftritt. Beim Aufruf von `_servant_postinvoke()` muß die vorher ermittelte Instanz von `ServantObject` übergeben werden.

Der eigentliche Servant-Aufruf läuft damit innerhalb des Threads für den Aufruf auf der Objektreferenz ab und ist somit außerhalb der Kontrolle durch den verantwortlichen POA. Durch die Einrahmung des Servant-Aufrufes mit den Operationen `_servant_preinvoke` und `_servant_prostinvoke` kann die ORB-Implementierung die Kontext-Informationen verwalten, um innerhalb des Servant-Aufrufes darauf zurückgreifen zu können (siehe Abschnitt 4.7).

Einige Objekte einer Serveranwendung können **nur** lokal verwendet werden (siehe Abschnitt 4.8.2). Dazu zählen der Adapter-Activator und die Servant-Manager der POA-Spezifikation. Für diese Stub-Klassen ist es ausreichend, den Code für den lokalen Aufruf zu erzeugen. Für einige dieser Objekte ist dies sogar zwingend notwendig, da die IDL-Interfaces teilweise Parameter-Typen enthalten, für die kein fehlerfreier Marshaling-Code generiert werden kann. Da sich die IDL-Interface-Definition aber nicht von denen anderer Schnittstellen unterscheidet, jedenfalls nicht offensichtlich, ist eine automatische Behandlung des Problems schwierig. Eine ORB-Implementierung kann sich damit behelfen, die fehlerhaft generierten Stub-Klassen zu überschreiben oder ein Pragma einzuführen, um ein solches Interface im IDL-Text als "*locality constrained*" zu kennzeichnen.

6. Implementierung eines POAs für JacORB

Im Rahmen dieser Diplomarbeit wurde der POA für JacORB implementiert. Seit 1997 wird JacORB im wesentlichen von Gerald Brose an der Freien Universität Berlin als eine freie Java-Implementierung des CORBA-Standards entwickelt. Die Implementierung des POAs war Bestandteil des Bestrebens, JacORB an die mittlerweile fortgeschrittene Standardisierung von CORBA anzupassen. Am Beginn der Arbeit für den POA befand sich JacORB auf dem Entwicklungstand der CORBA-Version 2.0. Zwar handelte es sich nicht um eine vollständige Implementierung des Standards, der größte Teil der Funktionen war aber integriert und erlaubte eine Entwicklung von *multi-threaded* Client/Server-Anwendungen in Java. Neben der Unterstützung statischer Aufruffunktionalität (Generierung statischer Stubs/Skeletons durch einen IDL-Compiler) und von Interoperabilität durch die Implementierung von IIOP als Inter-ORB-Protokoll war die Implementierung eines *Naming Services* und eines *Event Services* enthalten. In der Version 0.9 besaß JacORB nur eine vereinfachte BOA-Implementierung (nur *Persistent Servers*, ohne *Implementation Repository*).

Parallel zur Implementierung des POAs wird JacORB an den aktuellen Stand des *Language Mappings* für Java angepaßt (siehe Kapitel 5). Die Fortentwicklung des Stub- und Skeleton-Designs ist die Voraussetzung für die Entwicklung des POAs und portabler POA-basierter Serveranwendungen. Dieser Prozeß ist noch nicht abgeschlossen. Es existieren noch leichte Differenzen zum aktuellen Standard, wie das Kapitel 7 zeigen wird.

Die vereinfachte Implementierung des BOAs wurde aus der Distribution entfernt und durch die POA-Implementierung ersetzt. Mittlerweile stellt JacORB in der aktuellen Version 1.0 den größten Teil der CORBA-2.3-Funktionen bereit, inklusive dynamischer Aufruffunktionalität und der Unterstützung persistenter Objekte (*Server Startup Daemon* und *Implementation Repository*). Zur vollständigen Implementierung des CORBA-Standards fehlen: ein *Interface Repository* und die Implementierung von *Value Types.* Bei einer Beschränkung auf die unterstützte Funktionalität erlaubt JacORB eine CORBA-2.3-konforme Entwicklung verteilter Java-Anwendungen.

Dieses Kapitel beschreibt den Implementierungsentwurf eines POAs für JacORB. Darüber hinaus soll im Detail auf die Implementierung wichtiger POA-Funktionalität und deren Anforderungen an die Synchronisation in einer *multi-threaded* Umgebung eingegangen werden. Zusätzlich zur Implementierung des POAs habe ich mich entschlossen, eine Benutzeroberfläche zu entwickeln, die den Ressourcenverbrauch und die Aktivitäten einer Serveranwendung graphisch darstellt und somit dem Anwendungsentwickler ein hilfreiches Werkzeug zum Entwurf und zum Test seiner Anwendung in die Hand gibt. Im Zusammenhang mit dem Entwurf der Oberfläche habe ich eine Anzahl relevanter POA-Ereignisse definiert und eine Implementierung des von Java bekannten *Event-Listener Patterns* eingeführt. Das Konzept der POA-Ereignisse ist so allgemein gehalten, daß es als flexibler und universeller Erweiterungsmechnismus der POA-Implementierung angesehen werden kann. In einem ersten Schritt werde ich nur die grundsätzliche POA-Funktionalität betrachten. Die sich aus der Einführung der POA-Ereignisse ergebenden Erweiterungen werde ich erst im Anschluß daran darstellen.

6.1. Entwurfsziele

- Der POA-Entwurf soll die spezifizierte Funktionalität vollständig unterstützen. Als Referenz dient die CORBA-Spezifikation v2.3a - *Core final revision* vom Dezember 1998 [22].

- Die POA-Implementierung wird in das JacORB-Framework eingebettet. Dabei sollen Abhängigkeiten von anderen Framework-Klassen minimiert werden.

- Die POA-Implementierung soll *multi-threaded* sein und die in Java vorhandene Thread-Unterstützung voll ausnutzen *(ORB Controlled Model)*. Sie muß alle Vorkehrungen zur Synchronisation treffen, um auch bei der Verwendung von JacORB in einer *multi-threaded* Serveranwendung die in der Spezifikation geforderten Auflagen einzuhalten.

- Für Serveranwendungen, die nicht thread-sicher sind, kann ein POA auch wahlweise als *single-threaded* konfiguiert werden *(Single Thread Model)*, wobei alle Aufrufe des POAs von Anwendungscode (Servant, Servant-Manager, Adapter-Activator) serialisiert werden.

- Es soll ein hohes Maß an Anpaßbarkeit an die besonderen Anforderungen einer Serveranwendung und eine einfache Konfigurierbarkeit der POA-Implementierung ermöglicht werden (ohne die Veränderung des Quellcodes). Dazu gehört die bedarfsgerechte Bereitstellung von Systemressourcen und von Funktionalität.

- Es wird eine leichte Erweiterbarkeit der POA-Implementierung angestrebt. Neben einem erweiterbaren objektorientierten Entwurf sollen für bestimmte, vorgedachte Erweiterungen Systemschnittstellen angeboten werden, die das Hinzufügen von Funktionalität ohne Veränderungen an Implementierungsklassen ermöglichen.

- POA-Implementierungsklassen sollen weitestgehend unabhängig von der Benutzeroberflächenfunktionalität entworfen, benutzt und erweitert werden können, wobei hier ein geeigneter Kompromiß zwischen Laufzeiteffizienz, Flexibilität und Erweiterbarkeit gefunden werden muß.

- Eine hohe Laufzeiteffizienz ist ein wichtiges Entwurfsziel für die POA-Implementierung.

6.2. Implementierungsmodell

Die im Kapitel 4 beschriebene POA-Funktionalität wird im Implementierungsentwurf auf mehrere Implementierungsklassen aufgeteilt. Die einzelnen Klassen und deren Verantwortlichkeiten werden nun übersichtsartig skizziert. Zusätzlich wird auf die Schnittstellen zum restlichen JacORB-Framework und die vom Anwendungsentwickler zur Verfügung gestellten Komponenten eingegangen. Die dabei beschriebenen Klassen sind zwar nicht Teil der POA-Implementierung, ihre Funktionalität wird aber innerhalb der POA-Implementierungsklassen verwendet. Somit ist eine Erklärung für das Verständnis des POA-Entwurfs unerläßlich. Nachdem die einzelnen Modell-Komponenten dargestellt wurden, werden sie mit der Beschreibung der Modell-Architektur in einen Gesamtzusammenhang gestellt. Dieser grobe Blick wird dann später durch die detaillierte Darstellung einzelner, POA-relevanter Prozesse verfeinert. Für das Verständnis dieser Detailerläuterungen ist eine genaue Kenntnis der POA-Spezifikation die Voraussetzung. Sie nutzen dem interessierten Leser vor allem bei der Einarbeitung in den Implementierungsquellcode.

6.2.1. Modell-Komponenten

jacorb.poa.POA Diese Klasse bildet das Kernstück der POA-Implementierung. Sie implementiert die IDL-Schnittstelle `PortableServer::POA`. Diese, vor allem für den Anwendungsentwickler relevante, Schnittstelle wird um eine JacORB-interne und eine paket-interne Schnittstelle erweitert. Die öffentliche, JacORB-interne Schnittstelle dient zur Kommunikation mit dem ORB und beinhaltet eine Klassenmethode zur Erzeugung des Root-POAs und einige Instanzmethoden zur Unterstützung der Anfragebearbeitung und Objektreferenz-Erzeugung. Über die paket-interne Schnittstelle werden anderen Komponenten Zustandsinformationen zugänglich gemacht und Verwaltungsoperationen der POA-Hierarchie angeboten. Der POA-Manager benutzt die paket-interne Schnittstelle zur Durchführung eines Zustandswechsels.

`POA`-Objekte werden bei ihrer Erzeugung mit verschiedenen Policies parametrisiert. Jedes `POA`-Objekt wacht über die Einhaltung der mit den jeweiligen Policies verbundenen Verhaltensregeln. Zu den Aufgaben eines `POA`-Objektes gehört die Erzeugung und Verwaltung von Kind-POAs, die Unterstützung des ORBs beim Auffinden des zuständigen POAs, die Steuerung der POA-Zerstörung, die Steuerung der Zustandswechselprozesse, sowie die Erzeugung von Objektreferenzen und von `Policy`-Objekten. Aufgaben der Anfragebearbeitung werden an ein `RequestController`-Objekt delegiert. Die Aktivierung und Deaktivierung von CORBA-Objekten ist ein weiteres Aufgabenfeld des POA. Die dafür notwendige Funktionalität wird in Kooperation mit dem `RequestController`-Objekt und einer weiteren Komponente, der *Active Object Map* (AOM) geleistet.

jacorb.poa.AOM Ein Objekt dieser Klasse verwaltet eine Datenstruktur zur Speicherung von Objekt-ID-zu-Servant-Beziehungen für einen POA. Synchronisationserfordernisse schließen auch die Einleitung der Inkarnierung und Etherealization von Servants ein, die im Zusammenhang mit einer folgenden Objektaktivierung bzw. einer vorausgehenden Objektdeaktivierung durchgeführt werden. Über eine paket-interne Schnittstelle können Eintragungen vorgenommen bzw. entfernt und die Inkarnation eines Servant eingeleitet werden. Die Abbildung von Objekt-IDs auf

Servants und umgekehrt wird über entsprechende Zugriffsoperationen geleistet.

jacorb.poa.RequestController Ein `RequestController`-Objekt steuert die Anfragebearbeitung für einen einzelnen POA und nimmt die Ressourcenzuweisung für die Bearbeitung einer Anfrage vor. Nach außen wird eine paket-interne Schnittstelle zur Verfügung gestellt, die dem POA die Möglichkeit bietet, eine Anfrage zu übergeben und seinerseits auf die Bearbeitung Einfluß zu nehmen. `RequestProcessor`-Objekte nutzen diese Schnittstelle für den Zugriff auf andere Komponenten und die Benachrichtigung über die Fertigstellung eines Aufrufes. Weitere Operationen der paket-internen Schnittstelle werden später im Detail erläutert.

Der Request-Controller ist das zentrale Element zur Steuerung der Anfragebearbeitung für einen einzelnen POA. Zu seinen Aufgaben gehört, die Pufferung eingehender Aufträge zu veranlassen, die Voraussetzungen für eine Bearbeitung zu prüfen bzw. zu schaffen, sowie die Abarbeitung anzustoßen und zu überwachen. Dazu arbeitet er mit weiteren Komponenten zusammen: der AOM, einer Request-Queue, einem Pool von `RequestProcessor`-Objekten und einem Pool-Manager. Am Ende jeder Anfragebearbeitung steht die Benachrichtigung des ORBs. Außerdem ist der Request-Controller am Deaktivierungsprozeß eines CORBA-Objektes, der Durchführung eines POA-Zustandwechsels und der POA-Zerstörung beteiligt, da er die entscheidenen Synchronisationspunkte herbeiführt, die diese Prozesse erfordern. Für das Auslesen der anstehenden Anfragen aus der Request-Queue und ihre Bearbeitung wird ein eigener Thread bereitgestellt.

jacorb.poa.RequestQueue Ein `RequestQueue`-Objekt dient zur Pufferung eingehender Anfragen. Dafür wird die Datenstruktur einer Queue zur Speicherung von `ServerRequest`-Objekten verwendet. Jedes `RequestQueue`-Objekt kennt seinen Request-Controller und benachrichtigt diesen, wenn neue Anfragen zur Bearbeitung anstehen. Die paket-interne Schnittstelle stellt die auf der Datenstruktur arbeitenden Zugriffsoperationen zusammen.

jacorb.orb.dsi.ServerRequest Diese Klasse implementiert sowohl die IDL-Schnittstelle `CORBA::ServerRequest`, die als Parametertyp für Aufrufe auf DSI-basierten Servants verwendet wird als auch `CORBA::portable::ResponseHandler` und eine JacORB-interne Schnittstelle, die die einfache Verwendung für Aufrufe auf streambasierten Servants erlaubt. Die öffentliche, JacORB-interne Schnittstelle enthält Operationen, mit denen auf Anfrageparameter (z.B. Request-ID, Objekt-Key und Objekt-ID) zugegriffen werden kann.

Die Referenz auf ein `ServerRequest`-Objekt wird dem POA bei der Übergabe eines Aufrufes vom ORB ausgehändigt. Sie eignet sich hervorragend dazu, innerhalb der POA-Implementierung bis hin zum eigentlichen Aufruf verwendet und weitergegeben zu werden. Ein `ServerRequest`-Objekt ermöglicht der POA-Implementierung, vollständig von den Details einer Netzwerkverbindung, dem *Marshaling*- und *Unmarshaling* von Aufrufparametern und Rückgabewerten und ihrer Rückübertragung, inklusive auftretender Exceptions, zu abstrahieren. Es kapselt die bestehende Netzwerkverbindung und liefert der POA-Implementierung alle notwendigen Parameter für den Aufruf einer Operation auf dem Servant.

jacorb.poa.Current Diese Klasse implementiert die IDL-Schnittstelle `PortableServer::Current`, die einen Zugriff auf Informationen innerhalb eines Aufrufkontextes bietet. Bei einem Aufruf kann der zuständige POA und die für diesen Aufruf gültige Objekt-ID ermittelt werden. Die JacORB-interne Schnittstelle enthält eine Klassenmethode zur Erzeugung eines *singelton* POA-Current-Objektes und Operationen zum An- und Abmelden von Kontext-Informationen.

jacorb.poa.InvocationContext Dieses Interface definiert vier Operationen zum Zugriff auf den Aufrufkontext. Die Operationen erlauben die Ermittlung des ORBs, des POAs, der Objekt-ID und des Servants.

jacorb.poa.RPPoolManager Ein `RPPoolManager`-Objekt verwaltet einen Pool von einsatzbereiten `RequestProcessor`-Objekten. Es stellt einem Request-Controller eine Menge von gestarteten Threads zur Ausführung von Aufrufen auf Servant und Servant-Manager zur Verfügung. Die paket-interne Schnittstelle besteht im wesentlichen aus Operationen zum Erhalt und zur Rückgabe eines `RequestProcessor`-Objektes. Der Pool-Manager ist für die Erzeugung und die Zerstörung eines Request-Processors zuständig.

jacorb.poa.RequestProcessor Ein `RequestProcessor`-Objekt stellt einen Thread und damit eine Ablaufumgebung für Aufrufe auf Servant und Servant-Manager zur Verfügung. Über die Operationen einer paket-internen Schnittstelle wird es vom Request-Controller mit den nötigen Parametern, insbesondere dem aktuellen `ServerRequest`-Objekt, versorgt und die Abarbeitung gestartet. Sie enthält weiterhin Operationen zum Beenden eines Request-Processor-Threads. Außerdem werden die Operationen des `InvocationContext`-Interfaces zum Zugriff auf die Parameter des Aufrufkontextes implementiert.

`RequestProcessor`-Objekte kooperieren mit ihrem Pool-Manager. Sie sind nur für die Zeit der Bearbeitung eines Aufrufes aktiv und melden sich nach dessen Fertigstellung wieder beim Pool-Manager zurück. Wenn der Request-Controller im Vorfeld keinen zuständigen Servant ermitteln kann, aber ein Servant-Manager vorhanden ist, stellt der Request-Processor-Thread ebenso eine Ablaufumgebung für eventuell notwendige Aufrufe auf dem Servant-Manager zur Verfügung. Ggf. auftretende Exceptions werden dem `ServerRequest`-Objekt übergeben, bevor dem Request-Controller die Fertigstellung des Aufrufes angezeigt wird. Zusätzlich zu den Basisaufgaben leistet ein Request-Processor Unterstützung beim Aufruf der serverseitigen *Interceptors*. Bei einer entsprechenden Konfiguration des Serveranwendungsprozesses wird das aktuelle `ServerRequest`-Objekt dem ORB ausgehändigt, der dann seinerseits die registrierten *Interceptors* aufrufen kann.

jacorb.poa.LocalInvocationContext Instanzen dieser Klasse sind einfache Daten-Objekte, die die Parameter des Aufrufkontextes speichern können. Für den Zugriff auf dieses Parameter wird das `InvocationContext`-Interface implementiert.

jacorb.poa.POAManager Diese Klasse implementiert die IDL-Schnittstelle `PortableServer::POAManager`. Ein `POAManager`-Objekt kapselt den Zustand aller assoziierten POAs. Der Serveranwendungsentwickler kann die erzeugten POAs durch die

Zuweisung eines `POAManager`-Objektes gruppieren und den Anfragefluß, der die assoziierten POAs erreicht, steuern und diese ggf. gemeinsam deaktivieren. Eine paketinterne Schnittstelle stellt Operationen zum An- und Abmelden von `POA`-Objekten zur Verfügung. Bei einem Zustandswechsel werden alle assoziierten POAs informiert.

Servant Die Implementierung eines oder mehrerer CORBA-Objekte wird vom Anwendungsentwickler zur Verfügung gestellt. Innerhalb einer Serveranwendung werden Instanzen dieser `Servant`-Klassen bei einem POA für eine oder mehrere Objekt-IDs aktiviert. Der JacORB-IDL-Compiler generiert stream-basierte Skeleton-Klassen (siehe Abschnitt 5.1.1), die die Klasse `org.omg.PortableServer.Servant` um das Interface `org.omg.CORBA.portable.InvokeHandler` erweitern. Die Operation `_invoke` wird innerhalb des Request-Processor-Threads aufgerufen. Der POA kann aber ebenso Aufrufe auf DSI-basierten Servants durchführen.

Servant-Manager Bei einem Servant-Manager handelt es sich um ein vom Anwendungsentwickler bereitgestelltes *locality constrained object* (siehe Abschnitt 4.8.2). Dies setzt die Aktivierung eines Servant-Manager-Objektes bei einem weiteren POA voraus. Servant-Manager gibt es in zwei verschiedenen Ausprägungen, einem Servant-Activator, der das Interface `org.omg.PortableServer.ServantActivator` bereitstellt und die Inkarnation und die Etherealization eines Servants übernimmt sowie einem Servant-Locator, der das `org.omg.PortableServer.ServantLocator` Interface implementiert und einen Servant für die Dauer eines einzelnen Objektaufrufes zur Verfügung stellt (siehe Abschnitt 4.4.4).

Adapter-Activator Bei einem Adapter-Activator handelt es sich ähnlich wie bei einem Servant-Manager um ein vom Entwickler bereitgestelltes *locality contrained object* und muß deshalb bei einem weiteren POA aktiviert werden. Ein Adapter-Activator implementiert das Interface `org.omg.PortableServer.AdapterActivator` und ist für die POA-Erzeugung "bei Bedarf" zuständig (siehe Abschnitt 4.5).

Policies Alle Implementierungsklassen der POA-bezogenen Policies sind in dem Paket `jacorb.poa.policy` zusammengefaßt. Policy-Objekte werden über Fabrikmethoden der `POA`-Schnittstelle erzeugt. Der Wert einer Policy wird bei ihrer Erzeugung angegeben. Die bei der POA-Erzeugung übergebenen Policies werden mit einer `POA`-Instanz assoziiert, und deren Werte haben Einfluß auf das Verhalten des POAs (siehe Abschnitt 4.2).

Exceptions In dem Paket `jacorb.poa.except` sind alle POA-bezogenen Fehler- und Ausnameklassen enthalten. Sie werden von der POA-Implementierung verwendet, um POA-interne Fehlerzustände (`POAInternalError`) oder Fehlanwendungen der POA-Klassen (`ApplicationError`) anzuzeigen. Diese Fehler können auch nicht behoben werden und führen zum Abbruch der Serveranwendung. Demgegenüber stehen Ausnahmen, die bestimmte POA-interne Ereignisse anzeigen. Sie bedürfen einer gesonderten Behandlung, führen aber keinesfalls zu einem Programmabbruch. Die Ursache für derartige Ereignisse können das Erreichen von Ressourcengrenzen oder die Änderung des POA-Zustandes sein, die es einem POA bsw. nicht mehr ermöglichen, weitere Anfragen zu bearbeiten. Diese eigenen Exceptions werden meist in geeignete System-Exceptions übersetzt und dem Client übermittelt.

Serviceklassen

Das Paket `jacorb.poa.util` stellt einige für die POA-Implementierung hilfreiche Klassen zusammen. In der Klasse `POAUtil` sind ausschließlich Hilfsfunktionen versammelt, die an vielen Stellen innerhalb der POA-Implementierung verwendet werden. Beispielsweise sind in dieser Klasse die String-zu-Objekt-ID und Objekt-ID-zu-String Konvertierungsfunktionen enthalten. Alle Methoden werden als öffentliche statische Operationen implementiert. Die Klasse `IdUtil` stellt der POA-Implementierung die Basisfunktionalität zur Erzeugung, Bearbeitung und zum Vergleich von Objekt-IDs zur Verfügung. Auch diese Klasse versammelt ausschließlich öffentliche Klassenmethoden. Das Interface `LogTrace` und eine seiner Implementierungen, die `LogWriter`-Klasse, ist ebenfalls in diesem Paket enthalten. Sie stellen POA-Komponenten eine Schnittstelle für die Protokollierung von Ereignissen zur Verfügung. `LogWriter`-Objekte sorgen für eine einheitliche Formatierung der Ausgabedaten.

