INSTITUT FÜR INFORMATIK

DER

LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Diplomarbeit

Konzeption eines Policy-basierten Konfigurationsmanagements für nomadische Systeme in Intranets

Bearbeiter:	Igor Radisic
Aufgabensteller:	Prof. Dr. H.-G. Hegering
Betreuer:	Stephen Heilbronner
Abgabetermin:	12. August 1998

INSTITUT FÜR INFORMATIK

DER

LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Diplomarbeit

Konzeption eines Policy-basierten Konfigurationsmanagements für nomadische Systeme in Intranets

Bearbeiter:	Igor Radisic
Aufgabensteller:	Prof. Dr. H.-G. Hegering
Betreuer:	Stephen Heilbronner
Abgabetermin:	12. August 1998

Ich versichere, daß ich die vorliegende Diplomarbeit selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 12. August 1998

. . . .

(Unterschrift des Kandidaten)

Zusammenfassung

Untersuchungen zeigen, daß die Anzahl der mobilen Endsysteme in Unternehmensnetzen zunimmt und deren Anteil am Gesamtkontingent stetig steigt. Somit erhöht sich der Bedarf, diese ebenfalls vom integrierten Netz- und Systemmanagement zu erfassen. Als *nomadische Systeme* werden allgemein Objekte bezeichnet, die sich im Rahmen eines Netzes bewegen. In dieser Arbeit werden aber v.a. die tragbaren Computervarianten, wie z.B. Notebooks, Laptops und Personal Digital Assistants, als nomadische Systeme angesehen. Grundsätzlich soll den nomadischen Systemen, u.a. durch ein entsprechendes Managementsystem, ermöglicht werden, sich an beliebigen, zur Verfügung stehenden Stellen eines Unternehmensnetzes anzuschließen, um die am Anschlußort lokal vorkommenden Ressourcen nutzen zu können. Die Aufgabe dieser Diplomarbeit besteht zunächst darin, ein Managementsystem zu entwickeln, das einerseits die Mobilität von nomadischen Systemen unterstützt und andererseits deren Zugriff auf Ressourcen in geeigneter Weise verwaltet.

Der Forschungsbereich des *Policy-basierten Managements* beschäftigt sich seit einiger Zeit intensiv mit der maschinenverständlichen Formulierung von vorhandenem Managementwissen und -vorgehen der Administratoren heterogener Netze. Dabei werden, ab einem bestimmten Abstraktionsgrad, Regeln in einer formalen Sprachen definiert, die Anweisungen enthalten, wie auf ein eingetretenes Ereignis reagiert werden soll. Auch beim Management nomadischer Systeme werden regelähnliche Aussagen benötigt, die festlegen, in welchem Umfang ein nomadisches System eine Ressource nutzen darf. Der Gedanke liegt nahe, das Management nomadischer Systeme mit dem Konzept des Policy-basierten Managements, v.a. in Hinblick auf die dabei bereits entwickelten Formalismen für *Policies*, zu verknüpfen.

Im Rahmen dieser Diplomarbeit wird ein Konzept für das Policy-basierte Management nomadischer Systeme entwickelt. Es wird die dafür notwendige Managementarchitektur auch unter dem Gesichtspunkt untersucht, bereits vorhandene Intranet-Infrastrukturdienste und Protokolle, wie z.B. *DHCP*-Server, Switches und Email-Hubs, miteinzubinden. Desweiteren wird eine neue, dem Management nomadischer Systeme angepaßte, formale Sprache für Policies definiert. Das entwickelte Konzept wird zudem in einer prototypischen Implementierung umgesetzt. Hierbei dient CORBA als Middleware und Java als Programmiersprache, wobei sich diese Kombination als besonders geeignet für das Management in einem heterogenen Systemumfeld herausstellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Das integrierte Management von verteilten Netz- und Systemkomponenten ist in den letzten Jahren um einen Aspekt reicher geworden. Durch die hohe Zunahme von mobilen Endgeräten in Unternehmen, wächst im Bereich der Netzbetreuung der Bedarf, diese ebenfalls in das Netz- und Systemmanagement mit einzubeziehen. Als *nomadische Systeme* (NoS) werden sich im Rahmen eines Unternehmensnetzes bewegende Objekte bezeichnet. In dieser Diplomarbeit werden in aller erster Linie sich bewegende Computer, wie z.B. Notebooks und Personal Digital Assistants (PDAs), als NoS angesehen. Grundsätzlich soll es NoS ermöglicht werden, sich an beliebigen, zur Verfügung stehenden Stellen an das Unternehmensnetz anzuschließen. Die Aufgabe eines Management Systems (MS) für NoS ist es nun, diesen Netzzugang in geeigneter Weise zu steuern und zu beeinflussen. Sicherheits- und Abrechnungsaspekte sind vor allem in Hinblick auf lokal vorkommende Ressourcen, wie z.B. einem Drucker, relevant, so daß es als sinnvoll erscheint, zwischen bekannten (privilegierten) und fremden NoS zu unterscheiden. Dem bekannten NoS könnte z.B. der Ausdruck von Dokumenten und somit die Nutzung dieses Druckers erlaubt werden, während dieses Recht dem fremden NoS verweigert bzw. eingeschränkt wird. Diese Rechte müssen zunächst in geeigneter Weise formuliert werden, um sie im nächsten Schritt durchsetzen zu können.