JacORB-Schnittstellenklassen

In diesem Abschnitt geht es um die Beschreibung der für die POA-Implementierung relevanten JacORB-Schnittstellen. Hier wird nicht auf die volle Funktionalität dieser Systemkomponenten eingegangen, sondern es werden nur die für den POA relevanten Funktionen herausgestellt.

jacorb.orb.LocalityConstrainedObject Alle Implementierungen von IDL-Schnittstellen, die als *"locality and invocation constrained"* spezifiziert sind, erweitern diese abstrakte Klasse. Sie implementiert die IDL-Schnittstelle `CORBA::Object`, wobei alle Operationen eine **NO_IMPLEMENT** System-Exception werfen. Außerdem wird die Methode `toString` von `java.lang.Objekt` überschrieben, so daß sie eine **MARSHAL** System-Exception wirft. Damit können alle Implementierungsklassen derartig spezifizierter Schnittstellen das generierte Java-Interface direkt implementieren und brauchen selbst keine Implementierung der Operationen von `CORBA::Object` bereitstellen (POA, POA-Manager, Policies).

jacorb.orb.ORB Die `ORB`-Implementierung unterstützt einen POA bei der Erzeugung und Verwaltung von Objektreferenzen. In allen Operationen, bei denen der POA eine Objektreferenz zurückgeben muß, wird diese beim `ORB`-Objekt ermittelt. Der POA stellt den generierten Objekt-Key und die Repository-ID zur Verfügung. Die POA-Implementierung unterstützt den ORB beim Aufruf von registrierten *Request-* und *Object-Level-Interceptors*. Ist die Verwendung der entsprechenden *Interceptors* für einen Serverprozeß eingeschaltet, wird dem ORB kurz vor und nach dem eigentlichen Aufruf auf dem Servant das aktuelle `ServerRequest`-Objekt zur Bearbeitung übergeben. Die `ORB`-Klasse stellt dafür die Methoden bereit. Jede `POA`-Instanz ist mit einer `ORB`-Instanz assoziiert.

jacorb.orb.BasicAdapter Diese Klasse gehört inhaltlich zur ORB-Implementierung und implementiert das serverseitige Verbindungsmanagement. Als Schnittstelle zur POA-Implementierung übernimmt sie die ORB-seitige Kommunikation zur Vermittlung eines Objektaufrufes an den zuständigen POA. Nach dessen Ermittlung übergibt sie ihm ein `ServerRequest`-Objekt. Die POA-Implementierung benachrichtigt den ORB über die Fertigstellung der Anfragebearbeitung. Dazu ermittelt sie über die `ORB`-Instanz den Basic-Adapter und übergibt ihm das aktuelle `ServerRequest`-Objekt als Parameter einer Operation. Ein weiterer Berührungspunkt mit dem Basic-Adapter ergibt sich aus einer provisorische Auslieferung eines `ServerRequest`-Objektes an einen POA. `ServerRequest`-Objekte können unter Umständen nicht gleich an den zuständigen POA ausgeliefert werden. In diesem Fall übernimmt ein übergeordneter POA die Pufferung des `ServerRequest`-Objektes. Später wird dieser Sachverhalt erkannt und die Auslieferung des `ServerRequest`-Objektes durch die Übergabe an den Basic-Adapter erneut vorgenommen. Der Kontakt zum Basic-Adapter wird vom Request-Controller hergestellt.

jacorb.orb.Delegate Diese Klasse erweitert die im Language-Mapping definierte abstrakte Klasse `org.omg.CORBA.portable.Delegate` um die JacORB-spezifische Funktionalität einer Objektreferenz-Implementierung. Unter anderem speichert sie den Objekt-Key, der den qualifizierten POA-Namen und die Objekt-ID enthält. An verschiedenen Stellen innerhalb der POA-Implementierung muß die Objekt-ID aus einer Objektreferenz extrahiert werden, um beispielsweise eine Objektreferenz auf einen Servant abzubilden. Nach der Ermittlung des `Delegate`-Objektes kann von diesem der Objekt-Key erfragt werden. Der POA hat nun die Möglichkeit, aus diesem Objekt-Key die Objekt-ID zu extrahieren.

jacorb.orb.Environment Diese Klasse bietet eine Sammlung von öffentlichen Klassenmethoden für den Zugriff auf Umgebungsparameter. Durch die Verwendung von festgelegten Java-Properties können bestimmte Systemparameter von außen, ohne die Veränderung von Implementierungscode, für jeden einzelnen Serveranwendungsprozeß konfiguriert werden. Außer der Abfrage von Systemparametern stellt diese Klasse auch noch Protokollierungsoperationen zur Verfügung, die JacORB-weit für die Ausgabe von Systemmeldungen genutzt werden.

6.2.2. Modell-Architektur

Dieser Abschnitt beschreibt die Architektur des Implementierungsmodells, die Interaktionen und Beziehungen zwischen den Komponenten der POA-Implementierung. Die im vorherigen Abschnitt übersichtsartig skizzierten, einzelnen Bestandteile der POA-Implementierung werden somit in einen Gesamtzusammenhang gebracht. Die getroffenen Entwurfsentscheidungen werden an gegebener Stelle diskutiert und die Existenz bestimmter Komponenten motiviert. Zum besseren Verständnis ist in Abbildung 6.1 ein Klassendiagramm angegeben. Auf die Darstellung der Vererbungsbeziehungen im Zusammenhang mit der Implementierung von IDL-Schnittstellen, auf die Paketnamen und die Angabe von Attributen und Operationen habe ich aus Gründen der Übersicht verzichtet. Außerdem sind in dieser Darstellung die Beziehungen zur Log-Schnittstelle (`jacorb.poa.util.LogTrace`) und für den Zugriff auf Systemparameter (`jacorb.orb.Environment`) vernachlässigt. Nach der Zeichnung des Gesamtbildes wird dann später auf einzelne Aspekte der Implementierung näher eingegangen.

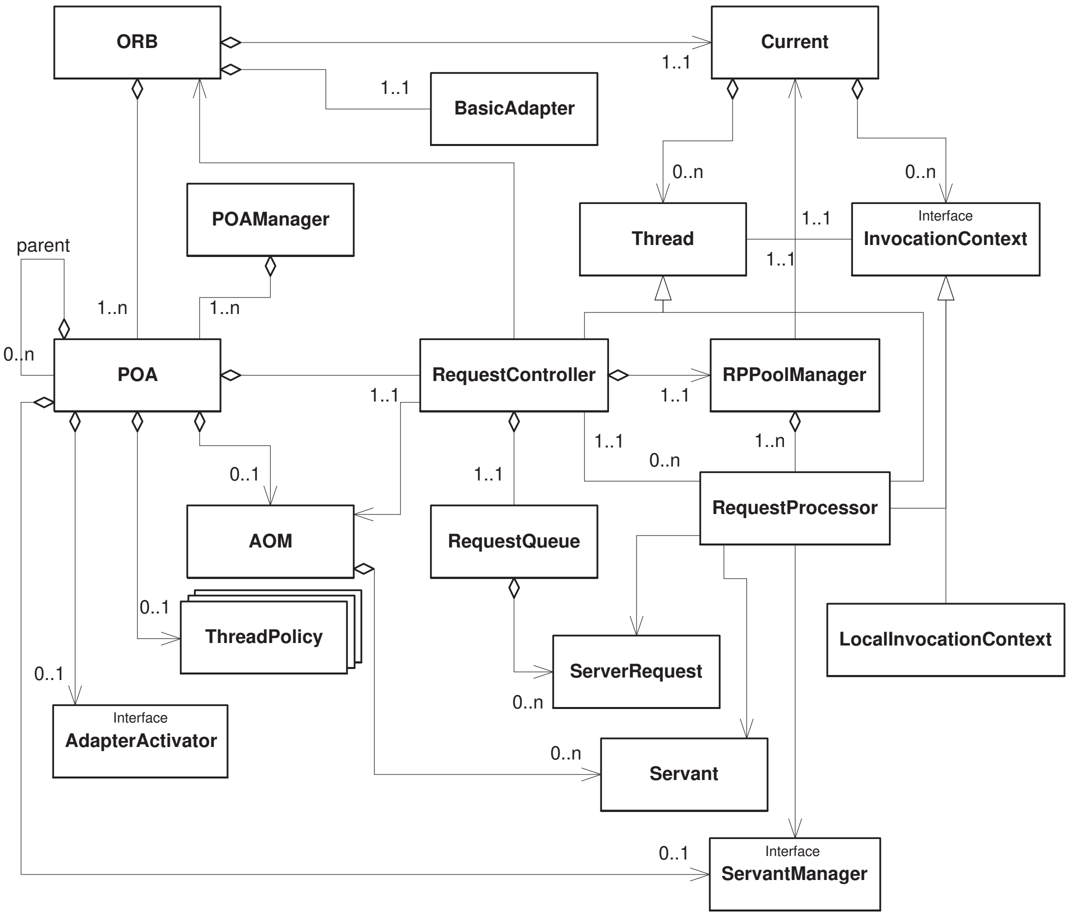


Abbildung 6.1.: Klassendiagramm der POA-Implementierung

Gesamtbild

Alle in einer JacORB-Serveranwendung verwendeten POAs, einschließlich dem Root-POA, sind Instanzen der Klasse `jacorb.poa.POA`. Neben der IDL-Schnittstelle für den POA wird die JacORB-interne Schnittstelle zur Kommunikation mit dem ORB von der selben Klasse implementiert, um den JacORB-internen Zugriff auf POA-Funktionen zu vereinheitlichen. Damit müssen andere Teile des ORBs nur Instanzen dieser Klasse referenzieren, um auf POA-Funktionen zugreifen zu können. Zusätzliche Komponenten, die POA-Funktionalität realisieren, bleiben dem ORB verborgen. Innerhalb einer Serveranwendung existiert genau eine Root-POA-Instanz, die der Entwickler über die ORB-Funktion `resolve_initial_references("RootPOA")` in Erfahrung bringen kann. Alle `POA`-Instanzen innerhalb einer POA-Hierarchie werden bei ihrer Erzeugung mit derselben ORB-Instanz assoziiert. Der Anwendungsentwickler kann weitere POA-Instanzen nur mit Hilfe der Operation create_POA auf einem schon vorhandenem POA-Objekt erzeugen. Ein neues POA-Objekt referenziert seinen Eltern-POA, und demgegenüber speichert ein Eltern-POA alle Referenzen auf die von ihm erzeugten Kind-POAs. Auf den POA-Erzeugungsprozeß wird weiter unten noch detailliert eingegangen.

Jedes POA-Objekt hält eine Referenz auf eine POAManager-Instanz. Diese kann ihm bei seiner Erzeugung übergeben werden. Ist dies nicht der Fall, wird eine neue Instanz der Klasse jacorb.poa.POAManager erzeugt und verwendet. Damit lassen sich mehrere POA-Objekte einem POAManager-Objekt zuordnen. Um Zustandswechsel, wie in der Spezifikation vorgegeben, bearbeiten zu können, reicht es nicht aus, daß jedes POA-Objekt seinen POA-Manager referenziert. Ein POAManager-Objekt muß auch die mit ihm assoziierten POAs kennen, also eine entsprechende Struktur verwalten und eine paket-interne Schnittstelle zum An- und Abmelden von POA-Objekten zur Verfügung stellen. Auf diese Weise wird vermieden, daß POA-Objekte ihren Zustand aktiv und in regelmäßigen Abständen erfragen müssen. Außerdem wird ein POA-Objekt bei seiner Erzeugung mit einem Array von Policy-Objekten parametrisiert. Diese werden sofort kopiert, und nur diese Kopien werden von einem POA referenziert. Policy-Objekte sind Instanzen der im jacorb.poa.policy Paket versammelten Implementierungsklassen. Ein POA-Objekt verwendet bis zu sieben verschiedene Policy-Objekte. Policy-Objekte werden aber nur für den Fall referenziert, wenn ihr Wert vom jeweiligen Standardwert abweicht. Für nicht referenzierte Policies werden die Standardwerte angenommen. Dieses Verhalten ist in verschiedenen *Convenience*-Operationen der paket-internen POA-Schnittstelle gekapselt, die auch von anderen Komponenten genutzt werden, um die Policy-Werte für einen POA zu erfragen. Jeder POA, dessen ServantRetentionPolicy den Wert **RETAIN** besitzt, benutzt eine AOM für die Abbildung von Objekt-IDs auf Servants. Die Kapselung dieser Datenstruktur mit den darauf arbeitenden Operation in einer eigenen Klasse fördert die Übersichtlichkeit und Flexibilität der Implementierung, zumal die Operationen besondere Synchronisationsmaßnahmen erfordern.

Die Aufgaben der Anfragebearbeitung werden an ein RequestController-Objekt delegiert. Damit ist ein wesentlicher Teil der POA-Funktionalität ausgelagert, und ein POA-Objekt nimmt nur noch durch die Bereitstellung von Informationen und bei bestimmten Ereignissen, wie z.B. einem Zustandswechsel, Einfluß auf die Anfragebearbeitung. Durch die Auslagerung der Funktionalität für die Anfragebearbeitung wird die POA-Klasse entscheidend entlastet, was zu einer größeren Übersichtlichkeit beiträgt. Die im Zusammenhang mit der Anfragebearbeitung auftretenden Synchronisationsprobleme können in der RequestController-Klasse gekapselt werden. Es wird ein eigener Thread zur Verfügung gestellt, der das Auslesen der anstehenden Anfragen aus der Request-Queue übernimmt und die Bearbeitung anstößt. Damit wird die Übername eines Requests durch den POA von der eigentlichen Bearbeitung entkoppelt. Der Request-Controller verschafft sich vom POA die nötigen Informationen zum aktuellen Zustand, den geltenden Policies und einem eventuell vorhandenen ServantManager-Objekt. Weiterhin besitzt er Zugriff auf die AOM, um den zuständigen Servant für eine Anfrage ermitteln zu können und auf den mit dem POA assoziierten ORB, um den Basic-Adapter zu erreichen.

Für die Anfragebearbeitung werden weitere Komponenten benötigt, ein RequestQueue-Objekt, ein Pool von RequestProcessor-Objekten und ein RPPool-

`Manager`-Objekt. Diese brauchen nur dem Request-Controller bekannt zu sein und nicht dem POA. Jeder Request-Controller benutzt ein `RequestQueue`-Objekt, um die ihm übergebenen `ServerRequest`-Objekte zu puffern. Auch hier sind wiederum Übersichtlichkeit und die damit einhergehende elegantere Lösung der Synchronisationsprobleme sowie die größere Flexibilität der Implementierung die Motivation für die Auslagerung in eine separate Klasse. Jedes `RequestQueue`-Objekt referenziert auch seinen Request-Controller und benachrichtigt diesen, wenn neue Anfragen zur Bearbeitung anstehen.

Für die eigentliche Abwicklung des Methodenaufrufes auf einem Servant wird ein weiterer Thread erzeugt, der Request-Processor-Thread. Dazu erweitert die Klasse `jacorb.poa.RequestProcessor` die Java-Klasse `java.lang.Thread`. Ein Request-Processor stellt die Ablaufumgebung für die Aufrufe auf dem Servant bzw. dem Servant-Manager dar. Um nicht für jede Anfrage einen neuen Thread erzeugen zu müssen, sind in einer POA-Umgebung gewöhnlich gleich eine ganze Reihe solcher `RequestProcessor`-Objekte vorhanden, die fertig gestartet von einem Pool-Manager verwaltet werden. Nach ihrer Verwendung gelangen sie wieder zurück in den Pool und können für einen neuen Aufruf benutzt werden. Wenn für die `ThreadPolicy` eines POAs der Wert **ORB_CTRL_MODEL** gilt, wird von einem Request-Controller ein eigener Pool-Manager, also eine Instanz von `jacorb.poa.RPPoolManager` erzeugt und verwendet. Alle anderen POA-Umgebungen, die mit der `ThreadPolicy` und dem Wert **SINGLE_THREAD_MODEL** parametrisiert wurden, benutzen eine gemeinsame Instanz eines Pool-Managers, der in diesem Fall nur ein einziges `RequestProcessor`-Objekt zur Verfügung stellt. Da der Request-Controller-Thread im wesentlichen in einer Schleife das Auslesen des nächsten Requests aus der Queue übernimmt, sich vom Pool-Manager einen freien Request-Processor holt, diesen initialisiert und startet, wird die eigentliche Anfragebearbeitung des Request-Processors wiederum vom Request-Controller-Thread entkoppelt. Demzufolge können mehrere Anfragen an einen POA gleichzeitig aktiv sein. Auf weitere Details zur Verwaltung des Request-Processor-Pools wird später noch genauer eingegangen.

`RequestProcessor`-Objekte kooperieren mit ihrem Pool-Manager. Sie sind nur für die Abarbeitung eines Aufrufes aktiv und melden sich nach dessen Fertigstellung wieder beim Pool-Manager zurück, bevor sie einen weiteren Auftrag, möglicherweise für einen anderen Servant, erledigen können. Bei der Initialisierung des `RequestProcessor`-Objektes durch den Request-Controller zur Durchführung eines Auftrages wird diesem eine Referenz des aktuellen Request-Controllers, das aktuelle `ServerRequest`-Objekt, und der verantwortliche Servant übergeben. Sollte im Vorfeld vom Request-Controller kein verantwortlicher Servant ermittelt werden können, wird an dessen Stelle ein eventuell vorhandener Servant-Manager übergeben. Außerdem leistet ein Request-Processor Unterstützung beim Aufruf von serverseitigen *Request-* bzw. *Message-Level-Interceptors*. Die dafür notwendige `ORB`-Instanz kann er über seinen Request-Controller ermitteln. Nach der Beendigung der Bearbeitung einer Anfrage meldet sich ein Request-Processor auch bei seinem Request-Controller zurück und händigt ihm das `ServerRequest`-Objekt wieder aus. Auf den genauen Ablauf einer Anfragebearbeitung wird später noch im Detail eingegangen, so daß ich es an dieser Stelle bei der Beschreibung der Beziehungen belasse, die ein `RequestProcessor`-Objekt für seine Arbeit benötigt. Der Request-Controller benachrichtigt den Basic-Adapter über die Fertigstellung der Anfrage.

Zugriff auf den Aufrufkontext

Auf den Aufrufkontext kann mit Hilfe des `Current`-Objektes zugegriffen werden. Das `Current`-Objekt wird als *Singleton* implementiert, d.h. innerhalb einer Serveranwendung existiert nur eine einzige Instanz der Klasse `jacorb.poa.Current`. Sie kann jederzeit über die ORB-Funktion `resolve_initial_references` mit dem Parameter "`POACurrent`" oder die JacORB-interne Klassenmethode `_Current_init` der `Current`-Klasse ermittelt werden.

Das `Current`-Objekt verwaltet in einer Tabelle zusammen mit einem `InvocationContext`-Objekt die in Ausführung befindlichen Aufrufthreads und stellt die Operationen zum An- und Abmelden dieser Assoziationen bereit. Bei einer Anfrage wird der aktuelle Thread und das assoziierte `InvocationContext`-Objekt ermittelt, das die Kontext-Informationen bereitstellt. Das `InvocationContext`-Interface wird von der `RequestProcessor`-Klasse und der `LocalInvocationContext`-Klasse implementiert.

Im Falle eines vom ORB vermittelten Fernaufrufes findet der Servant-Aufruf innerhalb des Request-Processor-Threads statt. Der Pool-Manager meldet einen neuen Request-Processor-Thread zusammen mit dem `RequestProcessor`-Objekt bei seiner Erzeugung beim `Current`-Objekt an und bei einer späteren Freigabe wieder ab. Für den lokalen Fall wird ein `LocalInvocationContext`-Objekt vom Delegate eines lokalen Stubs erzeugt, initialisiert und zusammen mit dem aktuellen Ausführungsthread beim `Current`-Objekt registriert und nach der Beendigung des Aufrufes wieder abgemeldet.

Konfiguration eines Serverprozesses

Die Klasse `jacorb.orb.Environment` bietet eine öffentliche Schnittstelle von Klassenmethoden für den Zugriff auf Umgebungsparameter. Durch die Verwendung von festgelegten Java-Properties können bestimmte Systemparameter von außen, ohne die Veränderung von Implementierungscode, für jeden einzelnen Serveranwendungsprozeß konfiguriert werden. Die für die POA-Implementierung relevanten Systemparameter sind im folgenden aufgeführt.

- `jacorb.implname` - Servername (korrespondierend mit Eintrag im ImR)
- `jacorb.reference_caching` - Schalter für das *Caching* von Objektreferenzen
- `jacorb.verbosity` - Protokollierungsstufe
- `jacorb.logfile` - Logdatei für Protokollierungsausgaben
- `jacorb.interceptor.server.Messages` - Schalter für *Message-Level-Interc.*
- `jacorb.interceptor.server.Requests` - Schalter für *Request-Level-Interc.*
- `jacorb.interceptor.perObject` - Schalter für *Object-Level-Interceptors*
- `jacorb.poa.thread_pool_max` - Maximum verfügbarer Request-Processors
- `jacorb.poa.thread_pool_min` - Minimum verfügbarer Request-Processors
- `jacorb.poa.queue_max` - Maximum pufferbarer Server-Requests

Request-Processor-Pool-Management

Die Anzahl der vorgehaltenen `RequestProcessor`-Objekte kann über Umgebungsparameter für eine Serveranwendung konfiguriert werden. Zumindest gelten dieser Parameter für einen *multi-threaded* POA (**ORB_CTRL_MODEL**). Im *single-threaded* Modus wird grundsätzlich nur ein `RPPoolManager`-Objekt verwendet, das nur einen einzigen Request-Processor zur Verfügung stellt. Diesen einen Request-Processor müssen sich alle *single-threaded* POAs innerhalb einer Serveranwendung für ihre Aufrufe auf Servants und Servant-Managers teilen.

Falls die verfügbaren `RequestProcessor`-Objekte ausgehen, können bis zu einer, wiederum von einem Umgebungsparameter festgelegten Obergrenze, einzelne Request-Processor-Threads nachgestartet und ausgegeben werde. Sollte auch diese Obergrenze erreicht werden, kann nur dann ein Request-Processor ausgegeben werden, wenn zwischenzeitlich wieder ein solcher zurückgegeben wurde. Wenn die Anzahl der im Pool befindlichen `RequestProcessor`-Objekte wieder die Mindestanzahl erreicht hat, werden alle weiteren zurückgegebenen Request-Processor-Threads beendet. Standardmäßig werden als Untergrenze 10 und als Obergrenze 20 `RequestProcessor`-Objekte verwendet.

Log-Mechanismus

Jede Komponente der POA-Implementierung, die Protokollausgaben wie Status- oder Fehlermeldungen machen muß, hat Zugriff auf ein `LogTrace`-Objekt. Dafür wird eine Instanz der Klasse `jacorb.poa.util.LogWriter` verwendet. Der eigentliche Gewinn dieser zusätzlichen Protokollierungsschnittstelle liegt in einer einheitlichen, an die POA-Implementierung angepaßten Formatierung der Ausgabe. Andere Implementierungsklassen müssen sich mit diesen Details nicht beschäftigen. Bei der Ausgabe der Systemmeldungen stützt sich ein `LogWriter`-Objekt auf eine weitere, JacORB-weit verwendete Schnittstelle. Es werden die Ausgabeoperationen der Klasse `jacorb.orb.Environment` benutzt. Damit sind zusätzliche Vorteile verbunden, wie ein JacORB-weit einheitlicher Ausgabepfad für Protokolldaten und eine Filterfunktion, die nicht gewollte Ausgaben unterbindet. Der Ausgabepfad ist über Umgebungsparameter konfigurierbar. Z.Zt. ist die Protokollierung über die Standardausgabe oder in eine Logdatei vorgesehen. Ein weiterer Vorteil liegt in der JacORB-weit einheitlichen Formatierung der Ausgabe.

Protokollausgaben werden in JacORB nach ihrer Wichtigkeit eingestuft. Die Einstufung für eine Ausgabe wird beim Operationsaufruf als Parameter festgelegt und folgt einer Konvention, über die sich die Entwickler von JacORB geeinigt haben. Die für einen Serverprozeß geltende Protokollierungsstufe wird ebenfalls über einen Umgebungsparameter konfiguriert. Die Anzahl der ausgegebenen Meldungen kann so stufenweise variiert werden. Das Spektrum reicht von "Keine Ausgabe" bis hin zu einer sehr ausführlichen Protokollierung, dem sog. "Debug-Modus".

6.2.3. POA-Erzeugung

Innerhalb einer Serveranwendung existiert genau eine Root-POA-Instanz. Die JacORB-interne Schnittstelle des POAs enthält eine Klassenmethode `_POA_init`, mit der der ORB diese Instanz erzeugen bzw. erfragen kann. Der Root-POA wird mit der übergebenen `ORB`-Instanz, einem neu erzeugten `POAManager`-Objekt und einem

`ImlicitActivationPolicy`-Objekt initialisiert. Die Referenz auf den Eltern-POA wird auf den Wert **null** gesetzt. Alle anderen Maßnahmen zur Initialisierung sind nicht Root-POA-spezifisch und werden durch den Aufruf des selben Konstruktors geleistet.

Der Anwendungsentwickler kann weitere `POA`-Instanzen nur mit Hilfe der Fabrikmethode `create_POA` auf einer schon vorhandenen `POA`-Instanz erzeugen. Die direkte Verwendung von Konstruktoren ist nicht möglich. Bei der POA-Initialisierung wird ein Wasserzeichen generiert, das den POA eindeutig identifiziert und zur Objekt-ID-Erzeugung verwendet wird (siehe Abschnitt 6.2.4). Prinzipiell werden bei der Erzeugung eines POAs alle notwendigen Objekte und Strukturen im voraus angelegt. Eine Ausnahme macht der Pool-Manager, der seinen Request-Processor-Pool erst dann initialisiert, wenn der erste Request-Processor vom Request-Controller angefordert wird. Dem Nachteil erhöhter Kosten für die Bearbeitung der ersten Anfrage stehen die eingesparten Ressourcen gegenüber, falls ein POA gar keine Anfrage bearbeiten muß. Zum Abschluß wird der neu erzeugte Kind-POA in eine Kind-POA-Tabelle des Eltern-POAs eingetragen.

Synchronisationsmaßnahmen

Die Erzeugung eines benutzerdefinierten POAs kann explizit innerhalb der Serveranwendung durch den Aufruf der Operation `create_POA` oder implizit, als Folge einer Anfrage an einen noch nicht existierenden POA erfolgen. Für diesen letzteren Fall muß ein Adapter-Activator beim erzeugenden Eltern-POA angemeldet sein. Der Programmcode zur Erzeugung eines POAs bleibt aber in dieser einen Methode gekapselt, denn auch die Implementierung der `unknown_adapter()` Operation des Adapter-Activators, die vom Anwendungsentwickler zur Verfügung gestellt wird, muß letztendlich diese Methode benutzen. Nun ist es durchaus möglich, daß sich mehrere Erzeugungsprozesse für den gleichen POA (einen Kind-POA mit gleichem Namen) überlappen. Auch läßt sich die Tatsache nicht vermeiden, daß in einer *multi-threaded* Serveranwendung mehrere `create_POA()` Aufrufe für den gleichen POA gestartet werden. Eine andere denkbare Konfliktsituation resultiert aus einem möglichen Aufruf der `destroy()` Operation verbunden mit nebenläufigen Erzeugungsoperationen für einen Kind-POA. Sind diese Annahmen auch eher theoretischer Natur, müssen doch Synchronisationsmaßnahmen getroffen werden, um einen korrekten Ablauf des Erzeugungsprozesses sicherzustellen. Auf den POA-Zerstörungsprozeß wird weiter unten genauer eingegangen.

Die POA-Erzeugung bzw. -Zerstörung wird über einen speziellen Monitor synchronisiert. Innerhalb dieses Monitors werden folgende Aktionen serialisiert ausgeführt:

- der Test, ob der übergebene Adaptername schon vorhanden und wenn ja, ob die Zerstörung eines registrierten Kind-POAs schon "sichtbar" ist,
- die komplette Erzeugung und Initialisierung eines POAs,
- die Eintragung des neu erzeugten Kind-POAs in die Kind-POA-Tabelle,
- das Abmelden eines Kind-POAs bei seinem Eltern-POA nach der vollständigen Zerstörung des Kind-POAs und die Benachrichtigung wartender Erzeugungsprozesse.

Nebenläufige Aufrufe, auch mit unterschiedlichen Adapternamen, laufen dadurch serialisiert ab. Die komplette Serialisierung wäre zwar nicht notwendig, erspart aber die Verwendung einer zusätzlichen Struktur zur Protokollierung schon gestarteter POA-Erzeugungsprozesse. Ein POA, dessen Zerstörungsprozeß schon begonnen hat, darf keine Kind-POAs mehr erzeugen. Ein entsprechender Test findet ebenfalls in dem oben beschriebenen Monitor statt, ebenso die Bekanntmachung, daß der Zerstörungsprozeß eines POAs beginnt. Ein POA, dessen Zerstörung schon "sichtbar" ist, kann unter gleichem Namen neu erzeugt werden. Ein nebenläufiger Erzeugungsprozeß wartet auf die vollständige Zerstörung und wird über deren Beendigung informiert. Da der Monitor die komplette POA-Erzeugung einschließt, kann die Zerstörungsoperation nur auf vollständig erzeugte und initialisierte Adapter angewendet werden.

Durch die getroffenen Synchronisationsmaßnahmen ist zwar ein definiertes Verhalten sichergestellt, Anwendungen sollten aber mit möglichen Fehlern bei einer manuellen oder automatischen POA-Erzeugung umgehen können.

6.2.4. Erzeugung und Verwaltung von Objektreferenzen

Die `ORB`-Implementierung unterstützt einen POA bei der Erzeugung und Verwaltung von Objektreferenzen. Da Netzwerkverbindungen vom ORB verwaltet werden, sind nur diesem die Verbindungsdetails zur serverseitigen ORB-Umgebung bekannt. Speziell der Basic-Adapter stellt den Kommunikationsendpunkt für eine Serveranwendung bereit. Der POA ist für die Zusammenstellung des Objekt-Keys verantwortlich und übergibt diesen, zusammen mit einer Referenz auf sich selbst, einer `getReference()` Operation des ORBs. Die Zusammensetzung eines in JacORB verwendeten Objekt-Keys ist in der Abbildung 6.2 dargestellt.

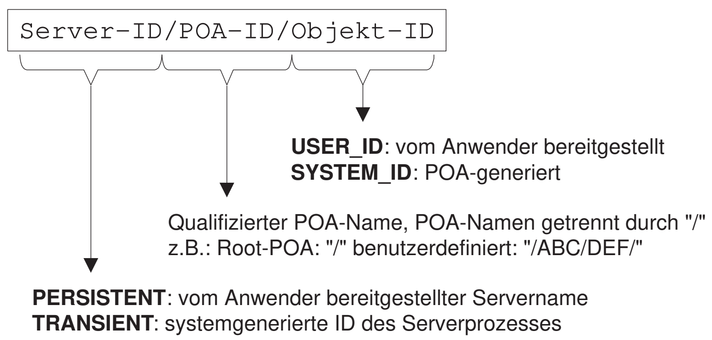


Abbildung 6.2.: Zusammensetzung eines Objekt-Keys

Die Erreichbarkeit persistenter Objekte wird mit Hilfe einer von JacORB zur Verfügung gestellten Implementierung eines *Implementation Repositories* (ImR) und eines *Server Startup Daemons* (SSD) sichergestellt. Der Anwender hat die Möglichkeit, die serverseitige ORB-Umgebung so zu konfigurieren, daß ein permanent laufender ImR-Prozeß bereitsteht, der Anfragen an Objekte entgegennimmt und mit Hilfe eines ebenfalls laufenden SSD den verantwortlichen Serverprozeß startet, wenn noch kein Server für die Bearbeitung der Anfrage aktiv ist. Die IOR eines persistenten

Objektes, das diesen Service nutzen möchte, muß anstelle der Adreßinformationen des Serverprozesses, die natürlich nur solange gültig sein können, solange der Serverprozeß existiert, die Adreßinformationen des ImRs enthalten. Bei Verfügbarkeit des ImRs werden vom ORB automatisch und transparent für den POA die Adreßinformationen des ImRs für die Erzeugung einer IOR eines persistenten Objektes benutzt. Erst nach einem erfolgreichen Start des Servers kann die Objektreferenz direkt an den Serverprozeß gebunden werden.[1]

Um Serverprozesse bei Bedarf starten zu können, muß das für persistente Objekte verantwortliche Serverprogramm im ImR registriert sein (Informationen zum Starten des Serverprozesses). Zur Registrierung von Serverprogrammen kann der Anwender einen logischen Servernamen angeben. Für den Start des verantwortlichen Servers wäre eine Beschränkung auf den vollständigen POA-Namen unzureichend, da von einem ImR mehrere Serverprozesse betreut werden können (mit potentiell gleichen POA-Namen). Diese Server-ID identifiziert alle von einem Server implementierten persistenten Objekte. Für die Eindeutigkeit der vergebenen Server-IDs ist der Anwender verantwortlich.

Der Serverprozeß, der persistente Objekte bereitstellt, muß ebenfalls mit dieser Server-ID, korrespondierend zu dem Eintrag im ImR, konfiguriert werden. Sie wird beim Start des Servers als Java-Property angegeben. Ein persistenter POA unterstützt den bedarfsgerechten Serverstart, indem er die Server-ID in jeden neu erzeugten Objekt-Key aufnimmt. Sobald eine Serveranwendung einen persistenten POA erzeugen will, wird das Vorhandensein dieser Property geprüft, da die Server-ID für die spätere Erzeugung von Objektreferenzen zwingend benötigt wird. Wenn keine Server-ID konfiguriert wurde, bricht die Anwendung mit einer Fehlermeldung ab. Das ImR hat nun die Möglichkeit, bei einer Anfrage die Server-ID aus dem Objekt-Key zu extrahieren und den registrierten Serverprozeß zu starten.

Für den Objekt-Key eines transienten Objektes wird ebenfalls eine Server-ID verwendet, die aber vom Serverprozeß selbst generiert wird und diese Server-Instanz eindeutig identifiziert. Sie hat aber keine vergleichbare funktionale Bedeutung, da die Objektreferenz eines transienten Objektes nicht an das ImR gebunden wird.

Die Objekt-ID identifiziert ein Objekt innerhalb eines POAs eindeutig. Der POA selbst verwendet zur Verwaltung der Objekte nur die Objekt-ID und muß eine Objektreferenz nur bei Bedarf erzeugen, z.B. bei bestimmten Operationen der `POA`-Schnittstelle, die anstelle von Objekt-IDs mit Objektreferenzen operieren. Die Zusammensetzung einer vom POA generierten Objekt-ID wird weiter unten beschrieben.

Für einen von JacORB erzeugten Objekt-Key wurde eine Repräsentation gewählt, die sich an einer URL orientiert. Der Server-ID folgen die einzelnen POA-Namen, ausgehend von einem Kind-POA des Root-POAs, und die Objekt-ID des Objektes. Alle Teilkomponenten eines Objekt-Keys werden einheitlich durch einen *Slash* ("`/`") getrennt. Jedes Auftreten dieses Trennzeichens innerhalb der Teilkomponenten wird maskiert. Somit existieren keine Einschränkungen bei der Auswahl der Zeichen bzw. Bytes für die einzelnen Teile des Objekt-Keys. Von der Klasse `jacorb.poa.util.POAUtil` werden Hilfsfunktionen zur (De)Maskierung und zur Extrahierung der Teilkomponenten eines Objekt-Keys bereitgestellt.

[1] Mit Hilfe des Location-Forward-Mechanismus können Stubs an einen anderen Serverprozeß gebunden werden. Wenn ein Objekt später nicht mehr erreichbar ist, sorgt ein Fallback-Mechanismus für die Wiederherstellung der ursprünglichen Adreßinformationen.

Standardmäßig verwendet JacORB für Objektreferenzen Instanzen der Klasse `jacorb.orb.Reference`, die erst bei ihrer Verwendung (Aufruf der `narrow()` Operation der entsprechenden Helper-Klasse) durch eine Instanz der generierten Stub-Klasse ersetzt werden. Als Delegate wird eine Instanz der Klasse `jacorb.orb.Delegate` verwendet.

Der ORB kann das *Caching* von einmal erzeugten Objektreferenzen übernehmen. Ob der ORB ein *Caching* durchführt, wird für einen Serverprozeß über eine Java-Property konfiguriert. Bei eingeschaltetem *Caching* werden Objektreferenzen innerhalb einer Serveranwendung für einen einzelnen Objekt-Key nur einmal erzeugt und bei einer weiteren Anfrage mit gleichem Objekt-Key wieder herausgegeben.

Objekt-ID-Generierung

Der geltende Wert für die Lifespan-Policy bestimmt die Zusammensetzung einer POA-generierten Objekt-ID. Bei der POA-Erzeugung wird jede POA-Instanz mit einem Wasserzeichen initialisiert. Dieses enthält den Hashwert eines Strings, der sich aus der Server-ID und dem vollständigen POA-Namen zusammensetzt. Die Klasse `jacorb.poa.util.IdUtil` stellt Basisroutinen zur Objekt-ID-Erzeugung bereit. Die Zusammensetzung einer in JacORB verwendeten Objekt-ID ist in der Abbildung 6.3 dargestellt.

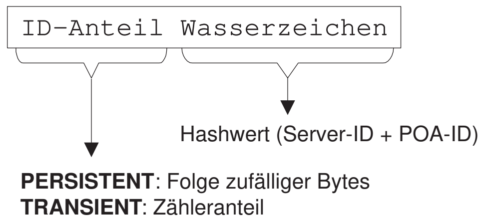


Abbildung 6.3.: Zusammensetzung einer POA-generierten Objekt-ID

Das Wasserzeichen geht in jede Objekt-ID ein. Damit wird sichergestellt, daß spätere Anfragen bzw. Operationsaufrufe, die eine Objekt-ID als Parameter enthalten, von einem POA nur dann bearbeitet werden, wenn er diese Objekt-ID auch generiert hat. Ein entsprechender Vergleich des POA-Wasserzeichens mit dem Wasserzeichenanteil in jeder Objekt-ID findet innerhalb der betroffenen Operationen statt. Dabei erfüllt die beschriebene Zusammensetzung des Wasserzeichens genau jene Anforderungen, die an einen transienten bzw. persistenten POA gestellt werden.

Im transienten Fall ist die Eindeutigkeit durch die Einbeziehung einer zufälligen Folge von Bytes für jedes neu erzeugte Wasserzeichen garantiert. Objekt-IDs, die nicht von dieser POA-Instanz generiert wurden, können somit zurückgewiesen werden. Für die eindeutige Identifizierung eines transienten Objektes innerhalb einer POA-Instanz ist in der Objekt-ID ein Zähleranteil enthalten, der vom POA mit jeder Objekterzeugung hochgezählt wird.

Bei einem persistenten POA muß für eine bestimmte Serveranwendung immer dasselbe Wasserzeichen generiert werden, die diesen POA, auch über mehrere Server-Instanzen hinweg, eindeutig identifiziert. Die vom Anwender bereitgestellte Server-ID darf sich somit natürlich nicht ändern. Objekt-IDs, die nicht von einem POA mit gleichem Namen und gleicher Server-ID generiert wurden, können somit zurückgewiesen werden. Da die von einem persistenten POA generierten Objekt-IDs über alle POA-Instanzen hinweg eindeutig sein müssen, geht in die Objekt-ID noch eine hinreichend lange Folge zufälliger Bytes ein.

6.2.5. Anfragebearbeitung

Die im vorigen Abschnitt beschriebene Erzeugung von Objektreferenzen bildet die Grundlage für die Zustellung von Objektaufrufen an die serverseitige ORB-Umgebung, das evtl. Starten eines Serverprozesses, die Identifizierung des zuständigen POAs und die Auslieferung an den verantwortlichen Servant. Ein *Listener-Thread* nimmt die Anfragen an die von einem Serverprozeß verwalteten Objekte entgegen. Danach startet er einen *Request-Receptor-Thread*, der die Verbindung zu einem einzelnen Client-Stub aufnimmt und zukünfig alle Anfragen von diesem Client-Stub einer Bearbeitung zuführt. Zunächst wird jedoch, nach dem Eingang einer Anfrage, eine Instanz der Klasse `jacorb.orb.dsi.ServerRequest` erstellt und mit der offenen Client-Server-Verbindung initialisiert.

Identifizierung des zuständigen POAs durch den ORB

Nach dem Empfang einer Anfrage durch den Basic-Adapter muß der zuständige POA ermittelt werden. Durch die Extraktion der einzelnen POA-Namen aus dem Objekt-Key kann der Pfad bis zum zuständigen POA, ausgehend vom Root-POA, durchlaufen werden. Jeder POA stellt über seine JacORB-interne Schnittstelle die Operation `_getChildPOA` zur Verfügung, der als Parameter der Adaptername eines Kind-POAs übergeben wird. Diese Operation, angewendet auf einen Eltern-POA, liefert die Instanz eines registrierten Kind-POAs zurück. Sollte kein Kind-POA unter diesem Namen registriert sein oder ein registrierter Kind-POA gerade zerstört werden (Zerstörung muß "sichtbar" sein), dann wird versucht, mit Hilfe eines Adapter-Activators und dem Aufruf der Operation `unknown_adapter` einen neuen POA zu erzeugen. Die Operation `_getChildPOA` kehrt erst wieder zurück, wenn die Erzeugung und Initialisierung des POAs abgeschlossen ist. Für diese Operation gibt es mehrere Fehlerzustände, auf die mit einer geeigneten System-Exception reagiert wird. Der Basic-Adapter fängt diese auftretenden Exceptions auf und sorgt für deren Rückübertragung zum Client.

Wenn sich der befragte Eltern-POA im Zustand **HOLDING** befindet, kann die Auslieferung des Requests trotzdem vorgenommen werden, wenn auch in einem abgewandelten Verfahren. Auf diesen speziellen Fall wird später gesondert eingegangen. Diese Situation wird dem Basic-Adapter mit einer **ParentIsHolding** Exception des `jacorb.poa.except` Paketes angezeigt. Der Normalfall während der Anfragebearbeitung wird aber sein, daß sich die auf dem Pfad zum zuständigen POA liegenden Eltern-POAs im Zustand **ACTIVE** befinden und jeweils ein Kind-POA ermittelt bzw. erfolgreich erzeugt werden kann. Somit hält der Basic-Adapter am Ende dieser Aufrufkette eine `POA`-Instanz in den Händen, der er das `ServerRequest`-Objekt übergeben kann.

Übernahme einer Anfrage durch den POA

Ein `ServerRequest`-Objekt wird dem zuständigen POA über seine JacORB-interne Schnittstelle durch den Aufruf der Operation `_invoke` ausgehändigt. Dieser Aufruf geschieht immer noch innerhalb des Request-Receptor-Threads. Um diesen Thread schnellstmöglich wieder für die Auslieferung einer neuen Anfrage freizugeben, müssen die Aktionen zur Übernahme des Requests durch den POA begrenzt und von der eigentlichen Anfragebearbeitung des POAs entkoppelt werden. Somit wird innerhalb dieses Threads nur das `ServerRequest`-Objekt an den Request-Controller übergeben, der es in die Request-Queue einfügt und den Request-Controller-Thread weckt, falls sich damit genau ein Element in der Queue befindet.

Auslesen der Request-Queue

Die weitere Bearbeitung der Anfrage übernimmt der Request-Controller-Thread. Er liest innerhalb einer Schleife die Request-Queue aus. Abhängig vom POA-Zustand wird entweder vom Beginn oder vom Ende der Queue gelesen. Die Zustände **INACTIVE** und **DISCARDING** verlangen die Entfernung des letzen Elements der Queue. Innerhalb des `ServerRequest`-Objektes wird daraufhin eine System-Exception gesetzt. Das Ende der Bearbeitung eines solchen Server-Requests durch den POA wird nun noch dem Basic-Adapter durch den Aufruf von `return_result()` und die Übergabe des `ServerRequest`-Objektes angezeigt. Auf diese Weise wird die Queue der Reihe nach abgeräumt, wobei die jüngsten Elemente der Queue zuerst an der Reihe sind. Vor jeder Entfernung eines Elements wird aber weiterhin der POA-Zustand abgefragt. Der an dieser Stelle ermittelte Zustand ist für einen einzelnen Schleifendurchlauf verbindlich. Der Request-Controller-Thread bestimmt somit den Takt, in dem vom POA-Manager initiierte Zustandsänderungen Einfluß auf die Anfragebearbeitung nehmen. Der Zustand **HOLDING**, eine leere Queue oder der Beginn des POA-Zerstörungsprozesses führen dazu, daß der Request-Controller-Thread blockiert und alle weiteren Anfragen in der Queue gepuffert werden. Nur im Zustand **ACTIVE** wird das erste Element aus der Queue gelesen und die Anfragebearbeitung fortgesetzt.

Zuweisung eines Request-Processors

Nachdem das aktuelle `ServerRequest`-Objekt aus der Queue gelesen wurde, müssen einige Vorbereitungen getroffen werden, um die Anfrage von einem freien Request-Processor bearbeiten lassen zu können. Als erstes wird geprüft, ob sich das Zielobjekt in seinem Deaktivierungsprozeß befindet. Dazu wird im Request-Controller eine Deaktivierungsliste verwaltet, die die Objekt-IDs der im Deaktivierungsprozeß befindlichen Objekte enthält. Liegt keine Deaktivierung vor, wird bei geltendem **RETAIN** Policy-Wert über die AOM versucht, den verantwortlichen Servant zu ermitteln. Ist kein Servant für diese Objekt-ID registriert oder liegt eine Objektdeaktivierung vor, wird ein eventuell vorhandener Default-Servant beim POA erfragt. Steht auch dieser nicht zur Verfügung, bleibt nur noch die Ermittlung eines Servant-Managers.

Entweder ein Servant oder ein Servant-Manager muß ermittelt werden können, damit die Anfragebearbeitung fortgesetzt werden kann. Ist dies der Fall, sind die Voraussetzungen für die Bereitstellung eines Request-Processors erfüllt. Vorher muß aber noch das `ServerRequest`-Objekt unter seiner Objekt-ID in eine ebenfalls vom Request-Controller verwaltete *Active Request Table* (ART) eingetragen werden. Mit dieser Tabelle lassen sich, allgemein oder für eine bestimmte Objekt-ID, alle z.Zt. in Abarbeitung befindlichen `ServerRequest`-Objekte ermitteln. Diese Informationen sind relevant für den Deaktivierungsprozeß eines Objektes, den Zustandswechsel eines POAs und den POA-Zerstörungsprozeß. Als nächstes wird ein freier Request-Processor vom Pool-Manager angefordert. An dieser Stelle blockiert der Request-Controller-Thread solange, bis ein freier Request-Processor zur Verfügung gestellt werden kann. Nach dem Erhalt des Request-Processors wird dieser mit einer Referenz auf den verantwortlichen Request-Controller, dem ermittelten Servant bzw. dem Servant-Manager initialisiert. Danach wird der Request-Processor-Thread gestartet, um mit der Anfragebearbeitung fortzufahren.

Damit ist des Auslesen der Queue von der eigentlichen Durchführung der Aufrufe auf Servant-Manager und Servant entkoppelt, und der Request-Controller-Thread kann das nächste `ServerRequest`-Objekt aus der Queue holen und einer Bearbeitung zuführen. Dieser Ansatz erlaubt die nebenläufige Ausführung von Objektaufrufen im Rahmen der vom Pool-Manager festgelegten Ressourcengrenzen. Der eben beschriebene Ablauf stellt in zusammenhängender Weise die optimistische Sicht auf den Prozeß der Zuweisung eines Request-Processors dar. Wenn eine der beschriebenen Vorbedingungen nicht erfüllt werden kann, wird die Bearbeitung des `ServerRequest`-Objektes beendet und eine geeignete System-Exception erzeugt, die dem `ServerRequest`-Objekt übergeben wird. Abschließend erfolgt die Rückmeldung des `ServerRequest`-Objektes beim Basic-Adapter.

Aufrufe auf Servant-Manager und Servant

Der startbereite Request-Processor-Thread wird noch innerhalb des Request-Controller-Threads mit den neuen Aufrufparametern initialisiert und zur Bearbeitung eines einzelnen `ServerRequest`-Objektes geweckt. Wenn bei der Initialisierung ein Servant-Manager übergeben wurde, muß als erstes festgestellt werden, ob es sich um eine Instanz eines Servant-Activators oder eines Servant-Locators handelt. Im Falle eines Servant-Activators wird über den Request-Controller die AOM ermittelt. Diese ist unter anderem für die Inkarnation eines Servants mit anschließender Objektaktivierung zuständig. Ein Aufruf von `incarnate()` auf der AOM liefert den aktivierten Servant zurück. Im Unterschied zum Servant-Activator wird bei einem Servant-Locator die Operation `preinvoke` direkt aufgerufen. Auch in diesem Fall wird der für diesen Aufruf gültige Servant zurückgeliefert. In beiden Fällen kann eine `org.omg.PortableServer.ForwardRequest` User-Exception auftreten. Die Bearbeitung der Anfrage ist damit beendet und die Exception wird im `ServerRequest`-Objekt gesetzt.

Wenn ein Servant zur Verfügung steht, der entweder bei der Initialisierung übergeben oder durch den Aufruf auf dem Servant-Manager ermittelt wurde, kann der eigentliche Servant-Aufruf erfolgen. Hierbei wird zwischen stream-basierten und DSI-basierten Servants unterschieden. Stream-basierte Servants implementieren das Interface `org.omg.CORBA.portable.InvokeHandler`. Die Operation `_invoke` verlangt als Parameter die Operationsbezeichnung als String, eine Instanz der Klasse `org.omg.CORBA.portable.InputStream` und eine Implementierung des Interfaces `org.omg.CORBA.portable.ResponseHandler`. Die Operationsbezeichnung und den Input-Stream stellt das `ServerRequest`-Objekt bereit. Da das `ServerRequest`-Objekt das `ResponseHandler`-Interface implementiert, kann es selbst auch als Parameter direkt übergeben werden. DSI-basierte Servants sind Instanzen der Klasse `org.omg.PortableServer.DynamicImplementation`. Die Aufrufoperation `invoke` verlangt als einzigen Parameter ein `ServerRequest`-Objekt. Da die `ServerRequest`-Klasse auch diese Schnittstelle bedient, kann es genauso als Parameter für DSI-basierte Aufrufe verwendet werden.

Nach der Beendigung des Aufrufes muß für den Fall, daß im Vorfeld ein Servant-Locator zum Einsatz kam, die `postinvoke()` Operation auf dem Servant-Locator ausgeführt werden. Da der Aufruf auf dem Servant und der Aufruf auf dem Servant-Locator als eine Einheit zu betrachten sind, und der Servant-Aufruf nur dann als erfolgreich gilt, wenn auch die `postinvoke()` Operation fehlerfrei durchgeführt werden konnte, muß eine hierbei auftretende Exception zur Rücksetzung des Bearbeitungsergebnisses führen. Das Setzen einer System-Exception innerhalb des `ServerRequest`-Objektes zieht genau dieses nach sich. Es besteht aber keine Transaktionssemantik für den Servant-Aufruf. Die aufgetretenen Effekte können nicht rückgängig gemacht werden.

Während der Aufrufe auf Servant-Manager und Servant können verschiedene Fehlersituationen auftreten, die alle aufgefangen werden und nicht zu einem Programmabbruch führen dürfen. Da es sich in beiden Fällen um Code handelt, den der Anwendungsentwickler zur Verfügung stellt und der sich damit außerhalb der Kontrolle der ORB-Entwickler befindet, soll kein fehlerhafter Code einer einzelnen Servant- oder Servant-Manager-Implementierung alle anderen Systemteile in Mitleidenschaft ziehen. In allen Fällen wird eine geeignete System-Exception erzeugt und dem `ServerRequest`-Objekt übergeben. Die Bearbeitung eines einzelnen `ServerRequest`-Objektes ist damit für diesen Request-Processor beendet.

Egal ob die Bearbeitung erfolgreich war oder nicht, nach der Beendigung der Aufrufe wird der Request-Controller über die Fertigstellung der Anfragebearbeitung informiert. Er übernimmt die Benachrichtigung des Basic-Adapters und trägt das `ServerRequest`-Objekt aus seiner ART aus. Transparent für den Request-Controller erfolgt noch die Rückgabe des Request-Processors. Der Request-Processor meldet sich bei seinem Pool-Manager zurück und kann von diesem in den Pool der freien Prozessoren eingeordnet werden. Der Request-Processor-Thread blockiert nun, um auf die Bearbeitung einer neuen Anfrage zu warten.