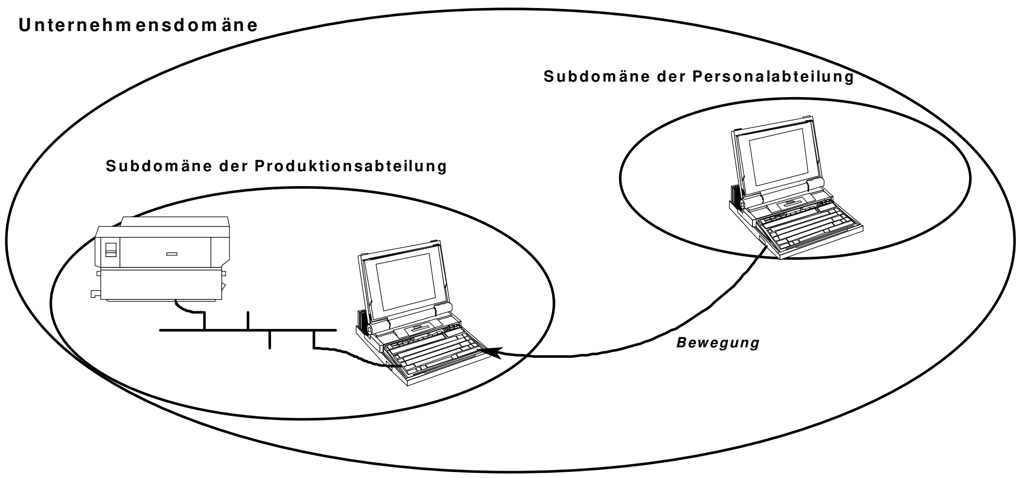

Abbildung 1-1: NoS-Beispielgraphik

Seit einiger Zeit beschäftigt sich ein Forschungsbereich des Managements intensiv mit der maschinenverständlichen Formulierung von vorhandenem Managementwissen und -vorgehen der Administratoren verteilter Systeme. Dieser Forschungsbereich wird als *Policy-basiertes Management* bezeichnet. Es wird dabei u.a. untersucht, wie dieses Expertenwissen in konkrete Regeln, den sog. *Policies*, umgewandelt werden kann. Wird das Management von NoS näher betrachtet, so kann festgestellt werden, daß auch hier zu Beginn gewisse Forderungen und Beschränkungen, also regelähnliche Formulierungen an die teilnehmenden Ressourcen und Netzkomponenten gestellt werden. Der Gedanke liegt in diesem Fall nahe, diese Forderungen und Beschränkungen ebenfalls durch Policies zu beschreiben, um diese gegebenenfalls später automatisch durchsetzen zu können. Die beim Policy-basiertem Management zu Tage geförderten Ergebnisse können z.T., v.a. aber in Hinblick auf die dabei entstandenen eindeutigen Formulierung von Policies durch eine gegebene Syntax und Semantik, auf den Bereich des Managements von NoS umgesetzt werden.

Das Ziel dieser Diplomarbeit ist es, eine *Managementarchitektur* (MA) zu entwerfen und prototypisch in einer Implementierung umzusetzen, die das Management nomadischer Systeme durch die Synthese mit Policy-basiertem Management verteilter Systeme ermöglicht. Ebenso sollen bereits bestehende Ansätze und Standards, die die Mobilität von Endgeräten unterstützend ermöglichen (wie z.B. MobileIP [RFC2002] und DHCP [RFC2131]), betrachtet und auf ihre Integration in die zu entwickelnde MA hin untersucht werden.

1.1 Nomadische Systeme

Nomadische Systeme sind, wie schon weiter oben definiert, sich im Rahmen eines Unternehmensnetzes *bewegende Objekte*. Als sich bewegende Objekte werden angesehen:

Mobile Computer
: Hierzu zählen v.a. die tragbaren Varianten, wie z.B. Laptops, Notepads und Personal Digital Assistants (PDAs). Zukünftig wäre es aber auch denkbar Desktop PCs miteinzubeziehen, die z.B. innerhalb eines Unternehmens mit einem Mitarbeiter von einer Abteilung in die nächste umziehen. Die Zugriffsrechte bzw. -beschränkungen an Ressourcen innerhalb der Unternehmens- bzw. Netzdomänen, werden jeweils für jeden Rechner einzeln oder für Rechnergruppierungen festgelegt.

Personen
: In diesem Fall werden i.d.R. den Personen Rollen, wie z.B. „Mitarbeiter“ oder „Abteilungsleiter“ zugewiesen. Jede Rolle wird wiederum mit gewissen Privilegien und Einschränkungen bezüglich vorhandener Netzressourcen assoziiert.

Subnetze
: Mobile Subnetze sind vernetzte Rechnergruppierungen, die insgesamt als Einheit zu sehen sind. Das bedeutet, daß diese Subnetze immer komplett von einem Standort zum nächsten wechseln. Ein Einsatz dieser Art von Subnetzen wäre z.B. auf Messen denkbar.

Es liegt auf der Hand, daß je nach Managementziel und dem vorliegenden nomadischen Systemtyp, diesen jeweils mit den entsprechend aussichtsreichsten Mitteln begegnet werden sollte, so daß ein einziger Lösungsansatz für die Managementaufgabe zunächst als nicht möglich erscheint.