Interceptors

Im vorherigen Abschnitt blieben, aus Gründen der besseren Übersicht, die Aufrufe auf den beim ORB registrierten *Interceptors* unerwähnt. Diese Aufrufe werden durchgeführt, sofern serverseitige *Interceptors* für den Serverprozeß konfiguriert wurden. *Message-Level-Interceptors* bekommen die Gelegenheit, den Anfragepuffer innerhalb des `ServerRequest`-Objektes zu manipulieren. Ist die Verwendung der *Request-Level-Interceptors* für einen Serverprozeß eingeschaltet, wird dem ORB kurz vor und nach dem eigentlichen Servant-Aufruf das aktuelle `ServerRequest`-Objekt zur Bearbeitung durch die registrierten *Interceptors* übergeben.

Provisorische Auslieferung einer Anfrage an einen Eltern-POA

`ServerRequest`-Objekte können unter Umständen nicht gleich an den zuständigen POA ausgeliefert werden. Dies ist der Fall, wenn sich ein übergeordneter POA, in der Kette ausgehend vom Root-POA, im Zustand **HOLDING** befindet. Diese Situation erfordert die Pufferung der Anfrage, bis der blockierende POA wieder in den aktiven Zustand versetzt wird. Zur Lösung dieses Problems wird das `ServerRequest`-Objekt vorerst dem im Zustand **HOLDING** befindlichen POA übergeben, der es in seiner Queue zwischenspeichert. Bei einer Fortsetzung der Anfragebearbeitung wird dieser Sachverhalt erkannt. Nach dem Auslesen eines `ServerRequest`-Objektes überprüft er, ob es für ihn bestimmt ist. Nicht für ihn bestimmte `ServerRequest`-Objekte enthalten einen "verbleibenden POA-Namen", der die Namenskette bis zum eigentlich zuständigen POA enthält. Die Auslieferung dieser `ServerRequest`-Objekte wird innerhalb des Request-Controller-Threads erneut in Gang gesetzt, wobei aber nur noch die verbleibenden POAs auf dem Weg zum zuständigen POA durchlaufen werden müssen. Dem Basic-Adapter werden dabei das `ServerRequest`-Objekt und die eigene `POA`-Referenz ausgehändigt, von der aus die Auslieferung erneut gestartet werden kann.

Lokale Aufrufe

Die innerhalb eines Serverprozesses von einem POA erzeugten Objektreferenzen verweisen natürlich auf lokale Objekte. Um lokale Aufrufe in JacORB zu optimieren, wird ein Delegate mit der erzeugenden `POA`-Instanz assoziiert. Die generierten Stub-Klassen enthalten Code, der im lokalen Anwendungsfall ausgeführt wird. Jeder Aufruf auf einer lokalen Objektreferenz wird nicht wie sonst üblich über den ORB vermittelt, sondern derartige Stubs ermitteln durch den Aufruf von `servant_preinvoke()` auf ihrem Delegate den zuständigen Servant und führen die Operation auf diesem direkt als normalen Objektaufruf aus. Die im Delegate gespeicherte `POA`-Referenz dient dazu, den aktuell zuständigen Servant herauszufinden. Die Ermittlung des zuständigen Servants wird für jeden Aufruf neu vorgenommen.

Die Umsetzung lokaler Aufrufe weist in JacORB noch einige Schwächen auf. Diese müssen in zukünftigen Versionen beseitigt werden, um der in der Spezifikation geforderten Lokalitätstransparenz zu genügen (siehe Abschnitt 4.8). Durch die direkte Assoziation eines Delegates mit der erzeugenden `POA`-Instanz existieren in JacORB noch Unterschiede in der Semantik des Auslieferungsprozesses lokaler und entfernter Anfragen, bsw. haben POA-Zustände und das eingestellte Threadmodell keinen Einfluß auf die Anfragebearbeitung. Außerdem wird der optimierte Auslieferungsmechanismus momentan nicht von jedem POA unterstützt. Nur entsprechend konfigurierte POAs (**RETAIN** oder **USE_DEFAULT_SERVANT**) können einen Servant für eine lokale Anfrage ermitteln, eine Servant-Inkarnation bei Bedarf ist nicht möglich. Zur Beseitigung dieser Implementierungsschwächen sollten lokale Anfragen nicht an den POA direkt gerichtet werden. Ein einheitlicher Auslieferungsmechanismus über den ORB mit einem *Redesign* der Auslieferungsklassen- bzw. -schnittstellen und einer Anpassung an den lokalen Anwendungsfall ist notwendig.[2] Die `servant_preinvoke()` Operation würde die Auslieferung der Anfrage

[2]Die `ServerRequest`-Klasse ist momentan nur für die Auslieferung entfernter Anfragen ausgelegt. Ein *Redesign* müßte eine einheitliche Schnittstelle für lokale und entfernte Anfragen definieren,

und die Ermittlung des Servants kapseln. Die POA-Implementierung könnte bis zu dem Punkt, an dem der eigentliche Servant-Aufruf stattfindet, jede Anfrage gleich behandeln. Im lokalen Fall würde das Delegate über die Bereitstellung des Servants informiert werden, damit der Stub im Anschluß den Servant-Aufruf direkt durchführen kann.

Synchronisationsmaßnahmen

Der Request-Controller-Thread konkurriert bei der Entnahme von `ServerRequest`-Objekten aus der Request-Queue mit potentiell beliebig vielen Request-Receptor-Threads, die ihrerseits neue Elemente in die Queue einfügen. Es muß sichergestellt werden, daß die Zugriffsoperationen der Request-Queue atomar ausgeführt werden, damit jede Operation auf einer konsistenten Datenstruktur arbeitet. Wenn sich mit einer Einfüge-Operation genau ein Element in der Queue befindet, muß der Request-Controller-Thread geweckt werden. Der Request-Controller wird über dieses Ereignis informiert. Er übernimmt die Synchronisation zum Starten und Anhalten des Request-Controller-Threads.

Während der Arbeit des Request-Controller-Threads müssen ebenfalls Synchronisationsmaßnahmen getroffen werden. Beliebige Threads der Serveranwendung können jederzeit, konkurrierend mit dem Request-Controller-Thread, den POA-Zustand ändern, den POA herunterfahren oder ein Objekt deaktivieren. Vor dem Starten eines Request-Processors muß sichergestellt werden, daß sich der POA im Zustand **ACTIVE** befindet, daß der POA nicht heruntergefahren wird und daß ein Deaktivierungsprozeß des Zielobjektes noch nicht begonnen hat. Im Gegenzug muß natürlich gewährleistet werden, daß der POA erst zerstört werden kann, wenn alle aktiven Requests beendet wurden. Ähnliches trifft auch auf die Deaktivierung eines Objektes zu. Diese soll erst möglich sein, wenn alle Anfragen an das zu deaktivierende Objekt beendet wurden.

Die Einhaltung dieser Bedingungen liegt in der Verantwortung des Request-Controllers. Er verwaltet die *Active Request Table* (ART) für die in Bearbeitung befindlichen `ServerRequest`-Objekte. Außerdem führt er eine Deaktivierungsliste für Objekt-IDs, deren Objektdeaktivierung gerade läuft, und er stellt zur Synchronisation einen Monitor zur Verfügung. Kritischer Code wird in diesem Monitor zusammengefaßt und läuft serialisiert ab. Dazu gehört die Feststellung des POA-Zustandes, die Tests, ob der POA gerade zerstört wird und ob sich das Zielobjekt im Deaktivierungsprozeß befindet, sowie die Ermittlung von Servant und Servant-Manager, die Eintragung des `ServerRequest`-Objektes in die ART und dessen Austragung. Der POA muß für die Durchführung eines Zustandswechsels, für seine Zerstörung und eine Objektdeaktivierung ebenfalls Zutritt zu diesem Monitor erlangen. Auf diese Prozesse wird im folgenden noch detailliert eingegangen.

6.2.6. Aktivierung und Deaktivierung von Objekten

Die Objektaktivierung wird durch eine `add()` Operation auf der AOM vorgenommen. Die Objektdeaktivierung erzeugt einen extra Thread für den Deaktivierungsprozeß eines Objektes. Die `deactivate_object()` Operation kehrt nach dem Start des Deaktivierungsthreads sofort wieder zurück. Innerhalb des Threads wird zuerst

die von spezialisierten Klassen implementiert werden.

der Request-Controller über die vorzunehmende Objektdeaktivierung informiert. Damit trägt er die Objekt-ID in die Deaktivierungsliste ein. Erst mit dem Eintrag in diese Liste werden keine weiteren Anfragen an dieses Objekt gestartet. Nach der Rückkehr der Operation wird die `remove()` Operation der AOM aufgerufen. Sollte ein POA mit einem Servant-Activator konfiguriert worden sein, wird innerhalb dieser Operation automatisch eine Servant-Etherealization durchgeführt. Zum Abschluß wird der Request-Controller über die Fertigstellung des Deaktivierungsprozesses informiert. Dieser trägt die Objekt-ID aus der Deaktivierungsliste aus. Eine Servant-Inkarnation wird ebenfalls über die AOM durchgeführt. Die AOM stellt dafür eine `incarnate()` Operation bereit, die den Aufruf auf dem Servant-Activator durchführt und eine Objektaktivierung zur Folge hat.

Synchronisationsmaßnahmen

Objektaktivierungen bzw. -deaktivierungen können nebenläufig, innerhalb mehrerer Anwendungsthreads auftreten. Der Request-Controller-Thread konkurriert durch seine fortlaufenden Ermittlung eines aktuellen Servants für die Initialisierung der von ihm gestarteten Prozessoren mit dem Deaktivierungsthread beim Zugriff auf die AOM. Vor der Austragung einer Objekt-ID aus der AOM müssen alle aktiven Requests für dieses Objekt beendet sein, und es dürfen bis zum Zeitpunkt der endgültigen Austragung keine weiteren Anfragen an dieses Objekt gestartet werden. Wenn der Request-Controller-Thread einen Servant ermitteln konnte, gilt dieses Objekt für die Dauer der Anfragebearbeitung als aktiviert. Mehrere Request-Processor-Threads können, untereinander und mit manuellen Objektaktivierungen konkurrierend, Servant-Inkarnationen vornehmen. Hinzu kommen Objektdeaktivierungsthreads, in denen eventuell eine Servant-Etherealization auftritt. In der Spezifikation ist vorgeschrieben, daß alle Aufrufe von `incarnate()` auf einem Servant-Activator serialisiert durchzuführen sind, und Aufrufe von `incarnate()` und `etherealize()` müssen sich gegenseitig ausschließen. All diese Aktivitäten können potentiell nebenläufig auftreten, es sind demzufolge geeignete Synchronisationsmaßnahmen zu treffen, die das spezifizierte Verhalten sicherstellen.

Von einem `AOM`-Objekt wird ein Monitor zur Synchronisation bereitgestellt. Alle Einfüge- und Entferne-Operationen, sowie die `incarnate()` Operation der AOM werden innerhalb des Monitors ausgeführt. Wenn eine Operation Code ausführt, der vom Anwender bereitgestellt wird, wie es bsw. bei der `incarnate()` Operation der Fall ist, reicht die einfache Verwendung des Java-Sprachausdrucks `synchronized` für die Sicherstellung des serialisierten Ablaufs nicht aus. Es kann nicht verhindert werden, daß der Monitor im Servant-Activator-Code verlassen wird. Als zusätzliche Maßnahmen werden innerhalb der AOM zwei Listen verwaltet, eine, in die Objekt-IDs eingetragen werden, die Ziel einer Servant-Inkarnation sind, und eine weitere Liste von Objekt-IDs, die der Protokollierung der aktiven Etherealization-Aufrufe dient. Eine Servant-Inkarnation blockiert nun solange, bis sich in beiden Listen keine Einträge mehr befinden, und eine Servant-Etherealization wird erst ausgeführt, wenn die Inkarnationsliste leer ist. Jedem Entfernen eines Listenelements folgt die Benachrichtigung aller blockierten Threads.

An dieser Stelle möchte ich noch die besondere Anforderung an einen *single-threaded* POA erwähnen. Alle Aufrufe auf einem Servant-Manager sollen serialisiert werden. Die Aufrufe von `incarnate()` und die auf einem Servant-Locator laufen schon aus dem Grund serialisiert ab, da nur ein einzelner Request-Processor zur Verfügung steht. Einzig die Aufrufe von `etherealize()` bedürfen einer besonderen Betrachtung. Für diesen speziellen Fall wird vor einem solchen Aufruf auch getestet, ob die Etherealizationliste leer ist.

Eine Servant-Inkarnation gilt erst dann als erfolgreich, wenn auch die Objektaktivierung durchgeführt werden kann. Aus diesem Grund wird die `AOM`-Operation `incarnate` komplett im Monitor ausgeführt. Erst wenn diese erfolgreich abgeschlossen wurde, inklusive der Objektaktivierung, kann eine nebenläufige Aktivierungsoperation Zugang zum Monitor erhalten. Dies ist auch gewährleistet, wenn bei der Servant-Inkarnation innerhalb des Servant-Activator-Codes der Monitor freigegeben wird, denn eine Aktivierungsoperation blockiert solange, bis für diese Objekt-ID kein Eintrag mehr in der Inkarnationsliste verzeichnet ist.

Ein weiteres Feld betrifft die Synchronisation der Objekt(de)aktivierungsprozesse mit der Anfragebearbeitung. Die Ermittlung des Servants und die Eintragung des aktuellen `ServerRequest`-Objektes in die *Active Request Table* (ART) führt der Request-Controller-Thread innerhalb seines Monitors aus. Der Deaktivierungsthread muß bei der Benachrichtigung des Request-Controllers ebenfalls Zugang zu diesem Monitor erhalten. Diese Operation blockiert nun solange (und gibt damit den Monitor wieder frei), bis für die zu deaktivierende Objekt-ID kein Eintrag mehr in der ART verzeichnet ist. In der Zwischenzeit kann der Request-Controller aber weitere Anfragen an dieses Objekt starten. Erst wenn die Objekt-ID nicht mehr verzeichnet und der Deaktivierungsthread wieder im Besitz des Monitors ist, wird die Objekt-ID in die Deaktivierungsliste des Request-Controllers eingetragen. Dies führt zu dem im Abschitt 4.10 beschriebenen Effekt der "Deaktivierungsblockade bei fortlaufenden Aufrufen auf einem Objekt".

In der Zeit, in der der Deaktivierungsthread im Besitz des Monitors ist, kann der Request-Controller-Thread keinen Servant ermitteln, und für die Zeit danach führt der Eintrag in der Deaktivierungsliste zur Abweisung aller Anfragen an dieses Objekt. Eine Ausnahme bildet ein POA, der über einen Default-Servant oder einen Servant-Manager verfügt. Dieser hat eine weitere Möglichkeit, einen Servant für die Anfrage zu ermitteln. Die Austragung der Objekt-ID aus der AOM kann nun ohne die Synchronisation mit der Anfragebearbeitung erfolgen. Der Eintrag in der Deaktivierungsliste des Request-Controllers wird erst nach der Austragung der Objekt-ID aus der AOM und einer eventuellen Servant-Etherealization entfernt. Auch für diese Operation muß der Deaktivierungsthread Zugang zum Monitor des Request-Controllers erhalten. Eine Reaktivierung des Objektes ist also erst mit dieser Austragung aus der Deaktivierungsliste wirksam.

Ich möchte noch auf einen weiteren, subtilen Sachverhalt hinweisen. Sollte es innerhalb des Request-Controller-Threads nicht gelungen sein, einen Servant für eine Objekt-ID zu ermitteln, der POA aber mit einem Servant-Activator konfiguriert worden sein, dann wäre es theoretisch möglich, daß bis zum Aufruf der Operation `incarnate` auf der AOM innerhalb des Request-Processors ein nebenläufiger Anwendungsthread die Aktivierung dieses Objektes vornimmt. Dieser Umstand ist der fehlenden Synchronisation mit dem Request-Controller-Thread und dessen Entkopplung vom Request-Processor-Thread geschuldet. Eine einfache Lösung des Problems besteht darin, daß die `incarnate()` Operation der AOM einen zwischenzeitlich aktivierten Servant ermittelt und zurückliefert. Die Servant-Inkarnation wird dann fallengelassen.

6.2.7. POA-Zustandswechsel

Die Benachrichtigung der bei einem POA-Manager registrierten POAs wird bei allen Zustandswechsel-Operationen in einem extra Thread durchgeführt, damit diese sofort zurückkehren können. Diese generelle explizite Benachrichtigung der POAs wäre nicht in jedem Fall notwendig, da ein POA, der sich nicht im Zustand **HOLDING** befindet, bei eintreffenden `ServerRequest`-Objekten im Zyklus des Request-Controller-Threads den aktuellen Zustand erfragt. Da aber die Ausgangssituation bei der Implementierung keine Rolle spielen soll und alle Zustandswechsel-Operationen so aufgerufen werden können, daß sie sofort nach der Aktualisierung des Zustandes im POA-Manager zurückkehren, wird dieser Mechanismus einheitlich für alle Operationen gewählt. So ist die Benachrichtigung der POAs als eine Art "Weckdienst" zu verstehen, der "nicht schadet, wenn man von allein aufgewacht ist". Der POA stellt für jede der vier Zustandswechsel-Operationen über seine paket-interne Schnittstelle eine Operation bereit, die der POA-Manager zur Benachrichtigung aufrufen kann und die die vorgeschriebenen Maßnahmen für einen einzelnen POA vornimmt.

Der POA kann seinerseits Operationen des Request-Controllers aufrufen, um Einfluß auf die Anfragebearbeitung zu nehmen, bzw. um die Beendigung aller aktiven Anfragen abzuwarten. Im einzelnen handelt es sich um die Operation `continueToWork`, die den Request-Controller zwingt, den Request-Controller-Thread aufzuwecken, falls er sich aus Mangel an Anfragen in der Queue oder einem vormals gelesenen Zustand **HOLDING** schlafen gelegt hat. Damit ist immer das Neulesen des aktualisierten POA-Zustandes verbunden. Die Operation `waitForCompletion` läßt den aufrufenden Thread solange blockieren, bis alle aktiven Requests zum Ende gekommen sind. Der Aufruf der Operation `resetPreviousCompletionCall` macht einen vorher getätigten Aufruf von `waitForCompletion()` unwirksam und beendet das Warten aller vorangegangenen, aber noch nicht abgeschlossenen Zustandswechsel-Operationen. Mit Hilfe dieser drei Request-Controller-Operationen wird das spezifizierte Verhalten der Zustandswechsel-Operationen implementiert.

Synchronisationsmaßnahmen

Der Request-Controller-Thread bleibt nur dann aktiv, wenn die Request-Queue mindestens ein `ServerRequest`-Objekt enthält, der POA sich nicht im Zustand **HOLDING** befindet und nicht heruntergefahren wird. Ist eine dieser Bedingungen nicht gegeben, legt sich der Request-Controller-Thread schlafen. Der Test wird vom Request-Controller-Thread innerhalb eines speziellen Monitors durchgeführt. Die Operation `continueToWork` muß ebenfalls Zutritt zu diesem Monitor erlangen, um einen eventuell blockierten Request-Controller-Thread zu wecken. Diese Operation wird ebenfalls von der Request-Queue verwendet, um das Eintreffen eines neuen `ServerRequest`-Objektes anzuzeigen. Der Request-Controller-Thread kann nun den aktualisierten POA-Zustand erfragen und das Auslesen und die Bearbeitung der `ServerRequest`-Objekte fortführen.

Die Operation `waitForCompletion` synchronisiert externe Threads, die das Ende aller aktiven Requests abwarten wollen. Dabei ist wichtig, daß diesem Aufruf eine Aktion vorausging, die das Starten neuer Prozessoren unterbindet. Derartige Aktionen können sein: der Wechsel in die POA-Zustände **HOLDING**, **DISCARDING** oder **INACTIVE** oder das Herunterfahren des POAs. Die Operation `waitForCompletion` blockiert solange, bis in der *Active Request Table* keine Einträge mehr enthalten sind oder ein Aufruf von `resetPreviousCompletionCall()` erfolgt. Dazu müssen beide Operationen Zutritt zu dem Monitor erhalten, in dem die Einträge in die ART vorgenommen werden (der Monitor, über den die Anfragebearbeitung des Request-Controllers synchronisiert wird). Außerdem muß ein Flag innerhalb des Request-Controllers gesetzt werden, das den Aufruf von `waitForCompletion()` anzeigt. Die Operation `resetPreviousCompletionCall` setzt dieses Flag zurück und benachrichtigt alle wartenden externen Threads.

6.2.8. POA-Zerstörung

Die Zerstörung eines POAs wird über die `POA`-Operation `destroy` durchgeführt, der zwei Parameter übergeben werden, die bestimmen, ob eine Servant-Etherealization durchzuführen ist und ob die Beendigung des Zerstörungsprozesses abgewartet werden soll. Während des Herunterfahrens einer `POA`-Instanz gibt es drei Zustände, die bis zur vollständigen Zerstörung durchlaufen werden. Der aktuelle Zustand wird in einem privaten `POA`-Attribut festgehalten. Der Zerstörungsprozeß läßt sich in drei Phasen aufteilen. Die Konsequenzen jeder einzelnen Phase werden in der folgenden Beschreibung deutlich. Für Phase 2 und Phase 3 wird ein extra Thread gestartet, so daß die Aufrufe, die nicht auf die Beendigung der Aktion warten sollen, nach dem Starten des Zerstörungsthreads zurückkehren können.

Phase 1 - Zerstörung aller Kind-POA

Als erstes wird das Herunterfahren des POAs durch das Setzen des Zustandsattributes angezeigt. Der POA befindet sich nun im Zustand **SHUTDOWN_IN_PROGRESS**. Dies hat zur Konsequenz, daß eine Erzeugung weiterer Kind-POAs für diesen POA nicht mehr möglich ist. Als nächstes wird die Liste der registrierten Kind-POAs durchlaufen und auf diesen ebenfalls die `destroy()` Operation aufgerufen. Mit der eigenen Zerstörung wird erst dann fortgefahren, wenn alle Kind-POAs den hier beschriebenen Zerstörungsprozess vollständig durchlaufen haben. Der eigene Zerstörungsprozess wird in einem extra Thread ausgeführt. Sollte der Parameter `wait_for_completion` nicht gesetzt sein, kehrt die Operation nach dem Starten des Threads zurück.

Phase 2 - Zerstörung wird "sichtbar" gemacht

Der Request-Controller wird darüber informiert, daß die Anfragebearbeitung infolge des Herunterfahrens des POAs einzustellen ist. Dazu wird auf dem Request-Controller die Operation `waitForShutdown` aufgerufen. Diese beendet die Anfragebearbeitung vollständig und dauerhaft und blockiert solange, bis alle aktiven Requests zum Ende gekommen sind. Der Request-Controller-Thread wird blockiert. Der POA verhält sich nun wie im Zustand **HOLDING**, alle zukünftig eintreffenden Anfragen werden gepuffert. Es gibt nun auch keine Möglichkeit mehr, den Request-Controller-Thread zu reaktivieren. Nachdem die Anfragebearbeitung ruht, meldet sich der POA bei seinem POA-Manager ab und setzt zum Abschluß dieser Phase das Zustandsattribut. Der POA befindet sich nun im Zustand **DESTRUCTION_APPARENT**. Damit ändern alle anderen Operationen der POA-IDL-Schnittstelle ihr Verhalten. Sollte eine Operation aufgerufen werden, nachdem die POA-

Zerstörung "sichtbar" ist, wird eine **OBJECT_NOT_EXIST** System-Exception geworfen. Eine Ausnahme bildet nur die `destroy()` Operation selbst. Ein weiterer Effekt ist der, daß ein solcher POA vom ORB nicht mehr erreicht werden kann. Die `_getChildPOA()` Operation, die vom Basic-Adapter aufgerufen wird, erkennt bei der Ermittlung eines Kind-POAs diesen Zustand. Damit ist er für den Eltern-POA nicht mehr existent.

Phase 3 - Aufräumungsarbeiten und Freigabe der benutzen Ressourcen

In der letzten Phase folgen noch einige Aufräumungsarbeiten. Als erstes wird die Request-Queue geleert. Alle noch darin enthaltenen `ServerRequest`-Objekte werden nach dem Setzen einer **OBJECT_NOT_EXIST** System-Exception dem Basic-Adapter zurückgegeben. Dann folgt die Servant-Etherealization. Sollte der übergebene `etherealize` Parameter gesetzt und ein Servant-Activator beim POA registriert sein, wird auf der AOM die `removeAll()` Operation aufgerufen, die für jede enthaltene Objekt-ID die `etherealize()` Operation auf dem Servant-Activator durchführt. Damit sind alle Verpflichtungen erfüllt, die der POA gegenüber dem ORB besitzt. Die restlichen Aktionen dienen nur noch der Freigabe der benutzen Systemressourcen. Die eigentliche Freigabe wird natürlich dem *Java Garbage Collector* überlassen, es müssen nur die Voraussetzungen für dessen Arbeit geschaffen werden. Als erstes folgt die Beendigung aller aktiven Threads. Dazu wird dem Pool-Manager die Aufforderung geschickt, alle Request-Processor-Threads terminieren zu lassen. Ebenso wird mit dem Request-Controller-Thread verfahren. Alle weiteren Objekte, die vom POA verwendet wurden, werden nur von diesem referenziert, so daß eine Abmeldung des POAs beim Eltern-POA ausreicht, um die nun nicht mehr benötigten Objekte vom *Garbage Collector* abräumen zu lassen. Diese Abmeldung signalisiert dem Eltern-POA gleichzeitig die Fertigstellung der Zerstörung seines Kind-POAs. Er hat nun in Zukunft die Möglichkeit, einen neuen POA unter diesem Namen zu erzeugen.