Es ist zum einen denkbar, nicht nur Personen Rollen zuzuordnen, sondern auch mobilen Computern. Mobile Computer, die z.B. über eine besondere Ausstattung verfügen, könnten andere, u.U. mehr Zugriffsrechte auf Ressourcen zugewiesen werden. Eine Unterscheidung zwischen den verschiedenen Ausstattungen von Computern ist vor allem dann sinnvoll, wenn es um die Nutzung von Software geht. Es liegt auf der Hand, daß die Zuweisung einer Softwarelizenz durch einen License-Server an NoS grundsätzlich unterbunden werden sollte, falls die Voraussetzungen der Software nicht erfüllt werden, sei es weil bestimmte Basissoftware fehlt oder ein relevantes Hardware-Bauteil nicht vorhanden ist. Diese Unterscheidung wäre dann durch die Zuordnung von Rollen realisierbar, indem gezielt Klassen eingeführt werden, in die jedes NoS eindeutig zugeteilt werden kann. Folglich könnten daraufhin Regeln derart aufgestellt werden, daß z.B. bestimmte Software nur von NoS ab der Klasse (≈ Rolle) 3 genutzt werden kann.

Weiterhin ist eine injektive Abbildung zwischen einer Person und mobilen Computern möglich. In diesem Fall würden Rechte, die mit der Person, z.B. in der Rolle des „Abteilungsleiters" identifiziert werden, den persönlichen mobilen Computern zugewiesen. Problematisch wird diese Art der Rechtezuweisung dann, wenn dadurch auch der Zugriff auf sicherheitssensible Daten gesteuert wird und nur eine Authentifizierung der mobilen Endgeräte und nicht deren gegenwärtige Nutzer stattfindet.

Im Laufe dieser Arbeit werden die verschiedenen Arten des Policy-basierten Managements genauer betrachtet und im Hinblick auf das Management von NoS diskutiert.

1.2 Aspekte und Ziele des Managements

Aspekte des Managements von NoS		
	Beispiele	**zu managende Infrastruktur**
Konfigurierung	IP-Adresse, DNS-Name	Konfigurationsserver, Boot-Dienste (DHCP-Server, PPP-Server)
Ressourcenrechtevergabe	Netzzugang, Drucker, WWW-Server, E-Mail-Dienst	Netzkomponenten (Switch, Router, etc.), Server-Dienste (lpd, httpd, sendmail)
Abrechnung	Anzahl der gedruckten Seite, Zeitspanne der Nutzung einer Ressource	Abrechungsdienste

Abbildung 1-2: Überblick der Managementaspekte

Das primäre Ziel, das mit dem Management von NoS verfolgt wird, ist zunächst die Mobilität dieser Objekte durch unterstützende Maßnahmen zu ermöglichen. Dies wird dadurch erreicht, daß dem NoS durch das Managementsystem (MS) gewisse *Grundkonfigurationsdaten*, wie z.B. IP-Adresse und DNS-Name, zur Verfügung gestellt werden, indem geeignete *Konfigurationsserver* und *Boot-Dienste* eingesetzt werden. Um von Managementseite die Möglichkeit zu haben, dynamisch auf neue Situationen reagieren zu können, müssen diese Server und Dienste eine Managementschnittstelle aufweisen. Im Rahmen einer Diplomarbeit [Rieg96] wurde am Lehrstuhl bereits das Management von DHCP-Servern betrachtet und eine entsprechende Managementschnittstelle entworfen, die in einem Fortgeschrittenenpraktikum [Demm98] implementiert wird. Das zu entwickelnde Managementsystem wird sich auf die Ergebnisse dieser beiden Arbeiten stützen.

Ein weiteres Ziel des NoS-Managements ist es, wie schon eingangs angesprochen, den *Ressourcenzugriff* der NoS in geeigneter Weise zu steuern und zu beeinflussen. Als *(Netz-) Ressourcen* werden (wie in [HeAb93]) im Rahmen dieser Arbeit angesehen:

Netzkomponenten
Es liegt auf der Hand, daß nicht jedem NoS der Zugang zum Netz im vollem Umfang ermöglicht und erlaubt werden soll. Dies wird nur denjenigen NoS vorbehalten, die gewisse Kriterien erfüllen, wie z.B. daß bereits eine Authentifizierung des Objekts stattgefunden hat. Um dies bewerkstelligen zu können, muß eine Managementschnittstelle des Netzzugangs (z.B. Switch oder Hub) und der übrigen Netzkomponenten (z.B. Router, Bridge, etc.) vorhanden sein.

Peripheriegeräte
Zu den Peripheriegeräten zählen alle möglichen Einlese- und Ausgabegeräte, wie z.B. Scanner, Plotter, Laserdrucker, etc. Auch in diesem Fall ist ein Management erst möglich, wenn diese Geräte eine Managementschnittstelle bereitstellen. Dies kann auch dadurch realisiert werden, wenn der entsprechende Dienst, der den Zugriff auf das Gerät steuert, wie z.B. eine Druckerwarteschlange (lpd), eine entsprechende Schnittstelle vorweisen kann.

Server-Dienste
Server-Dienste stellen den Netzteilnehmern eine gewisse Funktionalität zur Verfügung. Dazu zählen u.a. WWW-Server (httpd) und Mailserver (z.B. sendmail)[1]. Hier soll i.d.R. je nach Identität des NoS, die Funktionalität dieser Dienste eingeschränkt werden.