Synchronisationsmaßnahmen

Ein mehrmaliges Aufrufen der `destroy()` Operation auf derselben `POA`-Instanz ist möglich. Der `etherealize` Parameter, der im ersten Aufruf übergeben wurde, soll auch für alle weiteren Aufrufe gelten. Demzufolge wird der Wert im POA abgespeichert und für alle weiteren Aufrufe genutzt. Der `wait_for_completion` Parameter wird im Gegensatz dazu für jeden Aufruf individuell gehandhabt. Auch später ausgeführte `destroy()` Aufrufe blockieren bis die Zerstörung des POAs beendet wurde. Weitere Synchronisationsmaßnamen beziehen sich vor allem auf den Schutz konkurrierender POA-Erzeugungsprozesse, die schon im Abschitt 6.2.3 beschrieben wurden.

6.3. Einführung von POA-Ereignissen

Da man davon ausgehen kann, daß die Prozesse, in denen die POA-Implementierung involviert ist, auch an anderer Stelle der ORB-Implementierung von Interesse sind, habe ich eine Reihe von POA-Ereignissen definiert, die über festgelegte Schnittstellen an interessierte Komponenten propagiert werden. Es ist leicht vorstellbar, wie spätere Erweiterungen der POA-Implementierung auf solchen Ereignisse aufbauen können, um Funktionlität hinzuzufügen. Nur als ein Beispiel sei erwähnt, daß der POA maßgeblich am Lebenszyklus eines Objektes beteiligt ist und die damit verbundenen Aktionen kapselt. Der POA könnte bsw. andere Komponenten von der Erzeugung eines Objektes informieren, die ihrerseits diese Information in darauf aufbauenden Aktionen verwerten.

Bei der Realisierung der POA-Ereignisse habe ich mich für die Implementierung eines *Event-Listener Patterns* als eine Variante des *Observer Patterns* entschieden [6]. Ein Event-Mechanismus stellt für einen Java-Programmierer ein gewohntes Programmiermodell dar, da er an vielen Stellen angewendet wird. Ein solcher Entwurf ermöglicht die Assoziation von Komponenten ohne die Implementierungsklassen so eng aneinander zu koppeln, daß ihre Wiederverwendbarkeit eingeschränkt ist. Die Klassen können zusammenarbeiten aber auch unabhängig voneinander eingesetzt werden. Die Anzahl der abhängigen Objekte bleibt variabel. Alles, was ein Event-Erzeuger weiß, ist, daß er eine Liste von Event-Listenern besitzen kann, die die festgelegte Listener-Schnittstelle implementieren. Die Kopplung zwischen Event-Erzeuger und Event-Listener ist somit auf ein Minimum reduziert. Für die POA-Implementierung bedeutet dies, daß sie die potentiellen Interessenten an POA-Ereignissen nicht kennen muß.

Eine ereigniserzeugende POA-Komponente benachrichtigt seine Listener wenn ein Ereignis auftritt. Um die notwendigen Interaktionen zwischen den Objekten einzuschränken, wurden, in Abwandlung zum *Observer Pattern,* eine Reihe von Ereignissen willkürlich definiert, und die relevanten Parameter werden mit den Benachrichtigungsoperationen übergeben (Push-Modell). Durch die explizite Festlegung der interessierenden Ereignisse wird zwar die Flexibilität für potentielle Listener eingeschränkt, die Effizienz der Aktualisierungen wird aber verbessert.

An dieser Stelle sei schon vorweggenommen, daß die wichtigste Motivation für die Einführung von POA-Ereignissen in der Bereitstellung einer Benutzeroberfläche lag, die die Aktivitäten des POAs visualisiert. Demzufolge wurden nicht nur Ereignisse allgemeiner Art, die sich aus dem CORBA-Objektmodell ableiten lassen, in die Auswahl aufgenommen, sondern auch implementierungsspezifische Ereignisse, die sich aus der konkreten POA-Funktionalität ergeben.

Damit interessierte Listener nicht alle möglichen POA-Ereignisse verarbeiten müssen, wurde eine Aufteilung der Benachrichtigungsoperationen auf mehrere Listener-Interfaces vorgenommen. Die Gruppierung orientiert sich dabei an den ereigniserzeugenden Komponenten der POA-Implementierung. Es werden vier verschiedene Interfaces angeboten. Eine Implementierung dieser Interfaces kann sich als Listener bei der jeweiligen Komponente registrieren. Die Abbildung 6.4 stellt die ereignisbezogenen Klassen und Schnittstellen und ihre Beziehungen zueinander dar und erweitert damit das Klassendiagramm der POA-Implementierung.

jacorb.poa.EventListener Es handelt sich um ein leeres Interface, von dem alle anderen Interfaces erben.

jacorb.poa.POAListener Dieses Interface definiert 3 Operationen, die über grundlegende POA-Ereignisse informieren. Dies betrifft die Erzeugung eines neuen Kind-POAs, die Änderung des POA-Zustandes und die Erzeugung von Objektreferenzen.

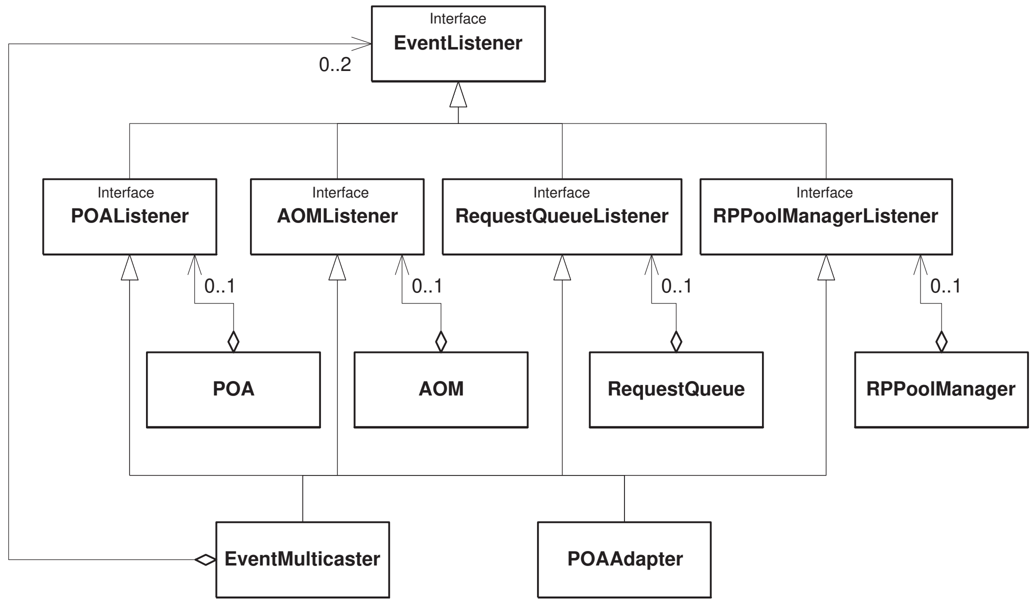


Abbildung 6.4.: Klassendiagramm der ereignisbezogenen Klassen und Schnittstellen

jacorb.poa.AOMListener Dieses Interface definiert 4 Operationen, die über AOM-Ereignisse informieren. Es wird die Aktivierung eines Objektes, seine Deaktivierung, die Servant-Inkarnation und eine Servant-Etherealization propagiert.

jacorb.poa.RequestQueueListener Dieses Interface definiert 2 Operation, die über den Zustand der Request-Queue informieren. Sie werden aufgerufen, wenn ein `ServerRequest`-Objekt der Queue hinzugefügt bzw. aus ihr entfernt wird.

jacorb.poa.RPPoolManagerListener Dieses Interface definiert 2 Operation, die das Thread-Pool-Management betreffen. Wenn der Pool-Manager einen Request-Processor herausgibt, bzw. wenn dieser wieder zurück in den Pool gelangt, benachricht er einen Listener.

jacorb.poa.POAAdapter Diese Klasse zu erweitern, stellt eine bequeme Möglichkeit dar, sich für alle POA-Ereignisse registrieren zu lassen, wobei man aber selbst nur an einer Auswahl interessiert ist, die nicht unbedingt der vorgegebenen Unterteilung entspricht. Die Implementierungen der Operationen sind leer. Eine spezialisierte Klasse überschreibt die sie interessierenden Benachrichtigungsoperationen.

jacorb.poa.EventMulticaster Bei dieser Klasse handelt es sich um eine sehr effiziente Implementierung eines auf POA-Ereignisse spezialisierten *Event-Multicasters.* Er bietet für jeden spezialisierten Event-Listener zwei Klassenmethoden zum An- und Abmelden von Listeners an, die in den jeweiligen Registrierungsoperationen der Event-Erzeuger verwendet werden. Außerdem implementiert er alle Listener-Interfaces, um für die Weiterleitung jeder Benachrichtigungsoperation zu sorgen.

Um die benachrichtigungswilligen Komponenten nicht mit der Verwaltung der Listener zu belasten, und um die zur Aktualisierung der Listener benötigten Aktionen zu minimieren, wurde die `EventMulticaster`-Klasse bereitgestellt, die die An-, Abmelde- und Benachrichtigungsfunktionalität kapselt. Instanzen eines Event-Multicasters können anstelle eines speziellen Event-Listeners referenziert werden. Sie realisieren eine Kette von Event-Listenern, wobei die Instanzen des Event-Multicasters als Verbindungstücke zwischen den Listenern fungieren und für die Propagierung des Ereignisses sorgen.

Da die einzelnen POA-Komponenten der ORB-Implementierung unbekannt sind, und der ORB im allg. nur eine POA-Referenz in den Händen hält, stellt die JacORB-interne `POA`-Schnittstelle zwei Operationen zum An- und Abmelden beliebiger Event-Listener bereit. Die `POA`-Implementierung sorgt dafür, daß der übergebene Listener bei den richtigen Teilkomponenten an- bzw. abgemeldet wird.

Der Implementierung aller Event-Erzeuger müssen ebenfalls die Operationen zum An- und Abmelden der Listener hinzugefügt werden. Einem Event-Erzeuger ist maximal ein Event-Listener bekannt. Wenn mehrere Listener bei einem Erzeuger registriert werden, referenziert dieser eine `EventMulticaster`-Instanz. An den Stellen der POA-Implementierung, an denen ein Ereignis auftritt, wird, falls ein Event-Listener assoziiert ist, der Aufruf der entsprechenden Benachrichtigungsoperation ausgeführt. Es sei noch erwähnt, daß die Benachrichtigung der Listener synchron erfolgt, d.h. alle Listener werden nacheinander aufgerufen, und der Erzeuger fährt erst fort, wenn alle Listener das Ereignis konsumiert haben. Durch diese Vorgehensweise werden die Listener mit der POA-Implementierung synchronisiert, und es steht ihnen frei, ihre Aktionen in einem extra Thread durchzuführen.

Die Auswahl der POA-Ereignisse und die Festlegung der Ereignisparameter sind erst einmal fest vorgegeben, spätere Veränderungen lassen sich aber relativ einfach integrieren. Bei der Auswahl standen natürlich die für die Benutzeroberfläche benötigten Aktualisierungen im Vordergrund. Mit Blick auf die allgemeinen POA-Ereignisse, wie POA-Erzeugung und POA-Zustandsänderungen hat sich in der jüngsten Entwicklung von JacORB schon ein Anwendungsbeispiel ergeben.

Der ORB nutzt diese Ereignisse zur Versorgung des ImRs mit Informationen zu gestarteten Serverprozessen. Er registriert sich als POA-Listener beim Root-POA und wird u.a. über die Erzeugung weiterer Kind-POAs informiert. Der ORB hat nun die Möglichkeit, sich auch bei diesen als Listener anzumelden, um Kenntnis von allen POA-Instanzen zu erhalten. Nach dem Starten eines Serverprozesses durch das ImR (bzw. den SSD) bekommt das ImR vom ORB die Rückmeldung über einen hochgefahrenen POA, der nun bereit ist, Anfragen entgegenzunehmen.

6.4. Benutzeroberfläche

Während der Entwicklung am POA entstand der Gedanke, eine graphische Oberfläche zu schaffen, die den Ressourcenverbrauch und die Aktivitäten während der Anfragebearbeitung visualisiert. Die Oberfläche sollte im wesentlichen ein *Monitoring* des aktuellen POA-Zustandes erlauben, ohne eine Interaktion des Anwenders mit dem Serverprozeß zuzulassen. Der für die Benutzeroberfläche gefundene Name "POA-Monitor" unterstreicht diesen Ansatz. Später wurde dieses Konzept aber doch erweitert, da die Möglichkeiten, auf den Serverprozeß Einfluß zu nehmen, zu interessant erschienen, um sie zu negieren. Der Entwickler bzw. Administrator einer Serveranwendung hat nun die Möglichkeit, von Hand einzelne POAs herunterzufahren, ihre Zustände zu ändern, Objekte zu deaktivieren und noch nicht bearbeitete Anfragen aus der Request-Queue zu entfernen.

Jeder einzelne POA besitzt einen separaten POA-Monitor. In der Abbildung 6.5 wird ein Screenshot gezeigt, der den Monitor eines arbeitenden Root-POAs abbildet. Neben der Anzeige der unveränderlichen Werte, mit denen ein POA parametrisiert wurde, wie der Name und die geltenden Policy-Werte, werden eine Reihe weiterer Zustandsinformationen angezeigt und fortlaufend aktualisiert. Dazu gehören der aktuelle POA-Zustand, die Anzahl der aktivierten Objekte, der Zustand der Request-Queue und Informationen zum Thread-Pool-Management.

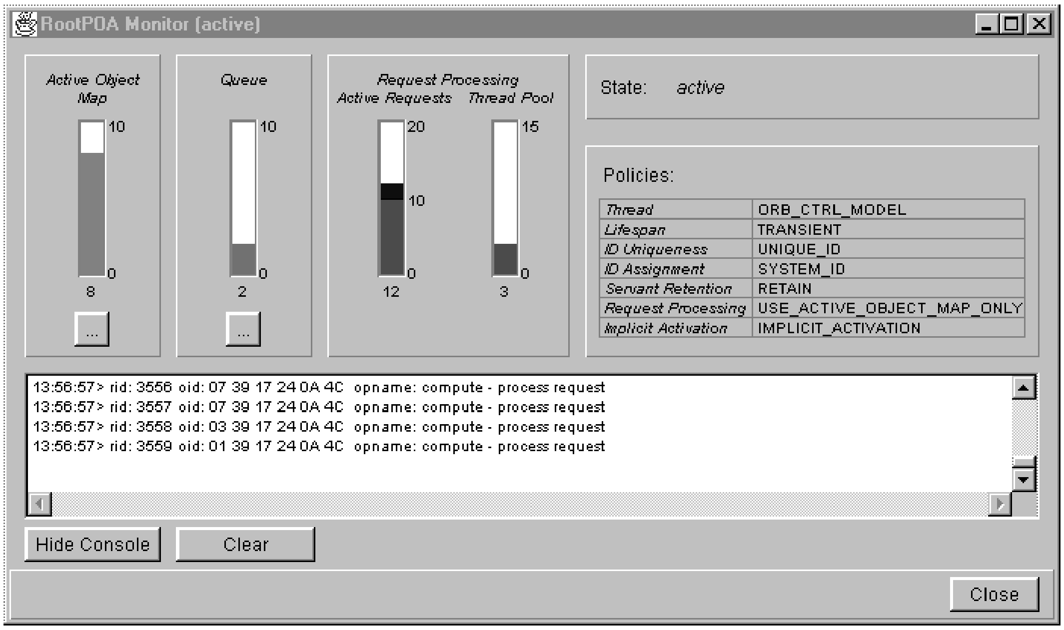


Abbildung 6.5.: Screenshot des POA-Monitors

Anhand der rechten zwei Balken können die momentan verbrauchten Thread-Ressourcen abgelesen werden. Der zweite Balken von rechts stellt die derzeit aktiven Anfragen dar. Überschreitet dieser Wert die konfigurierte Mindestanzahl bereitgestellter Threads, wird dieser Sachverhalt farblich hervorgehoben. Der Entwickler kann daraufhin Rückschlüsse auf die an seine Serveranwendung gestellten Anforderungen ziehen und bei dauerhafter Überschreitung die untere Grenze für die Anzahl vorgestarteter Threads anheben, also den Serverprozeß entsprechend konfigurieren. Entsprechend des im POA implementierten Thread-Pool-Managements zeigt der rechte Balken den momentanen Inhalt des Thread-Pools. Die Differenz zwischen der Obergrenze und der Anzahl der im Thread-Pool bereitstehenden Threads entspricht der Anzahl der ausgegebenen, aktiven Threads. Sicherlich könnte man diese Daten auch über eine Log-Schnittstelle in Erfahrung bringen, die Visualisierung erleichtert aber das Verständnis der POA-Funktionalität. Somit ist die Möglichkeit des Monitorings nicht nur für den erfahrenen Entwickler zum Test seiner Anwendung gedacht, sondern auch für die Lehre bietet sich die Visualisierung der POA-Prozesse an.

Über den Button unterhalb des ersten Balkens, der den Inhalt der AOM darstellt, hat der Anwender die Möglichkeit, einen Snapshot der AOM in einem separaten Fenster zu erhalten. Dieses erlaubt ihm einen Blick auf den aktuellen Zustand der AOM. Alle aktiven Objekte sind mit ihrer Objekt-ID und der Angabe der Implementierungsklasse aufgelistet. Innerhalb dieses Fensters kann auch ein einzelnes Objekt deaktiviert werden. Eine ähnliche Funktionalität wird für den aktuellen Inhalt der Request-Queue geboten. Der Anwender kann einzelne `ServerRequest`-Objekte aus der Queue entfernen. Der Client erhält eine **TRANSIENT** System-Exception und kann seine Anfrage wiederholen.

Einzelne POA-Monitore können geschlossen werden, ohne daß der POA davon betroffen ist. Um sie wieder hervorzuholen, benötigt man eine weitere graphische Komponente, den "POA-Manager-Monitor". Über diesen lassen sich auch die POA-Zustände der mit dem POA-Manager assoziierten POAs ändern. Die Abbildung 6.6 zeigt einen Screenshot des POA-Manager-Monitors. Die registrierten POAs werden in der rechten Listbox, untereinander mit ihrem vollständigen Namen angezeigt. Ein Popup-Menü gestattet den Zugriff auf die POA-bezogenen Operationen, wie das Herunterfahren eines POAs und das Öffnen bzw. Schließen der POA-Monitore. Erst wenn der POA-Manager-Monitor geschlossen wird, sind auch die geschlossenen POA-Monitore nicht mehr erreichbar.

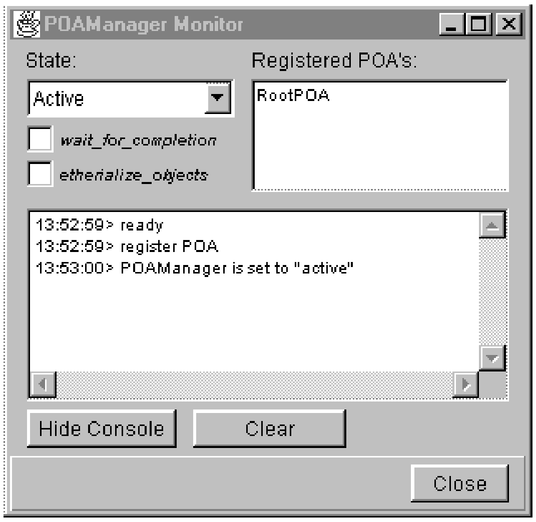


Abbildung 6.6.: Screenshot des POA-Mangager-Monitors

Beide gezeigten Oberflächen besitzen eine Konsole, in der Log-Ausgaben angezeigt werden. Bei geöffnetem Monitor werden diese Meldungen, die normalerweise auf die Standardausgabe gelangen, auf die Konsole des Monitors umgelenkt.

Eine detaillierte Beschreibung der Implementierung würde den Rahmen der Arbeit sprengen. Deshalb möchte ich an dieser Stelle nur die wichtigsten Fakten aufzählen. Die Klassen, die Oberflächenfunktionalität bereitstellen, befinden sich getrennt von den sonstigen Klassen der POA-Implementierung in Unterpaketen von `jacorb.poa.gui`. Für die Oberflächenprogrammierung wurde ausschließlich auf das *Abstract Window Toolkit* (AWT) zurückgegriffen, damit die Monitoring-Funktionalität auch in Java-Umgebungen der Version 1.1 zur Verfügung steht, ohne daß zusätzliche Graphik-Bibliotheken benötigt werden.

Für die Aktualisierung der Monitore werden die im vorherigen Abschnitt beschriebenen POA-Ereignisse genutzt. Die Kopplung zwischen den POA-Implementierungsklassen und den Darstellungsklassen ist somit auf ein Minimum reduziert. Zwischen den POA-Komponenten und den Darstellungsklassen vermittelt ein *Monitor-Controller* (*Model-View-Controller Design*). Ein *Controller* implementiert die erwähnten Listener-Interfaces. Er wird bei den ereigniserzeugenden POA-Komponenten registriert und über jede Zustandsänderung informiert. Diese Informationen leitet er mit oberflächenspezifischen Aufrufen an den *View* weiter. Durch die zugelassenen Interaktionsmöglichkeiten ist eine vollständige Entkopplung zwischen *Model* und *Controller* nicht möglich, der *Controller* muß die POA-Komponenten explizit kennen, um deren Operationen aufrufen zu können. Auf der anderen Seite haben zumindest POA und POA-Manager Kenntnis von ihrem Monitor-Controller, da sie die Erzeugung und Initialisierung des Controllers vornehmen. Insgesamt beschränken sich die Abhängigkeiten zwischen den POA-Komponenten und den Monitor-Controllern auf wenige Zeilen im Programmcode, der Hauptanteil der Kommunikation wird über die POA-Ereignisse abgewickelt.

Die Verwendung des Monitoring kann für einen Serverprozeß über die Java-Property `jacorb.poa.monitoring` konfiguriert werden. Auf der Kommandozeile reicht das Setzen der Property in der Form "-D`monitoring=on`". Damit wird für jede `POA`- bzw. `POAManager`-Instanz des Serverprozesses ein Monitor geöffnet. Sind nur einige der Monitore für den Anwender von Interesse, kann er die nicht benötigten einzeln schließen und jederzeit über einen POA-Manager-Monitor wieder hervorholen.

An dieser Stelle sei noch bemerkt, daß die Aktualisierung der Oberfläche natürlich Zeit kostet. Ein einfacher, von einem POA verarbeiteter Ping-Aufruf ist ungefähr um den Faktor 4 langsamer als mit ausgeschaltetem Monitoring. Bei umfangreicheren Berechnungen pro Operation sieht dieses Verhältnis günstiger aus. Dieser Fakt verzerrt natürlich die Ergebnisse hinsichtlich der Belastung von Request-Queue und Thread-Pool. Man sollte diesen Umstand zumindest im Hinterkopf behalten, wenn man den visuellen Eindruck von der Ressourcenauslastung gewinnt.

6.5. Evaluation

Die Implementierung eines POAs für JacORB hat in vollem Umfang die Umsetzbarkeit der POA-Spezifikation gezeigt. Ohne die Oberflächenfunktionalität besteht sie insgesamt aus 36 Klassen und Schnittstellen. Die am Anfang dieses Kapitels formulierten Entwurfsziele konnten im wesentlichen erreicht werden. Die spezifizierte POA-Funktionalität wird durch die Implementierung vollständig unterstützt.

Die Abhängigkeiten zwischen den POA-Implementierungsklassen und anderen Teilen von JacORB wurden sehr gering gehalten. Der ORB referenziert selbst nur den Root-POA, und die von ihm benutzte, JacORB-interne Schnittstelle besteht aus nur sieben Operationen. Auf der anderen Seite benutzt die POA-Implementierung eine überschaubare Anzahl weiterer JacORB-Schnittstellen, die im Abschnitt 6.2.1 dargestellt wurden. Diese klare Trennung der Verantwortlichkeiten erleichtert die unabhängige Entwicklung an einzelnen JacORB-Komponenten. Auch ein Austauschen des POA-Implementierung oder der Einsatz eines weiteren OAs wäre mit wenig Aufwand realisierbar.

Eine besondere Stärke der POA-Implementierung liegt in der vollen Ausnutzung der Threading-Fähigkeiten von Java. Die Implementierung des *ORB Controlled Model*s erlaubt die nebenläufige Abarbeitung von Anfragen. Durch das eingesetzte Thread-Pool-Management handelt es sich zugleich um eine sehr effiziente Implementierung. Das Zulassen dieser Nebenläufigkeit und der potentielle Einsatz von JacORB in einer *multi-threaded* Serveranwendung erfordern an vielen Stellen eine Synchronisation der POA-Prozesse. Diese Synchronisationsprobleme wurden durch die Implementierung gelöst.