Software
Der Zugriff auf bestehende und zur Verfügung gestellte Software erfolgt über einen Software-License-Server. Durch Steuerung dieses Servers kann, je nach Identität des NoS, der Umfang der Nutzung der entsprechenden Software eingeschränkt werden.

Ein dritter und heutzutage immer wichtiger werdender Aspekt des Managements von nomadischen Systemen, ist das *Accounting* (Abrechnung). Es besteht von Management-Seite her der Wunsch, die Bereitstellung gewisser Ressourcen gegenüber den Ressourcennutzern in Rechnung zu stellen. Ein Accounting wird meist dann eingesetzt, wenn Ressourcen auch Netzteilnehmern angeboten werden, die nicht zum „engeren Kreis“ gehören. Denkbar wäre der Einsatz eines Accounting-Dienstes, wenn z.B. den Mitarbeitern einer fremden Filiale eines Unternehmens grundsätzlich der Zugang zum Netz und die Nutzung der Ressourcen erlaubt wird, jedoch dies mit der entsprechenden Kostenstelle der Heimatfiliale abgerechnet wird.

In Kapitel 2 dieser Arbeit, werden u.a. anhand eines Beispielszenarios die Aspekte des Managements nochmals motiviert. In Kapitel 4 schließlich, wird u.a. die Kommunikation zwischen den Ressourcen und dem Managementsystem veranschaulicht, sowie die Bedingungen an die beteiligten Komponenten formuliert.

---

[1] Konfigurationsserver, wie z.B. DHCP-Server, die aus Gründen der Übersichtlichkeit getrennt betrachtet wurden, fallen zweifellos auch in diese Rubrik und werden im weiteren Verlauf dieser Arbeit ebenfalls als *Ressource* angesehen.

1.3 Policies

Um das Management von NoS so zu verwirklichen wie im vorhergehenden Unterkapitel beschrieben, ergibt sich die Möglichkeit der Verwendung von Policies. Wie schon eingangs erklärt, soll im engeren Sinne mit der Formulierung von Policies die Konfigurierung von NoS sowie dessen Zugriff auf Ressourcen gesteuert werden. Im weiteren Sinne jedoch, soll das gewünschte Verhalten von NoS in einem gegebenen Netz beschrieben werden.

Grundsätzlich enthält eine *Policy die Beschreibung, wie auf ein eingetretenes Ereignis reagiert* werden soll. Das eintretende Ereignis ist hier i.d.R. der Versuch der Nutzung einer Ressource und die entsprechende Reaktion eine Folge von Managementaktionen. Gelingt es diese Policies in einer formalen Sprache zu beschreiben, liegt der automatischen Bearbeitung der Policies nichts mehr im Weg.

Beispiel für eine Policy:

> *Die Mitarbeiter des Lehrstuhls für Datenbanksysteme dürfen an unserem Lehrstuhl nur eine begrenzte Anzahl an Seiten pro Semester drucken.*

Wie schon an diesem einfachen Beispiel erkennbar ist, enthält eine umgangssprachlich formulierte Policy zwar die Quintessenz der verfolgten Strategie, aber auch einige Ungenauigkeiten, die in weiteren Schritten erst beseitigt werden müssen, bevor eine automatisierte Interpretation der Policies überhaupt erst möglich wird. Um bei diesem Beispiel zu bleiben, muß zunächst geklärt werden, wie viele Seiten gedruckt werden dürfen, welche Personen, bzw. NoS, zum Lehrstuhl für Datenbanksysteme gehören, auf welchen Druckern ausgedruckt werden darf, etc.

Ein weiteres Problem stellt die *Auflösung von Konflikten* zwischen Policies dar. Würde z.B. bereits eine Policy bestehen, die besagt, daß für alle fremden Lehrstühle das Drucken am Lehrstuhl Prof. Dr. Hegering verbieten würde, so würde diese in Widerspruch zu der oben formulierten Beispielpolicy stehen. An dieser Stelle setzt eine *Konfliktlösungsstrategie* ein, die hier z.B. lauten könnte, daß eine genauer spezifizierte Policy immer eine allgemeiner gehaltene Policy „bricht". Diese Lösungsstrategie kann natürlich nicht für jeden denkbaren Konflikt angewendet werden, so daß hier eine Analyse der möglich auftretenden Konflikte nötig ist, um diesen eine Strategie entgegensetzen zu können.

Das Kapitel 3 dieser Arbeit beleuchtet ausführlich die verschiedenen Aspekte des Policy-basierten Managements.

1.4 Aufbau der Arbeit

Nach den einführenden Worten dieses ersten Kapitels, wird zunächst im *zweiten Kapitel* der momentane Istzustand des Managements von NoS betrachtet. Es wird ebenfalls ein Beispielsszenario erarbeitet, um die Motivation dieser Diplomarbeit hervorzuheben. Dieses Szenario wird in den darauffolgenden Kapiteln immer wieder aufgegriffen, um neue Ergebnisse daran vorzuführen.

Im *dritten Kapitel* wird das Policy-basierte Management von verteilten Systemen vorgestellt. Weiterhin wird in diesem Kapitel erarbeitet, wie die vorgestellten Konzepte auf das Management von nomadischen Systemen angewendet werden können und welche Anpassungen vollzogen werden müssen.