Ein Entwurfsziel bestand darin, die Anpaßbarkeit der POA-Implementierung an die unterschiedlichen Anforderungen einer Serveranwendung zu gewährleisten. Die wichtigsten Implementierungsparameter können für einen Serverprozeß konfiguriert werden. Der Anwender hat durch die Nutzung von Java-Properties die Möglichkeit, auf die bereitgestellte Funktionalität und die verwendeten Systemressourcen Einfluß zu nehmen (siehe Abschnitt 6.2.2). Erinnert sei nur an das konfigurierbare Thread-Pool-Management. Die gegebenen Konfigurationsmöglichkeiten orientieren sich aber an dem durch die POA-Spezifikation festgelegten Rahmen. Darüber hinausgehende substantielle Erweiterungen, wie bsw. die Konfiguration eines eigenen Thread-Managers, werden nicht geboten. Bei der Realisierung einer ersten POA-Version für JacORB stand die Umsetzung der spezifizierten Funktionalität und die Gewährleistung der Portabilität einer Serveranwendung im Vordergrund. Zusätzliche und damit JacORB-spezifische Erweiterungen bleiben zukünftigen Versionen vorbehalten.

Für die Weiterentwicklung von JacORB ist die Erweiterbarkeit der POA-Implementierung von Bedeutung. Der Implementierungsentwurf für den POA ist relativ modular aufgebaut. Die Kernfunktionalität ist auf elf Implementierungsklassen aufgeteilt.[3] Einige dieser Komponenten bilden eine existenzabhängige 1:1-Beziehung (Komposition). Bsw. wird die Anfragebearbeitung komplett aus der POA-Klasse ausgelagert und unter die Regie des Request-Controllers gestellt. Dieser benutzt Komponenten, deren Existenz an die eigene gebunden ist. Beim Entwurf dieser Implementierungsklassen spielten deren Wiederverwendbarkeit eine untergeordnete Rolle. Einige Kernkomponenten sind relativ stark aneinander gekoppelt. Diese enge Kopplung ist vornehmlich ein Resultat der komplexen Synchronisationsanforderungen. Das Ersetzen einzelner Implementierungsklassen durch eine spezialisierte Version dürfte nur in wenigen Fällen Sinn machen. Der endgültige Charakter der POA-Spezifikation trägt seinen Anteil an diesem Entwurf bei. Aber selbst wenn eine neue Policy eingeführt werden würde, hängt es sehr stark von ihrer Semantik ab, welche Systemteile davon betroffen wären und wie aufwendig eine Anpassung der Implementierung wäre. Die aus meiner Sicht wahrscheinlichste Änderung für den POA, ob im Zuge des Voranschreitens des Spezifikationsprozesses oder als JacORB-eigene Erweiterung, betrifft eine erweiterte Konfigurierbarkeit des Thread-Managements. Die Auslagerung des Thread-Pool-Managements in eine eigene Klasse eröffnet die Möglichkeit, in einer zukünftigen JacORB-Version zwischen einer Standard- und

[3] Nicht einbezogen sind die in den Unterpaketen versammelten Ausnahme-, Policy-, Hilfs- und Oberflächenklassen.

einer vom Anwendungsentwickler bereitgestellten Implementierung eines Thread-Managers zu wählen.

Größere Bedeutung für die Weiterentwicklung von JacORB besitzt die Öffnung des POAs für andere JacORB-Komponenten und die Möglichkeit, daß diese sich in POA-Prozesse einbringen können. Solche Erweiterungen werden durch das Konzept der POA-Ereignisse sehr gut unterstützt (siehe Abschnitt 6.3). Auch wenn die momentane Auswahl der Ereignisse willkürlich ist und nicht alle potentiell interessierenden Ereignisse einbezogen wurden, der Aufwand für eine spätere Anpassung ist gering. Neu eingeführte Ereignisse werden sofort von dem effizientem Benachrichtigungsmechanismus profitieren. Das Konzept der POA-Ereignisse hat sich bei der Aktualisierung der Benutzeroberfläche bestens bewährt. Als ein zweites Anwendungsbeispiel konnte die Aktualisierung der Informationen im ImR durch die Benachrichtigung des ORBs über die Erzeugung und Zerstörung von POA-Instanzen realisiert werden. Die Beteiligung des POAs am Lebenszyklus eines Objektes macht ihn für weitere Anwendungen interessant. Ein denkbarer Bereich wäre die Unterstützung des POAs bei der Realisierung von *Policy Domains*. Objekte können bei ihrer Erzeugung in eine oder mehrere Policy-Domains eingeordnet werden. Die mit einer Domain assoziierten Policies repräsentieren Regeln und Kriterien, die die Aktivitäten von Objekten einer Domain festlegen (z.B. *Security Policies*) [22]. *Object Services* könnten, wenn sie neu erzeugte Objekte mit einer Policy assoziieren wollen, an einer Objekterzeugung interessiert sein.

Durch die Nutzung der POA-Ereignisse konnten die POA-Implementierungsklassen weitestgehend unabängig von den Darstellungsklassen der Benutzeroberfläche entworfen werden. Es waren nur wenige Anpassungen der POA-Implementierungsklassen vorzunehmen (siehe Abschnitt 6.4). Durch diese lose Kopplung an die Darstellungsklassen brauchen keine Laufzeiteinbußen infolge vorhandener, aber ausgeschalteter Oberflächenfunktionalität inkauf genommen werden.

Der Entschluß zur Bereitstellung einer Benutzeroberfläche zur Visualisierung des POA-Zustandes und seiner Aktivitäten war nicht ganz so uneigennützig, wie es vielleicht erscheint. Vor allem die Möglichkeit, interaktiv auf den POA-Zustand Einfluß zu nehmen, war eine große Hilfe bei der Entwicklung am POA. In Verbindung mit einem ebenfalls interaktiv einstellbaren Test-Client konnte der POA so umfangreich getestet werden (siehe Abbildung B.1 im Anhang der Arbeit).

Die POA-Implementierung ist als Bestandteil von JacORB zusammen mit dem Quellcode und einem umfangreichen Beispiel zur Demonstration der POA-Funktionalität (mehrere verschiedene Server und der erwähnte Test-Client) im Internet frei verfügbar [12]. Eine erste Beta-Version von JacORB 1.0, in der der POA enthalten war, wurde im Juni 1999 veröffentlicht. Seitdem wird diese JacORB Version von einer breiteren Anwenderschaft verwendet. Eine Mailing-Liste [11] dient zum Austausch zwischen Anwendern und Entwicklern. Auf dieser Liste sind z.Zt. 172 JacORB-Anwender eingetragen.

Der Entwicklungsprozeß an JacORB ist mittlerweile soweit fortgeschritten, daß die aktuelle Version und insbesondere der POA zuverlässig arbeitet. Die Resonanz auf der Mailingliste bestätigt diese Aussagen. Das folgende Kapitel wird auch zeigen, daß es sich bei JacORB und damit auch bei der POA-Implementierung um eine effiziente Implementierung des CORBA-Standards handelt, die auch in den Fragen der Performance mit kommerziellen ORB-Produkten mithalten kann.

7. Vergleich existierender POA-Implementierungen

Die Situation

Die POA-Spezifikation ist nun schon seit eineinhalb Jahren veröffentlicht, und trotzdem hält sich die Anzahl vergleichbarer Java-ORBs, die eine POA-Implementierung bieten, in Grenzen. Meiner Meinung nach ist das auch ein Zeichen für die nicht geringe Komplexität, die bei der POA-Implementierung zu bewältigen ist. Selbst führende ORB-Hersteller haben noch keinen POA in ihren aktuellen ORB-Produkten integriert, und wenn, dann geschah dies erst in jüngster Zeit (z.B. beim *VisiBroker* von Inprise (Borland) in der Version 4.0 [9]). Ein anderer Grund dürfte sein, daß der Schritt in die POA-Generation für existierende, BOA-basierte ORB-Implementierungen (bis CORBA-Version 2.1) eine umfassende Überarbeitung ihrer serverseitigen ORB-Umgebung bedeutet. Zudem geht dieser Schritt mit einer Neugestaltung der Stub- und Skeleton-Architektur einher. Bsw. hat *IONA* [10], der bisher führende ORB-Hersteller, die komplette Neuentwicklung ihres ORB-Produktes *(Orbix2000)* angekündigt, das nach Fertigstellung CORBA-3.0-Konformität aufweisen soll. Ihr aktuelles ORB-Produkt, *OrbixWeb 3.2*, besitzt nur CORBA-2.1-Konformität. Es erlaubt also nur BOA-basierte Anwendungen. Die Liste der kommerziellen Java-ORB-Anbieter, die in ihren Produkten eine POA-Implementierung bereitstellen, ist also noch sehr kurz. Außer dem schon erwähnten VisiBroker von Inprise konnte ich noch zwei weitere Firmen ermitteln, die zumindest nach den eigenen Aussagen auf ihren Internetseiten ein CORBA-2.3-konformes Produkt anbieten (*JBroker 2.1* von ObjectEra [14] und *BlueORB 1.0* von Berry Software [1]).

Bei einem Blick auf die freie Softwarewelt läßt sich ein einziger weiteren Java-ORB finden, der eine POA-Untertützung bietet, und dessen Quellcode frei zugänglich ist. Der von Jerome Daniel entwickelte *JavaORB* [13] ist, ebenso wie JacORB, eine für jegliche Nutzung freie Implementierung des CORBA-2.3-Standards. Die im JavaORB enthaltenene POA-Unterstützung wurde parallel zu der in JacORB entwickelt, und sie ist, nach meiner Einschätzung zusammen mit der in JacORB, eine der ersten POA-Implementierungen in Java weltweit. Der für die nichtkommerzielle Nutzung freie und weit verbreitete ORB *ORBacus* [30] bietet noch keine POA-Unterstützung in seiner Java-Version. Ein POA ist nur in einer *Preview Release 4.0* für C++ enthalten. Die Java-Version basiert noch auf dem CORBA-2.0-Standard.

Da in diesem Kapitel Java-ORBs betrachtet werden, möchte ich die von Sun bereitgestellte CORBA-Unterstützung in ihrer *Java Platform 2 (JDK 1.2)* erwähnen [39]. Sun wird auf absehbare Zeit keine vollständige, CORBA-2.3-konforme ORB-Implementierung in ihr *Java Development Kit* (JDK) integrieren, um es Lizenznehmern von Java zu gestatten, eigene Implementierungen in ihren Java-Entwicklungsumgebungen bereitzustellen. Die im JDK enthaltene ORB-Funktionalität entspricht nur einer Teilmenge der CORBA-2.0-Funtionalität. Sun hat mit der *Remote Method Invocation* (RMI) Technologie eine eigene Verteilungsplattform in Java integriert. Einen Vergleich zwischen RMI und CORBA bietet [31].

Unter [5] findet man eine gut gepflegte Zusammenstellung existierender ORB-Implementierungen.

Auswahl der ORBs und des Testverfahrens

Ich werde zwei der erwähnten ORBs mit POA-Unterstützung herausgreifen und mit JacORB bzw. meiner POA-Implementierung vergleichen. Dabei soll nicht der Vergleich der gesamten ORB-Funktionalität im Vordergrund stehen, sondern der Schwerpunkt wird auf die POA-relevanten Details gelegt.

Die geringe Anzahl der in Frage kommenden ORBs legt eine Auswahl praktisch schon fest. Als Vertreter der kommerziellen Produkte habe ich VisiBroker von Inprise gewählt. Die Bewertung erfolgt auf der Grundlage der im Internet frei verfügbaren, zeitlich begrenzt nutzbaren, aktuellen Version 4.0. Da es sich bei Inprise um einen der führenden ORB-Hersteller handelt, hat JacORB in diesem Vergleich die Gelegenheit, sich mit einem weit verbreiteten Standard-ORB aus der CORBA-Welt zu messen.

Dabei soll festgestellt werden, ob JacORB mit VisiBroker zusammenarbeiten kann. Die Interoperabilität wird durch den gemischen Betrieb der ORBs mit wechselnden Client- bzw. Server-Rollen getestet. Da Java besondere Anforderungen an die Portabilität der generierten Stubs und Skeletons stellt und diese im Java-Mapping explizit festgelegt sind, besteht die Chance, daß jeder ORB mit fremden Stubs/ Skeletons arbeiten kann. Ich werde durch den wechselseitigen Austausch der Stubs- und Skeletons überprüfen, ob die vorgegebenen Konformitätsbekundungen zum aktuellen Standard der Wahrheit entsprechen.

Da mir bei einem kommerziellen Produkt der Blick in den Quelltext der POA-Implementierung verwehrt bleibt, blieb mir nur die Möglichkeit, VisiBroker durch eine geeignete Anwendung zu testen. Zu diesem Zweck habe ich ein Serverprogramm geschrieben, das insgesamt acht verschiedenen POA-Konfigurationen verwendet und damit ein weites Spektrum der Konfigurationsmöglichkeiten des POAs abdeckt. Im Anhang A ist der Quelltext des Serverprogramms aufgeführt. Damit können Aussagen über die unterstützte POA-Funktionalität getroffen werden.

Zusätzlich habe ich den für meine eigene Implementierung zu Testzwecken eingesetzten Test-Client (siehe Abbildung B.1) modifiziert, so daß dieser Performance-Daten für die von ihm durchgeführten Objektaufrufe ausgibt. Auszüge aus dem Quelltext des Test-Clients sind im Anhang B aufgeführt. Durch einen geeigneten Serverentwurf ist es dem Client möglich, eine über die Benutzeroberfläche einstellbare Anzahl von Objekten auf dem Server zu erzeugen und Anfragen an diese Objekte zu versenden. Nebenbei kann auch die Anzahl der Threads bestimmt werden, die Aufrufe auf einem Objekt durchführen. Um viele verschiedene und vor allem wirklichkeitsnahe Situationen nachbilden zu können, sind sowohl die Intervalle, in denen ein Objekt aufgerufen wird, als auch der Aufwand, der für einen Objektaufruf auf dem Server betrieben wird, beim Client einstellbar, bzw. variieren die Werte zufällig in dem eingestellten Bereich.

Die exakten Performance-Werte (Zeit bzw. durchschnittliche Zeit, die ein Aufruf dauert) sind an dieser Stelle zweitrangig. Durch mehrmalige Messungen unter vergleichbaren Bedingungen lassen sich aber trotzdem allgemeine Aussagen treffen. An diesem Punkt soll herausgefunden werden, ob die JacORB-Implementierung, und da ist nicht nur der POA einbezogen, dem Vergleich mit anderen Produkten standhält. Die Performance-Werte für unterschiedliche Client-Konfigurationen kombiniert mit Testausgaben der Servants auf dem Server geben Aufschluß über die Threading-Eigenschaften des POAs.

Zusammenfassend sollen folgende Bereiche getestet und verglichen werde:

- die Interoperabilität von JacORB durch den gemischen Betrieb der ORBs mit wechselnden Client- bzw. Server-Rollen,
- die Portabilität der generierten Subs/Skeletons durch den wechselseitigen Austausch der Stubs- und Skeletons,
- die Basisfunktionalität des ORBs/POAs durch 8 verschiedene POA-Konfigurationen,
- die Performance durch Messungen im Client,
- die Threading-Eigenschaften des POAs durch eine variable Anzahl aufrufender Client-Threads und Testausgaben der Servants.

Als zweiten Kandidaten, der sich einem ausführlichen Test unterziehen muß, habe ich den freien *JavaORB* von Jerome Daniel [13] ausgewählt. Er muß sich ebenfalls dem beschriebenen Testszenario unterziehen. Da er einen Blick in den Quellcode gestattet, sind aber weitergehende Untersuchungen möglich. Mit Hilfe meiner gesammelten Erfahrung bei der Umsetzung der POA-Spezifikation werde ich mein Augenmerk auf die "kritischen" Dinge lenken. Gleichzeitig soll aber auch der eigene Entwurf auf dem Prüfstand stehen, um eventuell vorhandene Anregungen aufgreifen zu können.

7.1. VisiBroker

VisiBroker ist in der Version 4.0 erst seit dem Dezember 1999 verfügbar. Dieser ORB kommt mit einer beeindruckenden Funktionalität daher. Die ORB-Runtime-Umgebung, die komplette Entwicklungsumgebung mit allen nötigen Werkzeugen zur Anwendungsentwicklung und Administration der ORB-Umgebung und eine grafische Oberfläche, die *VisiBroker Console*, können in einer 60-Tage-Evaluationsversions über das Internet geladen werden [9]. Die Installation lief ohne Probleme und die beigefügte Dokumentation ist reichhaltig. Es werden keinerlei Komponenten vermißt.

Als besondere Funktionalität wird eine *Smart Agent Architecture* geboten. *Smart Agents* können auf mindestens einer Station eines lokalen Netzes als Daemon-Prozeß auf einem *well-known* Port laufen. Sie kommunizieren untereinander und werden vom ORB benutzt, um persistente Objekte zu registrieren bzw. aufzuspüren und realisieren somit ein verteiltes Verzeichnis aller im Netz vorhandenen, aktiven persistenten Objekte. Der Entwickler hat über proprietäre Operationen der generierten Helper-Klassen die Möglichkeit, einen Client sehr komfortabel mit einer Referenz auf ein persistentes Objekt zu versorgen.

Ich möchte an dieser Stelle keine erschöpfende Übersicht über die Möglichkeiten von VisiBroker geben, sondern mich meinen Testerfahrungen zuwenden. Die Konfiguration des ORBs und das Ausprobieren eines ersten Beispielprogramms bereitete wiederum keinerlei Probleme. Beim nächsten Schritt, dasselbe Serverprogramm mit JacORB zu starten, trat eine Schwäche von VisiBroker hervor. Der fehlerhafte Beispielcode erzeugt einen persistenten POA mit geltendem **SYSTEM_ID** Policy-Wert und ruft darauf eine `activate_object_with_id()` Operation auf, um ein persistentes Fabrik-Objekt zu aktivieren, dessen Objekt-ID von der Anwendung festgelegt wird. Mit JacORB hatte diese Operation eine **BAD_PARAM** System-Exception zu Folge, da der POA erkannte, daß die ihm übergebene Objekt-ID nicht von ihm selbst oder einer früheren Instanz generiert wurde (vgl. Abschnitt 4.4.2). JacORB verwendet für diese Erkennung ein Wasserzeichen in seinen Objekt-IDs (vgl. Abschnitt 6.2.4). Laut Spezifikation ist der POA aber nicht verpflichtet, dies zu erkennen, demzufolge liegt der Fehler formal beim Serverprogramm. Nach der Verwendung des **USER_ID** Policy-Wertes lief das Beispiel auch mit JacORB.

Nun sollte meine Testanwendung mit VisiBroker zum Laufen gebracht werden. Dabei stellte sich aber heraus, daß der IDL-Compiler von VisiBroker keine Modulnamen in IDL-Dateien zuläßt, die von einem Klassenbezeichner verwendet werden (siehe Anhang A). Deshalb mußte erst einmal der Programmcode an die nun veränderten Paketnamen angepaßt werden, um diesen zusammen mit den generierten Stubs und Skeletons übersetzen zu können. Der darauffolgende Funktionstest lief ohne Beanstandungen durch, VisiBroker kam mit jeder im Testprogramm verwendeten POA-Konfiguration zurecht. Auch der Interoperabilitätstest, also die gemeinsame Verwendung von JacORB und VisiBroker in wechselnden Client- bzw. Server-Rollen, offenbarte keine Probleme.

Beim Portabilitätstest wurden die bisher getrennt für den jeweiligen ORB generierten Stubs und Skeletons ausgetauscht. Als positives Ergebnis bleibt festzustellen, daß VisiBroker sowohl in der Client- als auch in der Server-Rolle mit den von JacORB generierten Stubs und Skeletons zurechtkommt.

Im umgekehrten Fall offenbarten sich jedoch Probleme. Standardmäßig generiert VisiBroker Stubs, die eigene Schnittstellen und Klassen (`Object`, `ObjectImpl`) aus dem Paket `com.inprise.vbroker.CORBA` erweitern, und den generierten Helper-Klassen werden nichtstandardisierte Operationen hinzugefügt, die proprietäre ORB-Operationen nutzen. Wenn man den IDL-Compiler mit dem Schalter `-strict` startet, soll portabler Stub/Skeleton-Code generiert werden, der Schalter `-no_bind` unterbindet die Generierung der proprietären Helper-Operationen. Aber auch dann ließen sich die Skeletons nicht übersetzen, da sie eine VisiBroker-eigene Klasse benutzen, um die Auslieferungsfunktionalität bereitzustellen (`com.inprise.vbroker.CORBA.portable.MethodPointer`).

Die mit der `strict` Option generierten Stubs entsprechen dem Anschein nach dem aktuellen Java-Mappping. Die Helper-Klassen verwenden aber Schnittstellen des mit der CORBA-Version 2.3 eingeführten `CORBA_2_3`-Moduls. Diese werden von JacORB noch nicht unterstützt.

Zwar ließ sich mit dem Einbinden der VisiBroker-Runtime-Bibliothek alles übersetzen, ein JacORB-Server kann daraufhin auch problemlos gestartet werden, doch ein JacORB-Client versagt seinen Dienst, da er mit den von VisiBroker generierten Stubs/Helpers nicht klarkommt. Die Ursachen für diese Schwierigkeiten liegen bei JacORB. Die Implementierung von JacORB zeigt Schwächen bei der Unterstützung des aktuellen Java-Mappings. Es existieren neben den fehlenden Implementierungen der Schnittstellen des `CORBA_2_3`-Moduls auch noch Unzulänglichkeiten bei den vorhandenen Implementierungen. Bsw. generiert die JacORB-eigene Implementierung von `org.omg.CORBA.portable.Delegate` beim Aufruf von `_releaseReply()` noch eine **NO_IMPLEMENT** System-Exception. Diese Operation wird aber innerhalb standardkonformer Stubs verwendet.

Als Fazit des Portabilitätstests bleibt festzuhalten, daß die Portabiliät der Stubs/ Skeletons insgesamt nicht gewährleistet ist, VisiBroker die größere Nähe zum aktuellen Standard besitzt, und JacORB durch die fehlende Unterstützung einiger Schnittstellen die größeren Mängel aufweist.

Um die Threading-Eigenschaften von VisiBroker zu untersuchen und einige vergleichende Performance-Messungen vorzunehmen, habe ich einen JacORB-Client mit einem VisiBroker- oder, für den Vergleich, mit einem JacORB-Server laufen lassen. VisiBroker unterstützt vielfältige Konfigurationsmöglichkeiten zum Thread-Management. Es existieren zwei grundsätzliche Thread-Policies: *thread pooling* und *thread-per-session.* Standardmäßig wird *thread pooling* verwendet, wobei auch dort wieder weitere Konfigurationsmöglichkeiten bestehen, ähnlich dem Thread-Pool-Management in JacORB. Bei eingestelltem *thread-per-session* Modell wird pro Clientverbindung ein Thread zur Anfragebearbeitung bereitgestellt. Anfragen eines Clientprozesses teilen sich einen Bearbeitungsthread. Diese Möglichkeit gibt es in JacORB nicht.

Eine besondere Funktionalität von VisiBroker ist es, daß das Thread-Management für einzelne POAs einer Anwendung eingestellt werden kann. Dazu führt VisiBroker das Konzept von *Server Engines* ein, die einen logischen Bezeichner besitzen, vom Entwickler über Java-Properties konfiguriert werden und bestimmte Threading-Eigenschaften versammeln. Bei der Erzeugung eines POAs kann mit einer nichtstandardisierten Policy `SERVER_ENGINE_POLICY_TYPE` als Policy-Wert der Name der zu benutzenden *Server Engine* übergeben werden. Dieses Konzept geht über die von JacORB gebotenen Funktionalität hinaus. Besonders die Möglichkeit, das Thread-Management für einzelne POAs oder eine bestimmte Gruppe von POAs zu konfigurieren, ist eine sinnvolle Erweiterung. Die im Abschnitt 4.14 angesprochene Konfiguration eines eigenen Thread-Managers wird nicht geboten.

Die beschriebenen Möglichkeiten zur Konfiguration des Thread-Management habe ich nicht getestet, sondern vorzugsweise der mitgelieferten Dokumentation entnommen. Mit dem Einsatz der Standardwerte unterscheidet sich das von VisiBroker durchgeführte Thread-Management nicht von dem in JacORB. Anhand von Testausgaben im Servant wurde die korrekte Serialisierung der Servant-Aufrufe bei eingestelltem *Single Thread Model* überprüft.

Bei einem Blick auf die Performance-Werte trat Erfreuliches zu Tage. Die durchschnittliche Zeit für einen einfachen Aufruf auf dem Server (`compute()` Operation mit der Übertragung eines `int`) lag für den JacORB-Server generell leicht unter dem Wert des VisiBroker-Servers. Da das Client-Testprogramm bei seinen Messungen zwischen den Aufrufen, die eine POA-Aktivierung zur Folge haben, allen ersten Aufrufen die an ein Objekt gerichtet werden und allen Folgeaufrufen unterscheidet, konnte festgestellt werden, daß Aufrufe, die eine Aktivierung eines POAs nach sich ziehen, der mit einem Servant-Manager initialisiert wird (Server 1-3), bei einem VisiBroker-Server unverhältnismäßig lange dauern (ca. 16 Sekunden!). Ein Grund für dieses Verhalten wurde nicht klar.