Im *vierten Kapitel* wird die entwickelte Managementarchitektur (MA) präsentiert. Zunächst werden die Anforderungen an die MA formuliert, um im nächsten Schritt die auf CORBA basierende, tatsächlich umgesetzte Architektur vorzustellen. Es werden die einzelnen Bestandteile mit den zur Verfügung gestellten Schnittstellen dokumentiert. Zum Schluß wird die Interaktion mit bereits existierenden und noch zu entwickelnden Managementagenten dargestellt.

Im *fünften Kapitel* wird das Objektmodell mit Hilfe der *Object Modeling Technique (OMT)* veranschaulicht. Es werden alle im Prototypen entwickelten Klassen und Interfaces im Rahmen von Klassendiagrammen erläutert.

Im *sechsten Kapitel* werden Implementierungsgesichtspunkte betrachtet. Es wird der Entwicklungsprozeß der einzelnen CORBA-Objekte aufgezeigt, sowie die Implementierung der Policy Description Language (PDL) mit Hilfe des Einsatzes eines *Compiler-Compilers*. Abgerundet wird dieses Kapitel mit einem Erfahrungsbericht über die verwendeten Entwicklungstools.

Im *siebten Kapitel* wird ein abschließender Rückblick getätigt sowie ein Ausblick über nicht behandelte, bzw. nur angedeutete Themenbereiche, die einer weiteren Betrachtung würdig wären.

Jedem Kapitel wird zudem eine Einleitung vorangestellt, die einen Überblick des behandelten Themas beinhaltet. Ebenso enthält jedes Kapitel abschließend eine Zusammenfassung, die die Ergebnisse des Kapitels nochmals in aller Kürze festhält.

Kapitel 2

State of the Art

Dieses Kapitel legt seinen Schwerpunkt auf die Betrachtung der momentanen Unterstützung nomadischer Systeme bei deren Einbindung in Intranets. Zunächst wird ein Beispielszenario erarbeitet sowie ein erster Einblick in die Umsetzung der daraus resultierenden Managementaufgabe dargestellt, um die Motivation dieser Diplomarbeit nochmals zu unterstreichen. Dieses Szenario wird im Laufe der Diplomarbeit immer wieder aufgegriffen, um jeweils neue Erkenntnisse daran vorzuführen. Anschließend erfolgt eine Abgrenzung zu Themen- und Forschungsgebieten, die sich ebenfalls mit nomadischen (mobilen) Systemen beschäftigen. Zum Ende dieses Kapitels schließlich, wird eine Liste von Softwareprodukten sowie deren Beschreibungen vorgestellt, mit denen momentan der Nomadizität von mobilen Einheiten in Intranets begegnet wird.

2.1 Das Bibliotheksszenario

2.1.1 Szenariobeschreibung

Das folgende Szenario lehnt sich an das in [Heil98] entwickelte Beispielszenario an.

In einer Bibliothek wird Besuchern die Möglichkeit gegeben, ihre tragbaren Computer an dafür vorgesehene öffentliche Stellen an das dort etablierte Intranet anzuschließen. Den Besuchern soll dadurch die Gelegenheit geboten werden, auch durch Nutzung des *World Wide Web (WWW)* im Internet nach Büchern zu suchen sowie Daten untereinander auszutauschen. Die User bekannter NoS, denen zuvor ein Account eingerichtet wurde, wird auch die Nutzung des lokalen Druckers und ein erweiterter Internetzugang über das WWW erlaubt, der ebenfalls ein Senden von Emails ermöglicht. Für die Nutzung des Druckers muß aber im voraus zusätzlich ein variabler Seitenbestand eingekauft werden. Ist der eingekaufte Bestand aufgebraucht, wird jeglicher weitere Druckversuch abgelehnt. Neben den öffentlichen existieren auch nichtöffentliche Anschlußmöglichkeiten, die ausschließlich für Bibliotheksmitarbeiter reserviert sind. Da zwischen den Mitarbeitern auch sicherheitssensible Daten ausgetauscht werden, muß eine absolute Trennung zwischen dem öffentlichen und nichtöffentlichen Bereich erreicht werden. Ebenso befindet sich im nichtöffentlichen Bereich ein Drucker, den alle Mitarbeiter ohne Einschränkung benützen dürfen. Auch Mitarbeiter fremder Bibliotheken, die ebenfalls diesen Service anbieten, dürfen ihre mobilen Einheiten im nichtöffentlichen Bereich anschließen und können von dieser Stelle aus auch auf ihr Heimatnetz zugreifen. Die Druckernutzung ist jedoch nur möglich, falls ein Abrechnungsdienst eingerichtet ist und dies vorher mit der Heimatfiliale vereinbart wurde. Unbekannte Systeme haben hingegen im nichtöffentlichen Bereich keine Möglichkeiten Netzressourcen zu nutzen.

2.1.2 Lösungsansatz der Managementaufgabe

Die Szenariobeschreibung trennt den zu managenden Bereich in mehrere *Domänen*. Eine *Domäne* ist ein Bereich innerhalb eines zu managenden Netzes, der zu managende Ressourcen enthält. Die Art der Aufteilung eines Netzes in verschiedene Domänen ist abhängig von der Sicht des Betrachters auf den zu managenden Bereich. In Abbildung 2-1 ist die Domänenaufteilung aus Sicht der im vorhergehenden Abschnitt dargestellten Beschreibung des Szenarios veranschaulicht und besteht aus der Bibliotheksdomäne, die wiederum in einen öffentlichen und einen nichtöffentlichen Bereich gegliedert ist. Neben der Bibliotheksdomäne existieren weitere Filialdomänen anderer Bibliotheken.