Die Testergebnisse sind im Abschnitt 7.3 zusammengefaßt.

Fazit

Insgesamt kann festgestellt werden, daß die JacORB-Implementierung, und damit auch die POA-Implementierung, der von VisiBroker in den Fragen der Performance mindestens ebenbürtig ist. Außerdem hat JacORB seine Interoperabilität zu einem Standard-ORB unter Beweis gestellt. Bei der Nähe zum aktuellen CORBA-Standard sind bei beiden ORBs Schwächen zu verzeichnen, wobei diese bei JacORB schwerer wiegen. Sie verhindern die Portabilität der generierten Stubs/Skeletons. VisiBrokers POA-Implementierung unterstützt, ebenso wie JacORB, alle denkbaren Server- bzw. POA-Konfigurationen und bietet darüber hinaus eine Fülle von Konfigurationsmöglichkeiten, die über die in JacORB gebotenen hinausgehen. In Anbetracht der Tatsache, daß es sich bei VisiBroker um ein führendes kommerzielles ORB-Produkt handelt, und die Entwicklung an JacORB von wenigen Leuten vorangetrieben wird, war kaum ein anderes Ergebnis zu erwarten.

7.2. JavaORB

Der von Jerome Daniel entwickelte JavaORB ist, ebenso wie JacORB, eine frei verwendbare und mit dem Quellcode verfügbare CORBA-Implementierung. Nach den Angaben auf der Internetseite [13] unterstützt JavaORB in der Version 2.2.3 den aktuellen CORBA-2.3-Standard, inklusive der neusten IIOP-Version 1.2 mit einer Implementierung von bidirektionalem GIOP und einer Unterstützung des *Dynamic Management of Any Values* (DynAnys) und der *Object by Value* Spezifikation [27]. Neben einem *Name Service* stehen weitere Implementierungen von CORBA-Diensten zur Verfügung: ein *Event, Transaction*, *Collection* und ein *Notification Service*. Die Installation von JavaORB läuft ähnlich der von JacORB ab, mit dem Entpacken der Installationsdatei, der manuellen Anpassung einiger Konfigurationsdateien und Systempfade ist die Sache erledigt. Später mußte jedoch festgestellt werden, daß einige Werte vom ORB nicht korrekt eingelesen bzw. verarbeitetet werden (z.B. `VerboseLevel`). Als Dokumentation sind ein 30-seitiges Entwicklerhandbuch und zahlreiche Beispielprogramme in der Distribution enthalten.

Neben dem POA wird auch noch der BOA unterstützt. Die Aktivierung von Servern zur Bereitstellung persistenter Objekte wird durch einen *JavaORB Loader* realisiert, der sich mit seinen Fähigkeiten an der BOA-Spezifikation orientiert. Der *Loader* wird mit einem Port parametrisiert und auf dem Rechner gestartet, der den Server bereitstellt. Ein *Implementation Repository*, das systemweit genutzt werden kann, ist nicht vorhanden.

Zusätzlich zu den ebenfalls von JacORB bereitgestellten *Request-Level* und *Message-Level Interceptors* hat der Entwickler die Möglichkeit, durch die Implementierung eines JavaORB-Event-Listeners und einer Registrierung beim ORB sich über neu aufgebaute bzw. geschlossene Client-Verbindungen informieren zu lassen. Der Event-Listener bekommt dabei die Verbindungsdaten (Host, Port) übergeben. Kombiniert mit der Benutzung von *Interceptors* hat der Serverentwickler bsw. so die Möglichkeit, ein ausgefeiltes Speichermanagement zu implementieren. Objekte, die von einem Client referenziert wurden, können, nachdem der Client die Verbindung unterbrochen hat, deaktiviert werden (normalerweise hat eine Anwendung keine Möglichkeit zu erkennen, daß ein Objekt nicht mehr benötigt wird).

Zum Testverfahren selbst braucht nicht mehr viel gesagt werden, der Java-ORB mußte sich der gleichen, in den beiden vorherigen Abschnitt beschriebenen Prozedur unterziehen. Demzufolge möchte ich mich hier auf die Ergebnisse konzentrieren. Für eine eventuelle Problemsuche stand der Quellcode bereit.

Ein Interoperabilitätstest mit einer einfachen Beispielanwendung lief problemlos, JavaORB und JacORB können in unterschiedlichen Client/Server-Rollen zusammenarbeiten. Der darauffolgende Portabilitätstest deckte die schon im vorherigen Abschnitt angesprochenen Schwächen von JacORB auf. Die Testerfahrungen ähneln denen, die mit VisiBroker gemacht wurden. Die JavaORB-Implementierung und die generierten Stubs/Skeletons weisen eine größere Nähe zum aktuellen Standard auf.

Interessanterweise verhindern kleine Unterschiede im eingesetzten Java-Mapping auch die Portabilität der generierten Stubs/Skeletons zwischen JavaORB und VisiBroker. An dieser Stelle sei auch bemerkt, daß eine Interoperabilität zwischen JavaORB und VisiBroker nicht gegeben ist. Ein JavaORB-Client kann eine von VisiBroker generierte IOR nicht lesen, entsprechende Versuche hatten immer einen Java-Laufzeitfehler zur Folge (`StringIndexOutOfBoundException`). Auch der umgekehrte Fall funktioniert nicht, die Ursache des Problems konnte nicht festgestellt werden. Ein VisiBroker-Client bleibt hängen und gibt keinerlei Fehlermeldungen von sich.

Die bisherigen Tests wurden mit einer einfachen Beispielanwendung durchgeführt, da sich der Einsatz des eigentlichen Testprogramms problematisch gestaltete. Die Aktivierung eines POAs im Kontext eines lokalen Objektaufrufes und die Initialisierung dieses POAs mit einem Servant-Manager (innerhalb der Operation `unknown_adapter` des Adapter-Activators, siehe Anhang A Server 1-3) durch eine implizite Objektaktivierung beim Root-POA (Verwendung der `_this()` Operation) funktioniert nicht mit JavaORB. Das Serverprogramm bricht mit der Generierung einer System-Exception ab. Bei der anschließenden Fehlersuche stellte sich heraus, das JavaORB die Kontext-Informationen für Aufrufe auf lokalen Objekten (in diesem Fall auf dem Adapter-Activator) nicht verwaltet. Damit kann auch nicht der für den Aufruf auf dem Servant verantwortliche POA ermittelt werden. Dies wiederum hat zur Folge, daß eine implizite Objektaktivierung beim zuständigen POA nicht möglich ist. Erst nach einer Modifikation des Serverprogramms, bei der an den entsprechenden Stellen die implizite durch eine explizite Objektaktivierungen ersetzt wurde, konnten alle POA-Konfigurationen getestet werden. Dieser Funktionstest lief dann auch zufriedenstellend ab.

In diesem Zusammenhang wurde auch festgestellt, daß eine Optimierung für lokale Objektaufrufe zwar vorhanden ist, diese aber noch weitergehenden Einschränkungen als in JacORB unterliegt (nur für Objekte möglich, die in der AOM verwaltet werden, vgl. Abschnitt 6.2.5). Lokale Aufrufe werden ähnlich wie bei JacORB direkt an den POA weitergereicht. Durch die fehlende Kontext-Verwaltung für diese Aufrufe ist aber die Lokalitätstransparenz noch weniger gewährleistet, und dies führt zu ernsten Einschränkungen bei der Anwendungsentwicklung (Fehler bei der impliziten Objektaktivierung).

Aus der Dokumentation geht hervor, daß JavaORB zwei verschiedene ORB-bezogene und drei verschiedene objektbezogene Thread-Policies unterstützt. Diese können über JavaORB-eigene Operationen beim ORB eingestellt werden, oder es wird ein Standardwert verwendet, der auch über Java-Policies gesetzt werden kann. Auf der ORB-Ebene wird zwischen einem *thread pooling* bzw. *no thread pooling* unterschieden. JacORB verwendet standardmäßig ein *thread pooling.* Das Pool-Management unter JacORB ist ausgefeilter, es wird nicht nur eine Obergrenze zugelassen, sondern der Anwender kann auch eine Untergrenze für die Anzahl der vorgestarteten Threads definieren, so daß der Thread-Pool bei einem großen Anfragestrom eine Zeit wächst und bei einen bei einem Abflachen des Anfragestroms wieder schrumpft. Mit diesen Einstellungen kann der optimale Kompromiß zwischen dem Ressourcenverbrauch und der Effizienz bei der Bearbeitung stoßweise eintreffender Anfragen gefunden werden. Die Möglichkeit, das *thread pooling* ganz auszuschalten, halte ich nicht für entscheidend. Diesem Modus kann man sich auch mit einer geeigneten Wahl der Ober- bzw. Untergrenze annähern (bsw. mit einer Untergrenze von 1, vgl. Abschnitt 6.2.2).

Zusätzlich kann die Thread-Policy auf Objekt-Ebene variieren. Es kann ausgewählt werden, ob für jede Anfrage an ein bestimmtes Objekt ein weiterer Thread bereitgestellt wird (*per-request*), ob für alle Anfragen *eines* Clients ein einzelner Thread verwendet werden soll (*per-client*), oder ob clientunabhängig für alle Anfragen an dieses Objekt nur ein Thread bereitsteht (*single-threaded*). Zumindest sind Variante eins und drei sehr sinnvoll, denn damit kann der Zugriffsschutz für einzelne Objekte unabhängig von der gegebenen POA-Hierachie eingestellt werden (mit JacORB kann ein einzelnes Objekt nur dadurch geschützt werden, indem man es exklusiv bei einem *single-threaded* POA aktiviert). Erstaunlicherweise ist der Wert *per-client* als Standardwert eingestellt. VisiBroker bietet ähnliches mit der *thread-per-session* Policy, aber nicht als Standardwert. Die clientabhängige Serialisierung von Objektaufrufen ist wohl eher seltener gefragt, denn das Objekt selbst läßt sich damit nicht schützen. Zuviel Serialisierung auf der Serverseite birgt immer die Gefahren von *deadlocks*, man denke nur an einen Serverrückruf während eines Clientrückrufes.

Beim Testen der verschiedenen Thread-Policies mußte ich aber feststellen, daß bei der wichtigsten Policy, *per-request,* ein Laufzeitfehler auftritt (`NullPointer-Exception`), sobald eine komplexere Server-Konfiguration als ein einfaches Root-POA-Beispiel verwendet wird. Der einzige Policy-Wert, der wirklich funktioniert wie er soll, ist *per-client.* Auch bei dem Wert *single-threaded*, wird intern zuviel synchronisiert, Aufrufe von einem Client auf *unterschiedlichen* Objekten laufen immer serialisert ab. Erst die Aufrufe eines zweiten Clients erfolgen nebenläufig. Hier wird in der Dokumentation eindeutig mehr versprochen als gehalten. Die Ideen sind aber gut, auch in JacORB könnte man mit relativ geringem Aufwand eine objektbezogene Synchronisation einführen.

Die Performance-Werte von JavaORB sind deutlich schlechter als die von VisiBroker und JacORB (ca. Faktor 3 im lokalen Fall bei einem einfachen Aufruf der `compute()` Operation). Warum das so ist, sei dahingestellt. Die schon zuvor notwendige Fehlersuche im Quellcode verhagelte etwas die Lust, eine genaue Ursachenforschung zu betreiben. Bei einer Betrachtung der Implementierung der standardisierten POA-Operationen ist auch aufgefallen, daß die Spezifikation in einigen Fällen nicht korrekt umgesetzt wird. Bsw. wird in der `destroy()` Operation die Servant-Etherealization noch vor der Kind-POA-Zerstörung durchgeführt, die Semantik des `wait_for_completion` Flags ist nicht korrekt implementiert, und es wird nicht auf die Beendigung aller aktiven Anfragen gewartet (vgl. Abschnitt 4.12).

Auch die `create_POA()` Operation ist in keiner Weise mit dem Zerstörungsprozeß eines POAs synchronisiert (vgl. Abschnitt 4.3). Solche Nachlässigkeiten werden bei einfachen Anwendungsbeispielen kaum eine Rolle spielen bzw. nicht mal bemerkt werden. Bei komplexeren Anwendungen sind diese Feinheiten aber für das saubere Herunterfahren eines POAs und eine spätere Reaktivierung unter gleichem Namen von Bedeutung.

Fazit

Bei dem Test von JavaORB entstand der Eindruck, daß diese ORB-Implementierung oberflächlich betrachtet viel Funktionalität bietet, wobei aber, wenn man genauer hinschaut, zu viele Fehler und Ungenauigkeiten enthalten sind. Die Standardkonformität hinsichtlich der implementierten Schnittstellen ist weiter entwickelt als in JacORB. Anscheinend hat man aber das Testen der implementierten Funktionen vernachlässigt. Außerdem bestehen Defizite in der korrekten Umsetzung der standardisierten Funktionen (POA-Funktionalität). Eine ernsthafte Anwendungsentwicklung mit komplexeren Serverkonfigurationen ist aus diesem Grund nicht möglich. Der Blick auf die Performance-Werte ist ebenfalls enttäuschend. Einen sehr guten Eindruck hat nur der IDL-Compiler von JavaORB gemacht. Die generierten Stubs/Skeletons entsprechen dem aktuellen Java-Mapping.

JacORB fehlt in diesem Vergleich sicher noch etwas die Nähe zum aktuellen Standard, dafür ist die vorhandene Funktionalität wesentlich ausgereifter.

7.3. Testergebnisse

In diesem Abschnitt möchte ich die Testergebnisse in zwei Übersichten zusammenführen. In der Abbildung 7.1 werden die angesprochenen Eigenschaften der ORBs vergleichend dargestellt. Den Punkt "Portabilität der generierten Stubs/Skeletons" habe ich nicht in diese Übersicht aufgenommen, da diese momentan nicht gegeben ist. Bei den nichtstandardisierten Erweiterungen hinsichtlich des Thread-Managements muß man in Betracht ziehen, daß eine Serveranwendung, die diese benutzt, in ihrer Portabilität möglicherweise eingeschränkt ist. Bei der von mir entwickelten POA-Implementierung konnte es nur darum gehen, die standardisierte Funktionalität bereitzustellen.

In der Abbildung 7.2 ist ein Beispiel für die gemessenen Performance-Werte dargestellt. Die dabei verwendete Serverkonfiguration stellt vermutlich die höchsten Anforderungen an den ausliefernden Server-ORB unter den von mir getesten Server-Beispielen. Die POA-Hierarchie bestand insgesamt aus drei verwandten POAs. Nach der Server-Initialisierung existierten nur zwei POA-Instanzen, die dritte wurde als Folge eines ersten Aufrufes von `createFoo()` von einem Adapter-Activator erzeugt, der diese mit einem Servant-Locator initialisiert (siehe Anhang A, Server 3).

Die erste Zeile der Tabelle zeigt die im Client gemessene Zeit, die für diesen ersten Aufruf benötigt wurde. Alle Aufrufe von `compute()` erfolgten auf einem einzelnen Objekt, dessen Implementierung durch den Servant-Locator bereitgestellt wurde. Die zweite Zeile der Tabelle zeigt die durchschnittliche, im Client gemessene Zeit eines solchen Aufrufes, gemittelt über 100.000 Aufrufe. Bei einem Aufruf von `compute()` wurde ein `int` überragen. Für die Bearbeitung der Operation auf dem Server brauchte keine zusätzliche Zeit aufgewendet werden *(ping)*.

	VisiBroker	JavaORB	JacORB
Interoperabilität	(x) [1]	(x) [1]	x
Standardfunktionalität des POAs	x	(x) [2]	x
Multithreading	x	(x) [2]	x
Erweitertes Thread-Management			
thread pooling	POA-Level	ORB-Level	POA-Level [3]
per-session	POA-Level	Object-Level	-
per-object	-	(x) [2]	-
thread manager	-	-	-
Optimierung lokaler Objektaufrufe	x	x	x
Persistenz	x	x	x
Implementation Repository	x	-	x

[1] keine Interoperabilität zwischen VisiBroker und JavaORB

[2] fehlerhafte Implementierung

[3] einheitlich für einen Serverprozeß

Abbildung 7.1.: Vergleich der ORB-Funktionen

Die Messung erfolgte lokal (Client und Server) auf einem Rechner (Pentium II, Windows NT), um störende Faktoren durch eine Netzverbindung auszuschließen. Trotzdem handelte es sich um Fernaufrufe, da Client und Server in getrennten Prozessen liefen. Damit wurden lokale Optimierungen für diese Aufrufe ausgeschaltet. Das Marshaling mußte komplett durchgeführt werden, was sicher einen nicht unerheblichen Teil der Aufrufzeit verbrauchte. Für den Client wurde generell JacORB verwendet, um den Clientanteil bei der Rechnerbelastung konstant zu halten. Mir kam es darauf an, soweit wie möglich die reine Server-Performance zu messen. Die Messung wurde mehrmals durchgeführt. Die in der Abbildung angegebenen Werte liegen im Mittel der Messungen.

	VisiBroker	JavaORB	JacORB
1. Aufruf von `createFoo()` mit POA-Aktivierung und Servant-Manager-Initialisierung in msec	16204,00	140,00	210,00
Aufruf von `compute()` vermittelt über einen Servant-Locator in msec	1,52	5,40	1,48

Abbildung 7.2.: Vergleich der Performance-Werte (Servant-Locator)

8. Zusammenfassung

Im Rahmen dieser Diplomarbeit habe ich einen *Portable Object Adapter* für JacORB implementiert. Diese Arbeit hatte entscheidenden Anteil an dem Bestreben, JacORB an den CORBA-Standard 2.3 anzupassen.

Die CORBA-Spezifikation hat mit der Einführung der Version 2.2 die Serverarchitektur in einem wesentlichen Punkt geändert. Der ursprünglich als Standard-Objekt-Adapter vorgesehene *Basic Object Adapter* wurde durch den POA ersetzt. Das Kapitel 3 hat gezeigt, daß dieser Schritt zwingend notwendig war, da sich die BOA-Spezifikation als völlig unzureichend herausstellte, und die verschiedenen Interpretationen der ORB-Hersteller eine portable Entwicklung von CORBA-Anwendungen behinderte.

Der zweite Anlauf der OMG, einen Standard-Objekt-Adapter zu definieren, ist entschieden besser gelungen. Dabei wurde der Versuch unternommen, einen OA zu definieren, der ein möglichst weites Spektrum von Anforderungen, die eine Serveranwendung an einen OA stellen kann, erfüllt. Der POA konzentriert sich auf die Verwaltung von Servants und die Zustellung von Objektaufrufen, und er stellt damit das Bindeglied zwischen dem ORB und den von der Anwendung bereitgestellten Objekt-Implementierungen dar. Die Aktivierung von Serverprozessen hat man nun, im Gegensatz zur BOA-Spezifikation, anderen ORB-Komponenten überlassen. Damit kann der POA einem Anwendungsentwickler eine sehr flexible Schnittstelle für die Serverentwicklung bereitstellen. Durch die Fülle der Konfigurationsmöglichkeiten des POAs kann der Entwickler in vielfältiger Weise auf die verwendeten Systemressourcen Einfluß nehmen. Sie erlauben ihm effiziente und erweiterbare Serveranwendungen zu schreiben.

Im Kapitel 4 wurden die Fähigkeiten des POAs beschrieben. Es wurden auch die Bezüge zur Anwendungsentwicklung hergestellt, um die Einsatzmöglichkeiten des POAs kennenzulernen. Dieses Kapitel schließt mit der Feststellung, daß der POA die an ihn gestellten Erwartungen im Großen und Ganzen erfüllt. Vor allem ist die Spezifikation vollständig und läßt somit keine Interpretationsspielräume für die ORB-Hersteller. Die vermißte Möglichkeit eines ausgefeilteren Thread-Managements als es der POA bietet, bildete den einzigen Kritikpunkt.

Für die Implementierung eines POAs waren neben dem Verständnis der POA- und der CORBA-Kern-Spezifikation auch alle Belange des serverseitigen IDL/Java-Mappings von Bedeutung. Dieses definiert eine Stub- und eine Skeleton-Architektur, die, im Unterschied zu anderen Language-Mappings, die Portabilität der vom IDL-Compiler generierten Stubs/Skeletons sicherstellen soll. Das Kapitel 7 hat gezeigt, daß diese Portabilität nur dann gewährleistet ist, wenn sich die verschiedenen ORB-Hersteller an den Standard halten. Selbst wenn Hersteller das von sich behaupten, existieren zur Zeit immer noch Probleme bei der Umsetzung des Standards. Man muß der Situation aber zugute halten, daß die wenigen ORB-Hersteller, die eine

POA-Implementierung bieten, erst in jüngster Zeit ihr Produkt fertiggestellt haben. Auch JacORB hat in dieser Hinsicht noch Probleme. Es muß aber zwischen der Portabilität der generierten Stubs/Skeletons und der Portabilität des Anwendungscodes unterschieden werden. Letzere ist mit POA-basierten Anwendungsprogrammen gewährleistet.

Die Gefahr, daß sich die Entwicklung vom BOA wiederholt, daß Hersteller mit eigenen Erweiterungen des Standards die Portabilität untergraben, um den Komfort für den Anwender zu erhöhen oder die Funktionalität zu erweitern, ist natürlich gegeben. Zumindest bietet VisiBroker eine Möglichkeit, auf die eigenen Optimierungen bei der Generierung der Stubs/Skeletons zu verzichten. Letztendlich wird es diese Tendenzen immer geben. Mit der Einführung des POAs wurde lediglich der Abstand zwischen der spezifizierten Funktionalität zu den proprietären Erweiterungen der Hersteller für eine gewisse Zeit verkürzt. Was man vorher nur mit herstellerspezifischen Erweiterungen erreichen konnte, kann man nun zum großen Teil auf standardisierte Weise realisieren. Die Einführung einer nichtstandardisierten Thread-Policy, wie bsw. bei VisiBroker, sind nur ein Beispiel dafür, daß es proprietäre Erweiterungen des Standards auch in Zukunft geben wird.

Durch die fortlaufende Anpassung des Standards wird die Anwendungsentwicklung mit CORBA immer komplexer. Immer mehr Funktionalität wird hinzugefügt, wie bsw. die *Objects by Value* [27] und die *CORBA Messaging* Spezifikationen [23] zeigen. Dies betrifft auch die Veränderungen in der Serverarchitektur durch die Einführung des POAs. Die weitreichenden Möglichkeiten bei der Serverentwicklung zur dynamischen Verwaltung von Objekt–Implementierungen gehen einher mit einem Verlust an Verteilungstransparenz. Anwendungen, die diese Möglichkeiten ausreizen wollen, müssen als verteilte Anwendungen entworfen und programmiert werden. Der Entwickler einer Serveranwendung findet zur Zeit wenig Unterstützung durch Literatur, die auf Entwurfsmustern für POA-basierte Anwendungen eingeht. Gerade ein Buch [8] beschäftigt sich mit diesem Thema.

Die POA-Implementierung für JacORB hat die Umsetzbarkeit der POA-Spezifikation gezeigt. Die spezifizierte Funktionalität ist vollständig implementiert und der POA arbeitet zuverlässig. Die Resonanz auf einer Mailingliste für JacORB-Entwickler bestätigt diese Aussagen. JacORB steht seit Juni 1999 mit einer ersten Version des POAs im Internet frei zur Verfügung. Allein von der neuesten Version wurden bisher über 2000 Kopien vom FTP-Server der FU geladen [12].

Im Kapitel 6 wurde ausführlich auf die getroffenen Entwurfsentscheidungen und Implementierungsdetails eingegangen. Die Schwierigkeiten bei der Implementierung resultierten vor allem aus den zu lösenden Synchronisationsproblemen durch den Einsatz des POAs in einer *multi-threaded* Umgebung, den komplexen nebenläufigen Prozessen, in denen der POA involviert ist und den vielfältigen Konfigurationsmöglichkeiten des POAs. Die Abhängigkeiten zwischen dem POA und anderen Teilen von JacORB wurden auf ein Minimum reduziert. Der ORB und der POA kommunizieren über eine sehr schmale Schnittstelle. Ein Austauschen des Objekt-Adpaters wäre mit wenig Aufwand möglich.

Bei dem Entwurf des POAs wurde auf seine Erweiterbarkeit geachtet. JacORB-Entwicklern, die an den vom POA verarbeiteten Information interessiert sind, wird eine Schnittstelle zum Einfügen von Funktionalität geboten. Dieses Konzept der POA-Event-Listener ist so allgemein gehalten, daß es als flexibler und universeller Erweiterungsmechnismus für JacORB angesehen werden kann.

Zusätzlich zur POA-Implementierung wurde eine graphische Benutzeroberfläche bereitgestellt, die die Aktivitäten des POAs visualisiert. Durch die Realisierung als POA-Event-Listener kann diese Oberflächenfunktionalität dynamisch hinzugefügt bzw. entfernt werden.

JacORB gehört zu den wenigen existierenden Java-ORBs, die eine POA-Implementierung besitzen. Selbst kommerzielle Produkte verfügen, wenn überhaupt, erst seit jüngster Zeit über einen POA. Als Vertreter eines Standard-ORB-Produktes wurde VisiBroker getestet (siehe Kapitel 7). Die von VisiBroker gebotene POA-Funktionalität konnte in dem Test überzeugen. Sie war mit der von JacORB vergleichbar.