Es kann zwischen drei verschiedenen nomadischen Systemtypen unterschieden werden:

- *unbekannte Systeme*: Das sind tragbare Computer von Besuchern, die keinen eingerichteten Account haben und denen somit die Nutzung des Druckers und der erweiterte Internetzugang verwehrt bleibt.
- *authentifizierte Systeme der Bibliotheksbesucher*: In diesem Fall hat der Besucher einen eingerichteten Account für seinen tragbaren Rechner und darf grundsätzlich den lokalen Drucker und den erweiterten Internetzugang nutzen.
- *authentifizierte Systeme der Mitarbeiter*: Dies sind die Endgeräte der Mitarbeiter, die sich vor der Nutzung von Ressourcen im nichtöffentlichen Bereich, einer Authentifizierungsprozedur unterwerfen müssen.

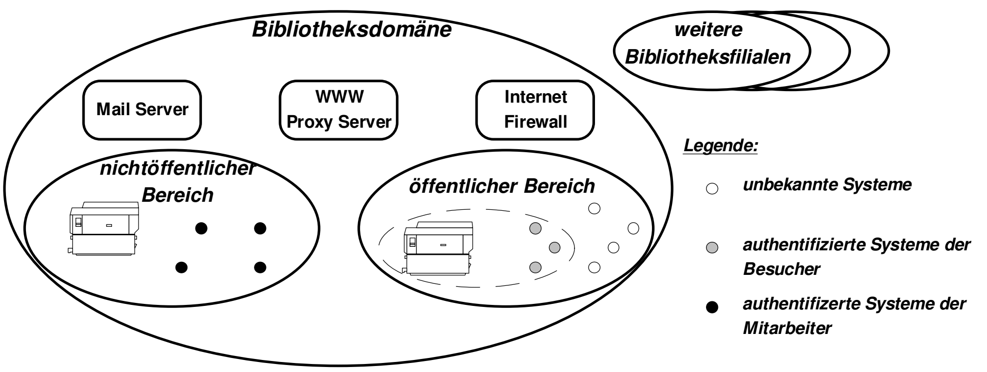

Abbildung 2-1: Bibliotheksszenario

Weiterhin lassen sich aus der Szenariobeschreibung folgende zu managenden Ressourcen identifizieren:

Switch
: Ein Switch dient als Anschlußstelle für die nomadischen Systeme und bietet u.a. die Möglichkeit, ggf. ein System ganz von der Kommunikation des Netzes auszuschließen.

Mailserver
: Der Mailserver leitet gesendete Emails an die Adressaten weiter.

WWW Proxy Server
: Der Proxy Server hält in seinem Cache die WWW-Seiten derjenigen Server, die die Suche nach Büchern realisieren. Die Besucher, die keinen Account im öffentlichen Bereich haben, können auf keine anderen Seiten zugreifen als die, die von diesem Proxy gehalten werden, während dies den Besuchern mit Account sowie den Mitarbeitern ermöglicht wird.

DHCP Server
: Da die nomadischen Systeme vor der Nutzung des Internets und der dazugehörigen Internetdienste zumindest eine IP-Adresse benötigen, wird der *DHCP Dienst* [RFC2131] für diese Aufgabe eingesetzt. Der DHCP Server weist bei einem Request eines DHCP Clients diesem dynamisch sowohl eine IP Adresse als auch einen *DNS-Namen* zu. Voraussetzung für alle nomadischen System ist es demzufolge, daß eine entsprechende Client-Software installiert ist. Die Managementschnittstelle des Servers ermöglicht die Zuweisung einer bestimmten IP Adresse für ausgewählte Systeme sowie das Ablehnen einer Adressenzuteilung.

Drucker
: Alle Druckaufträge werden, vor deren Weiterleitung zum Drucker, in eine Warteschlange eingefügt. Dieser Dienst ist um eine entsprechende Managementschnittstelle erweitert, die das Unterbinden der Weiterleitung an den Drucker ermöglicht.

Internet Firewall
: Die Firewall wird durch einen *Router* realisiert, der eine Verbindung außerhalb der Bibliotheksdomäne nur in dem Fall zuläßt, wenn der Verbindungsaufbau von einem authentifiziertem System aus dem nichtöffentlichem Bereich initiiert wurde.

Der nächste Schritt zur Lösung der Aufgabenstellung, ist die Umsetzung der in Abbildung 2-1 skizzierten Domänenaufteilung. In der Szenariobeschreibung wurde eine strenge Trennung zwischen öffentlichem und nicht öffentlichem Bereich gefordert, so daß eine Separierung des Datenverkehrs auf einer möglichst unteren OSI-Schichtebene das verfolgte Ziel sein muß. Dies ist mit einer Zusammenschaltung von mehreren Layer-2-Switches, wie es in Abbildung 2-2 veranschaulicht ist, auch durchsetzbar. Bei der bei einem *Switch* eingesetzten Technologie handelt es sich um die des Leitungsschaltens[2]. Ein Switch besteht aus mehreren *Ports* an denen Endgeräte angeschlossen werden können. Bei einem Verbindungsaufbau wird zwischen den beteiligten Ports meist temporär physisch eine Verknüpfung hergestellt. Durch eine entsprechende Erweiterung um eine Managementschnittstelle ist es möglich, *Virtuelle LANs (VLAN)* aufzubauen, indem bewußt bestimmte Ports miteinander verknüpft werden und von den übrigen getrennt werden. So ist es möglich, daß generell eine Trennung zwischen den

[2] Für den interessierten Leser seien die Bücher [Hals96] und [Kern95] empfohlen, die ausführlich die Switch-Technik beschreiben.