Der einzige weitere, frei verfügbare Java-ORB, der ebenfalls eine POA-Implementierung bietet, ist *JavaORB*. Dieser wurde parallel zu JacORB entwickelt. Der Vergleich hat aber gezeigt, daß JavaORB, und insbesondere seine POA-Implementierung, noch weitreichende Schwächen besitzt. Eine ernsthafte Anwendungsentwicklung ist damit z.Zt. nicht möglich.

JacORB hat in den durchgeführten Tests seine Interoperabilität, eine sehr gute Performance und die vollständige Implementierung der POA-Funktionalität unter Beweis stellen können. Mit JacORB existiert damit eine freie Java-Implementierung des aktuellen CORBA-Standards, die die Entwicklung verteilter, portabler und effizienter CORBA-Anwendungen erlaubt.

A. Quellcode des Test-Servers

Das an dieser Stelle aufgeführte Serverprogramm wurde zum Testen der POA-Funktionalität bei dem von mir durchgeführten ORB-Vergleich eingesetzt (siehe Kapitel 7). Auszüge aus dem verwendeten Clientprogramm sind im Anhang B aufgeführt.

Das Serverprogramm bietet eine Auswahl zwischen sechs verschiedenen Serverkonfigurationen, die per Kommandozeilenparameter festgelegt werden. Die Initialisierung des Servers ist für alle Konfigurationen gleich, ein persistentes `FooFactory`-Objekt wird bei einem `FooFactory`-POA aktiviert und die IOR dem Client über eine Datei zur Verfügung gestellt (eine Registrierung beim Namensdienst wäre ebenso möglich). Das `FooFactory`-Objekt hat die Aufgabe, einen Client mit der gewünschten Anzahl von Foo-Objektreferenzen zu versorgen. Diese werden von einem dritten POA verwaltet (`Foo`-POA). Die sechs verschiedenen Server unterscheiden sich nur in der Konfiguration dieses `Foo`-POAs.

Der erste Aufruf von `createFoo()` auf dem `FooFactory`-Objekt bewirkt die Aktivierung und Initialisierung des `Foo`-POAs. Dafür wird ein Adapter-Activator beim `FooFactory`-POA registiert. Die `FooFactory` erzeugt je nach Serverart Objektreferenzen bzw. aktiviert `Foo`-Objekte beim `Foo`-POA.

Server 1 Dieser Server besitzt einen Servant-Activator, der `Foo`-Objekte verwaltet. Die `FooFactory` erzeugt nur Objektreferenzen. Für alle ersten Aufrufe auf einem `Foo`-Objekt wird ein `FooImpl`-Servant vom Servant-Activator inkarniert. Alle folgenden Aufrufe können vom POA über die AOM vermittelt werden.

Server 2 Wie Server 1, mit dem Unterschied, daß einige Aufrufe von `Foo`-Objekten vom Servant-Activator an ein beim Root-POA aktiviertes `Foo`-Objekt weitergeleitet werden (Test des Location-Forwarding-Mechanismus). Spätere Aufrufe auf den betroffenen Objekten werden vom Root-POA verarbeitet.

Server 3 Der `Foo`-POA wird mit einem Servant-Locator initialisiert. Die `FooFactory` erzeugt nur Objektreferenzen. Aufrufe auf einem `Foo`-Objekt haben die Bereitstellung eines `FooImpl`-Servants für die Bearbeitung einer Anfrage zur Folge.

Server 4 Der `Foo`-POA benutzt einen Default-Servant, um Anfragen an `Foo`-Objekte zu vermitteln. Die `FooFactory` erzeugt nur Objektreferenzen.

Server 5 Dieser Server ähnelt dem Server 4 mit dem Unterschied, daß der `Foo`-POA für die Abbildung der Anfragen eine AOM benutzt. Aufrufe auf `Foo`-Objekten werden aber trotzdem auf denselben Servant abgebildet, da alle Objekte von der `FooFactory` mit diesem einen Servant aktiviert werden (**MULTIPLE_ID**-POA).

Server 6 Bei diesem Server wird der `Foo`-POA als *single-threaded* konfiguriert. Ansonsten entsprechen die Policy-Werte denen des Root-POAs. `Foo`-Objekte werden durch die `FooFactory` implizit aktiviert. Mit dieser POA-Konfiguration wurde die korrekte Serialisierung von Objektaufrufen getestet.

A.1. IDL-Interfaces

```
module foox {

   interface Foo {
      void compute(in long l);
   };
   interface FooFactory {
      Foo createFoo(in string id);
      string getServerDescription();
   };
};
```

A.2. Server-Klasse

```
package user_poa;

import org.omg.CORBA.ORB;
import org.omg.CORBA.Policy;
import org.omg.PortableServer.*;
import java.io.*;

public class Server {
   public final static String [] DESCRIPTIONS = {
                  "Servant Activator",
                  "Servant Activator & location forward",
                  "Servant Locator",
                  "Default Servant & RETAIN",
                  "One servant multiple oid's",
                  "SINGLE_THREAD_MODEL"
   };
   public static int    kind;
   public static String factoryPOAName = "factoryPOA";
   public static String fooPOAName;
   public static String description;

   public static void main(String[] args)  {
      if( args.length != 1 || (kind = Integer.parseInt(args[0])) < 1 || kind > DESCRIPTIONS.length) {
         String str = "";
         for (int i=1; i<=DESCRIPTIONS.length; i++)
            str = i==DESCRIPTIONS.length ? str+i : str+i+"|";
            System.err.println("\n<usage: jaco [properties] user_poa.Server "+str+">\n");
            System.exit(0);
         }
         fooPOAName = "fooPOA"+kind;
         description = DESCRIPTIONS[kind-1];
         try {
            ORB        orb     = org.omg.CORBA.ORB.init(args, null);
            POA        rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            POAManager poaMgr  = rootPOA.the_POAManager();

            // create a user defined poa for the foo factory
            Policy [] policies = {
               rootPOA.create_id_assignment_policy(IdAssignmentPolicyValue.USER_ID),
               rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
            };
            POA factoryPOA = rootPOA.create_POA(factoryPOAName, poaMgr, policies);
            for (int i=0; i<policies.length; i++) policies[i].destroy();

            // implicit activation of an adpater activator on root poa
            factoryPOA.the_activator(new FooAdapterActivatorImpl()._this(orb));

            // explicit activation of the factory servant on factory poa
             FooFactoryImpl factoryServant = new FooFactoryImpl();
             factoryPOA.activate_object_with_id(new String("FooFactory").getBytes(), factoryServant);

            // create a stringified object reference
            PrintWriter pw = new PrintWriter(new FileWriter("f:/Develop/examples/fooFactory.ior"));
            pw.println(orb.object_to_string(factoryServant._this()));
            pw.close();

            // activate the poa manager
            poaMgr.activate();

            System.out.println("[ Server ready ]");
            orb.run();
```

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

A.3. Adapter-Activator

```
package user_poa;

import org.omg.PortableServer.*;
import org.omg.CORBA.Policy;

public class FooAdapterActivatorImpl extends AdapterActivatorPOA {
   public boolean unknown_adapter(POA parent, String name) {
      try {

         if (name.equals(Server.fooPOAName) && (Server.kind == 1 ||
                                                Server.kind == 2)) {
            Policy [] policies = {
               parent.create_request_processing_policy(RequestProcessingPolicyValue.USE_SERVANT_MANAGER),
               parent.create_id_assignment_policy(IdAssignmentPolicyValue.USER_ID),
               parent.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
            };
            POA newPOA = parent.create_POA(name, parent.the_POAManager(), policies);
            for (int i=0; i<policies.length; i++) policies[i].destroy();

            newPOA.set_servant_manager(new FooServantActivatorImpl(_orb())._this(_orb()));
            return true;

         } else if (name.equals(Server.fooPOAName) && Server.kind == 3) {
            Policy [] policies = {
               parent.create_request_processing_policy(RequestProcessingPolicyValue.USE_SERVANT_MANAGER),
               parent.create_servant_retention_policy(ServantRetentionPolicyValue.NON_RETAIN),
               parent.create_id_assignment_policy(IdAssignmentPolicyValue.USER_ID),
               parent.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
            };
            POA newPOA = parent.create_POA(name, parent.the_POAManager(), policies);
            for (int i=0; i<policies.length; i++) policies[i].destroy();

            newPOA.set_servant_manager(new FooServantLocatorImpl()._this(_orb()));
            return true;

         } else if (name.equals(Server.fooPOAName) && Server.kind == 4) {
            Policy [] policies = {
               parent.create_request_processing_policy(RequestProcessingPolicyValue.USE_DEFAULT_SERVANT),
               parent.create_id_uniqueness_policy(IdUniquenessPolicyValue.MULTIPLE_ID)
            };
            POA newPOA = parent.create_POA(name, parent.the_POAManager(), policies);
            for (int i=0; i<policies.length; i++) policies[i].destroy();

            newPOA.set_servant(new FooImpl("0"));
            return true;

         } else if (name.equals(Server.fooPOAName) && Server.kind == 5) {

            Policy [] policies = {
               parent.create_id_uniqueness_policy(IdUniquenessPolicyValue.MULTIPLE_ID),
               parent.create_implicit_activation_policy(ImplicitActivationPolicyValue.IMPLICIT_ACTIVATION)
            };
            POA newPOA = parent.create_POA(name, parent.the_POAManager(), policies);
            for (int i=0; i<policies.length; i++) policies[i].destroy();
            return true;

         } else if (name.equals(Server.fooPOAName) && Server.kind == 6) {
            Policy [] policies = {
               parent.create_thread_policy(ThreadPolicyValue.SINGLE_THREAD_MODEL),
               parent.create_implicit_activation_policy(ImplicitActivationPolicyValue.IMPLICIT_ACTIVATION)
            };
            POA newPOA = parent.create_POA(name, parent.the_POAManager(), policies);
            for (int i=0; i<policies.length; i++) policies[i].destroy();
            return true;

         } else {
            System.out.println("unknown poa name (AdapterActivator)");
         }
      } catch (org.omg.PortableServer.POAPackage.AdapterAlreadyExists aae) {
          aae.printStackTrace();
      } catch (org.omg.PortableServer.POAPackage.InvalidPolicy ip) {
          ip.printStackTrace();
      } catch (org.omg.PortableServer.POAPackage.WrongPolicy wp) {
          wp.printStackTrace();
      }
      return false;
   }
}
```

A.4. Servant-Activator

```
package user_poa;

import foox.*;
import org.omg.PortableServer.*;

public class FooServantActivatorImpl extends ServantActivatorPOA {
   private org.omg.CORBA.Object forwardRef = null;

   public FooServantActivatorImpl(org.omg.CORBA.ORB orb) {
      if (Server.kind == 2) {
         try {
            forwardRef = new FooImpl("0")._this(orb);
         } catch (Throwable e) {
            e.printStackTrace();
         }
      }
   }

   public void etherealize(byte[] oid, POA     adapter,
                                       Servant servant,
                                       boolean cleanup_in_progress,
                                       boolean remaining_activations) {
      String oidStr = new String(oid);
      System.out.println("[ etherialize servant for oid: "+oidStr+" ]");
   }

   public Servant incarnate(byte[] oid, POA adapter) throws ForwardRequest {
      String oidStr = new String(oid);
      int    oidInt = Integer.parseInt(oidStr);
      if (oidInt >= 1000) {
         if (Server.kind == 2 && (oidInt % 2) == 0) {
            System.out.println("[ forward request ]");
            throw new ForwardRequest(forwardRef);

         } else {
            System.out.println("[ incarnate servant for oid: "+oidStr+" ]");
            return new FooImpl(oidStr);
         }
      } else {
         throw new org.omg.CORBA.OBJECT_NOT_EXIST();

      }
   }
}
```

A.5. Servant-Locator

```
package user_poa;

import foox.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.ServantLocatorPackage.CookieHolder;

public class FooServantLocatorImpl extends ServantLocatorPOA {
   private FooImpl foo = new FooImpl("0");

   public void postinvoke(byte[] oid, POA adapter, String operation, java.lang.Object cookie, Servant servant) {
      String oidStr = new String(oid);
      if (!oidStr.equals(cookie)) {
         System.out.println("[ postinvoke "+operation+" for oid: "+oidStr+": cookie is unknown ]");
         throw new org.omg.CORBA.OBJECT_NOT_EXIST();
      }
   }

   public Servant preinvoke(byte[] oid, POA adapter, String operation, CookieHolder cookie) throws ForwardRequest {
      String oidStr = new String(oid);
      int oidInt = Integer.parseInt(oidStr);
      if (oidInt >= 1000 && operation.equals("compute")) {
         cookie.value = oidStr;
         return foo;
      }
      return null;
   }
}
```

A.6. FooFactory-Implementierung

```
package user_poa;

import foox.*;
import org.omg.PortableServer.*;
```

```
public class FooFactoryImpl extends FooFactoryPOA {

   private static FooImpl single = new FooImpl("0");

   public POA _default_POA() {
      POA rootPOA = super._default_POA();
      try {
         return rootPOA.find_POA(Server.factoryPOAName, false);

      } catch(org.omg.PortableServer.POAPackage.AdapterNonExistent e) {
         e.printStackTrace();
         return rootPOA;
      }
   }

   public Foo createFoo(String id) {
      Foo result = null;
      try {
         POA fooPOA = _default_POA().find_POA(Server.fooPOAName, true);

         if (Server.kind == 1 || Server.kind == 2 || Server.kind == 3) {
            byte[] oid = id.getBytes();
            result = FooHelper.narrow(fooPOA.create_reference_with_id(oid, FooHelper.id()));

         } else if (Server.kind == 4) {
            result = FooHelper.narrow(fooPOA.create_reference(FooHelper.id()));

         } else if (Server.kind == 5) {
            result = FooHelper.narrow(fooPOA.servant_to_reference(single));
            /*** for testing the colocation optimization ***/
            System.out.println("[ invocation of the colocated Foo object ... ]");
            result.compute(0);
            System.out.println("[ ... done ]");
            /***********************************************/
         } else {
            result = FooHelper.narrow(fooPOA.servant_to_reference(new FooImpl("0")));
         }
      } catch (Exception e) {
         e.printStackTrace();
      }
      System.out.println("[ Foo created with id: "+id+" ]");
      return result;
   }

   public String getServerDescription() {
      String myOid = "unknown";
      try {
         org.omg.PortableServer.Current current =
         org.omg.PortableServer.CurrentHelper.narrow(_orb().resolve_initial_references("POACurrent"));
         myOid = new String(current.get_object_id());

      } catch (org.omg.CORBA.ORBPackage.InvalidName in) {
         System.out.println("POACurrent not available!");
      } catch (org.omg.PortableServer.CurrentPackage.NoContext nc) {
         System.out.println("Cannot determine the context oid!");
      }
      System.out.println("[ "+myOid+": description requested (\""+Server.description+"\") ]");
      return Server.description;
   }
}
```

A.7. Foo-Implementierung

```
package user_poa;

import foox.*;

public class FooImpl extends FooPOA {
   private String id;

   public FooImpl(String _id) {
      id = _id;
   }

   public void compute(int time) {
      try {
         for (int i=1; i<=time; i=i+100) {
            System.out.print(time+" ");
            Thread.currentThread().sleep(100);
         }
      } catch (InterruptedException e) {
      }
   }
}
```

B. Auszüge aus dem Quellcode des Test-Clients

Die in Abbildung B.1 dargestellte Benutzeroberfläche des Test-Clients gestattet die Festlegung der Anzahl der auf dem Server zu erzeugenden `Foo`-Objekte. Die Referenz auf ein `FooFactory`-Objekt wird aus einer Datei gelesen (die Verwendung eines Namensdienstes wäre ebenso möglich).

Der Client generiert Anfragen an diese Objekte innerhalb eines Request-Generator-Threads. Es gibt auch die Möglichkeit, mehrere Request-Generatoren für ein einzelnes `Foo`-Objekt zu erzeugen. Anhand eines weiteren Parameters kann festgelegt werden, in welchem Intervall die Aufrufe von `compute()` durchgeführt werden. Zusätzlich lassen sich die Kosten einstellen, die ein Aufruf von `compute()` auf dem Server verursacht. Dieser Wert wird als Parameter der `compute()` Operation zum Server übertragen und von diesem berücksichtigt. Ein Wert von **0** verursacht in beiden Fällen keine Kosten. Bei größeren Werten wird ein Zufallskomponente einbezogen, um die Generierung und Verarbeitung von Anfragen realistischer zu gestalten.

Die Dauer der Aufrufe wird gemessen. Die Daten werden vom Client verwaltet und am Ende in einer Gesamtübersicht ausgegeben.

B.1. Client

```
package client;

import foox.*;
import java.io.*;

public class Client {
   public static int speed = 0;
   public static int cost = 0;
   public static String serverDescription = "description not available";

   private static TestFrame frame;
   private static FooFactory factory;
   private static Foo [] foos;
   private static RequestGenerator [] [] generators;

   ...

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

         // read the IOR from file and create a fooFactory object
         BufferedReader br = new BufferedReader(new FileReader("f:/Develop/examples/fooFactory.ior"));
         org.omg.CORBA.Object object = orb.string_to_object(br.readLine());
         factory = FooFactoryHelper.narrow(object);
         System.out.println("[ FooFactory created ]");

         // get the server description
         serverDescription = factory.getServerDescription();

         // set up the gui
         frame = new TestFrame();
         frame.setVisible(true);
      }
      catch (Exception e) {
         e.printStackTrace();
      }
   }
```

B.2. Request-Generator

```
package client;

import foox.*;

public class RequestGenerator extends Thread {
   boolean active = true;
   private Foo foo;
   private boolean firstObjectContact;

   public RequestGenerator(Foo _foo, boolean _firstObjectContact) {
      foo = _foo;
      firstObjectContact = _firstObjectContact;
   }

   public void run() {
      long startTime = 0;
      long stopTime = 0;
      while (active) {
         try {
            int costs = (int) (Math.random() * Client.cost);

            startTime = System.currentTimeMillis();
            foo.compute(costs);
            stopTime = System.currentTimeMillis();

            Client.addTime((int)(stopTime-startTime), costs, firstObjectContact);
            firstObjectContact = false;

         } catch (org.omg.CORBA.SystemException e) {
            System.out.println("system exception received: "+e);
         } catch (Throwable e) {
            System.out.println("exception received: "+e);
            e.printStackTrace();
         }
         try {
            if (Client.speed != 0) sleep((int) (Math.random() * Client.speed));
         } catch (InterruptedException e) {
         }
      }
   }
}
```

B.3. Benutzeroberfläche des Test-Clients

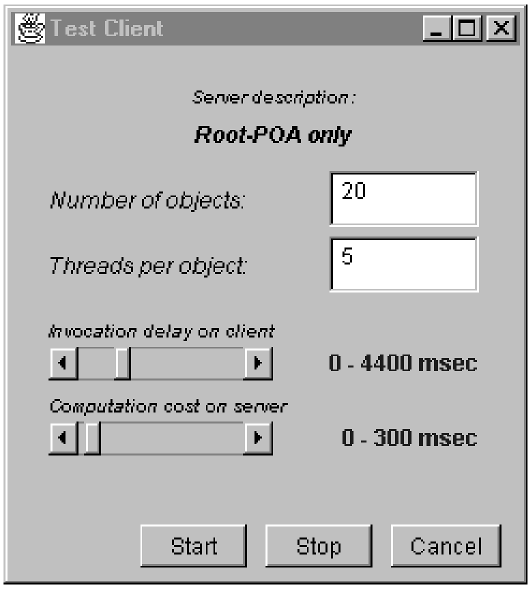


Abbildung B.1.: Screenshot des Test-Clients

Literaturverzeichnis

[1] Berry Software. *BlueORB*, January 2000.
`http://www.berry-software.com`

[2] Gerald Brose. JacORB — Design and Implementation of a Java ORB. In *Proc. International Conference on Distributed Applications and Interoperable Systems (DAIS'97)*, pages 143–154, Cottbus, Germany, September 1997. Chapman & Hall.

[3] Gerald Brose. *JacORB Programming Guide*, January 2000.
`http://www.inf.fu-berlin.de/~brose/jacorb/ftp/doc/ProgrammingGuide.pdf`.

[4] Gerald Brose and Boris Bokowski. Ein Object Request Broker für Java. *Informatik/Informatique, Zeitschrift d. schweiz. Informatikorganisationen*, 3:27–30, June 1997.

[5] Cetus Links. *Distributed Objects & Components: CORBA ORBs an other software*, January 2000.
`http://www.cetus-links.org/oo_object_request_brokers.html`

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison–Wesley, 1997.

[7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison–Wesley, 1996.

[8] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++.* Addison–Wesley, 1999.

[9] Inprise (Borland). *VisiBroker*, January 2000.
`http://www.inprise.com`

[10] IONA. *Orbix, OrbixWeb*, January 2000.
`http://www.iona.com`

[11] JacORB. *Developer Mailing List.* FU Berlin, January 2000.
`http://lists.spline.inf.fu-berlin.de/mailman/listinfo/jacorb-developer`.

[12] JacORB. *Gerald Brose.* FU Berlin, January 2000.
`http://www.inf.fu-berlin.de/~brose/jacorb`

[13] JavaORB. *Jerome DANIEL.* Distributed Object Group, January 2000.
`http://www.multimania.com/dogweb/Projects/JavaORB/javaorb.html`

[14] ObjectEra. *JBroker*, January 2000.
`http://www.objectera.com`

[15] OMG. *ORB Portability Enhancement RFP*, June 1995.
`ftp://ftp.omg.org/pub/docs/1995/95-06-26.pdf`

[16] OMG. *CORBA 2.0*, March 1996.
`ftp://ftp.omg.org/pub/docs/ptc/96-03-04.pdf`

[17] OMG. *A Discussion of the Object Management Architecture*, January 1997.
`http://www.omg.org/library/oma/oma-all.pdf`

[18] OMG. *Lifecicle Service*. COSS, December 1997.
`ftp://ftp.omg.org/pub/docs/formal/97-12-13.pdf`

[19] OMG. *ORB Portability Joint Submisssion*, May 1997.
`ftp://ftp.omg.org/pub/docs/orbos/1997/97-05-15.pdf`

[20] OMG. *Trading Service*. COSS, December 1997.
`ftp://ftp.omg.org/pub/docs/formal/97-12-23.pdf`

[21] OMG. *CORBA 2.2*, July 1998.
`ftp://ftp.omg.org/pub/docs/formal/98-07-01.pdf`

[22] OMG. *CORBA 2.3a Core Final Revision*, December 1998.
`ftp://ftp.omg.org/pub/docs/ptc/98-12-04.pdf`

[23] OMG. *CORBA Messaging Joint Revised Submission*, May 1998.
`ftp://ftp.omg.org/pub/docs/orbos/98-05-05.pdf`

[24] OMG. *Interoperable Naming Service*. Preliminary Specification, October 1998.
`ftp://ftp.omg.org/pub/docs/orbos/98-10-11.pdf`

[25] OMG. *PIDL & Pseudo-Objects Policy Paper*, January 1998.
`ftp://ftp.omg.org/pub/docs/ab/98-01-02.pdf`

[26] OMG. *CORBA 2.3 IDL to Java Mapping*, July 1999.
`ftp://ftp.omg.org/pub/docs/formal/99-07-53.pdf`

[27] OMG. *CORBA 2.3.1*, October 1999.
`ftp://ftp.omg.org/pub/docs/formal/99-10-07.pdf`

[28] OMG. *Issues for Object Request Broker 2.3a RTF*, May 1999.
`http://www.omg.org/issues/orb_revision.html`

[29] OMG. *Java Language to IDL Mapping*, June 1999.

[30] ORBacus. *Object Oriented Concepts*, January 2000.
`http://www.ooc.com/ob/`

[31] Robert Orfali and Dan Harkey. *Client/Server Programming with JAVA and CORBA*. Wiley, 1998.

[32] Frank Pilhofer. *Design and Implementation of the Portable Object Adapter*. Diplomarbeit am Fachbereich Informatik der Universität Frankfurt am Main, June 1999.

[33] Dale Rogerson. *Inside DCOM*. Microsoft Press, 1995.

[34] Douglas C. Schmidt and Steve Vinoski. Object Interconnections — Object Adapters: Concepts and Terminology. *SIGS*, 9(11), November 1997.

[35] Douglas C. Schmidt and Steve Vinoski. Object Interconnections — Using the Portable Object Adapter for Transient and Persistent CORBA Objects. *SIGS*, 10(12), April 1998.

[36] Jon Siegel. *CORBA Fundamentals and Programming*. Wiley, 1996.

[37] Dirk Slama, Jason Garbis, and Perry Russell. *Enterprise CORBA*. Prentice Hall, 1999.

[38] R. Srinivasan. *Remote Procedure Call Specifictaion Version 2*. RFC 1831, August 1995.

[39] Sun Microsystems. *Java*, January 2000.
`http://www.java.sun.com/`

[40] Steve Vinoski. New Features for CORBA 3.0. *CACM*, 41(10):44–52, October 1998.

[41] Andreas Vogel and Keith Duddy. *Java Programming with CORBA*. Wiley, 1998.

Erklärung

Hiermit versichere ich, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.