Switches des öffentlichen Bereichs mit denen des nicht öffentlichen Bereichs etabliert werden kann, wenn diese Switches an einer zentralen Stelle zusammengeführt werden, die eine Verbindung zum restlichen Netz zur Verfügung stellt. Weiterhin kann die Nutzung des Druckers auch dadurch gelöst werden, indem bekannte NoS demselben VLAN, in dem sich der Drucker befindet, hinzugefügt werden. Die unbekannten NoS werden dagegen in einem separierten VLAN zusammengeführt, von dem aus ein Zugriff auf den Drucker nicht möglich ist. Ebenso ist es denkbar, durch entsprechende Managementanweisungen einen Anschlußport abzuschalten, falls ein Switch aus dem nichtöffentlichen Bereich von dem Anschluß eines nicht authentifizierten Systems erfährt.

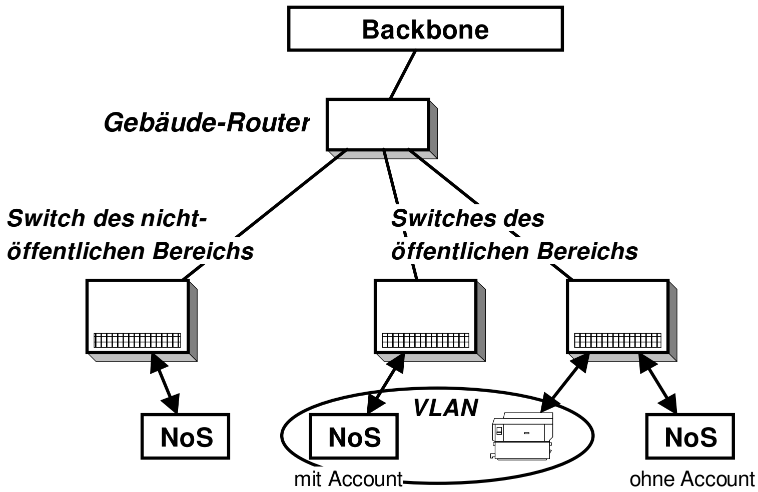

Abbildung 2-2: Switch-Verbindung im Bibliotheksszenario

Wird also davon ausgegangen, daß jede Ressource mit einer Managementschnittstelle ausgestattet ist, so erscheint die Managementaufgabe zunächst als durchaus lösbar. Probleme bereiten aber einerseits die Authentifizierung der verschiedenen Systeme sowie andererseits, die aufgestellten Regeln und Bedingungen, zu welchem Zeitpunkt eine bestimmte Handlung von einer Ressource durchgeführt werden soll. Die Authentifizierung ist besonders dann zwiespältig, wenn dies durch das Managementsystem einer „fremden“ Domäne durchgeführt werden soll, wie es im Beispielszenario bei Anschluß eines NoS einer fremden Bibliotheksfiliale im nichtöffentlichen Bereich der Fall sein würde. In diesem Fall muß zusätzlich eine gegenseitige Authentifizierung der MS vollzogen werden. Die aufgestellten Regeln hingegen, müssen in geeigneter Weise formuliert werden, um daraufhin von dem Managementsystem interpretiert und durchgesetzt zu werden.

Aus der obigen Beschreibung des Switch-Managements sind schon regelähnliche Formulierungen der Art „Wenn ein nicht authentifiziertes System im nichtöffentlichen Bereich angeschlossen wird, schalte den Port ab“ erkennbar. Von diesem Punkt aus ist es kein weiter Weg mehr, eine maschineninterpretierbare *Policy* zu formulieren. Diese einzelnen Schritte werden ausführlich in Kapitel 3 dieser Arbeit besprochen, so daß auf deren weitere Darstellung an dieser Stelle verzichtet wird.

Es zeigt sich also beim Policy-basiertem Management nomadischer Systeme folgende Vorgehensweise, um die Aufgabe erfolgreich zu bewältigen:

- Analyse der Managementaufgabe und Festsetzung der Managementziele
- Identifizierung der daran beteiligten Komponenten und insbesondere der zu managenden Netzressourcen
- Spezifizierung der Domänenstrukturierung sowie deren Umsetzung
- Formulierung der Policies, die sich meist aus regelähnlichen Aussagen der Ziele ergeben

In dieser Arbeit soll demzufolge ein Managementsystem (MS) entwickelt werden, das diese Art der Managementverfahrensweise verwirklicht.

2.2 Derzeitiges Management von nomadischen Systemen

Die Industrie engagiert sich bei nomadischen Systemen heutzutage v.a. in Hinblick auf deren Einbindung in bestehende LANs über Funkverbindungen (*wireless connections*) und per Modem über das Post/Telefonnetz. Hier kommt fast ausschließlich das Protokoll PPP (*Point-to-Point Protocol)* [RFC1548] für die Verbindung zu einem *Internet Service Provider (ISP)* zum Einsatz, der daraufhin, u.a. durch Vergabe einer IP-Adresse, den Zugang zum Internet schafft und damit die Grundlage für die Kommunikation mit dem gewünschten LAN etabliert. Oft wird jedoch, u.a. aus Gründen der Sicherheit, eine ausschließliche Kommunikationsverbindung zum LAN verlangt. In diesem Fall wird meist das Protokoll PPTP (*Point-to-Point Tunneling Protocol*) [PTHV96] verwendet, bei dem zwischen dem ISP und dem LAN ein „Tunnel" für die IP-Datagrammweiterleitung aufgebaut wird. Für den Anwender erscheint dies alles transparent und er meldet sich wie gewohnt nach der Verbindungsaufbauprozedur in seinem LAN an, als ob dieser sich direkt vor Ort befinden würde. Um die Authentifizierung und die Vergabe einer IP-Adresse muß sich in diesem Fall aber das Netzwerk des Service-Nutzers kümmern. Oft wird hierfür der *RAS-Dienst* von *Microsoft* eingesetzt, der diese Funktionalität anbietet. Diese Art der NoS-Einbeziehung in Intranets liegt jedoch, wie schon aus dem Beispielszenario geschlossen werden kann, außerhalb des Fokus dieser Arbeit. Vielmehr liegt der Ansatz dieser Arbeit in der Betrachtung der Mobilität von Systemen *innerhalb* von Intranets und nicht deren Einbeziehung bei Kommunikationsverbindungen, die außerhalb des bestehenden Intranets erst etabliert werden müssen.

Das Management von nomadischen Systemen in Intranets, d.h. basierend auf dem Protokollpaar TCP/IP, beschränkt sich gegenwärtig noch auf sehr grundlegende Elemente des Konfigurationsmanagements wie der dynamischen Zuweisung von IP-Adressen mittels des *Dynamic Host Configuration Protocol (DHCP)* [RFC2131], der dynamischen Zuteilung eines DNS Namen mittels *Dynamic DNS* und der automatischen IP Datagrammweiterleitung mittels *MobileIP* [RFC2002]. Eine standardisierte Managementschnittstelle ist bei diesen Spezifikationen nicht vorgesehen, so daß die Zusammenarbeit mit einem bereits vorhandenen MS nicht möglich ist. Das tatsächliche Management drückt sich darin aus, daß Parameter in den dazugehörenden Konfigurationsdateien der Implementierungen durch den Systemadministrator entsprechend gesetzt werden. Es liegt auf der Hand, daß dies nur sehr unbefriedigend aus Sicht des Netz- und Systemmanagers ist. Trotzdem wird nachstehend aus Gründen der Vollständigkeit, in aller Kürze die oben genannten Spezifikationen beschrieben, um daraufhin Produkte vorzustellen, die, basierend auf diesen Standards, nomadische Systeme unterstützen.

2.2.1 Dynamic Host Configuration Protocol (DHCP)

In RFC 2131 wird das *Dynamic Host Configuration Protocol (DHCP)* spezifiziert, das grundsätzlich eine automatische Konfiguration von Endgeräten (DHCP Clients) durch einen DHCP Server vorsieht. DHCP ist demzufolge nicht speziell für NoS konzipiert worden, sondern allgemein für Systeme in einem heterogenen Netzumfeld, mit dem Ziel deren Konfigurierung zu erleichtern. Neben der Festlegung eines Kommunikationsprotokolls, ist die Unterstützung der folgenden Punkte vorgesehen:

- Zuweisung einer IP-Adresse
- Weitergabe relevanter Netzwerkinformationen wie Subnet Maske, nächsten Router, sowie individuellen Parametern
- optionale Zuordnung eines DNS Namen mittels *Dynamic DNS*
- optionale Unterstützung von *MobileIP*

Es werden drei verschiedene Arten der Adressenzuteilung in der Spezifikation festgelegt:

Manuelle Zuweisung
: Die Abbildung zwischen IP Adresse und Endgerät wird von dem Systemadministrator festgesetzt und in einer Datei „manuell" eingetragen. Dies heißt demzufolge insbesondere, daß ein Client immer dieselbe IP Adresse von dem DHCP Server zugewiesen bekommt.

Automatische Zuweisung
: Der Systemadministrator räumt einem DHCP Server einen Adressenpool ein, aus dem dieser automatisch einem Client eine IP Adresse zuteilen kann. Der Systemadministrator muß in diesem Fall darauf Acht geben, daß die Pools verschiedener Server disjunkt sind. Die dem Client einmal zugewiesene Adresse bleibt diesem bis zu seiner Abmeldung erhalten.

Dynamische Zuweisung
: In diesem Spezialfall der *automatischen* Zuweisung, belegt der Server die dem Client zugeteilte IP Adresse mit einer zeitlich begrenzten Gültigkeit (*lease*). Ist die IP Adresse nicht mehr gültig, so muß sich der Client um eine Verlängerung oder um die Zuteilung einer neuen IP Adresse bemühen.

Für die Unterstützung nomadischer Systeme ist nur die automatische und dynamische Zuweisung interessant, da einerseits auch für unbekannte Systeme die Konfigurierung durch den DHCP Server ermöglicht werden soll, welches bei der manuellen Zuweisung ausgeschlossen ist, und andererseits dem Systemadministrator die Aktualisierungsarbeit der Adressenabbildungen erspart werden kann.

Weit aus üblicher im Intra- und Internet ist aber das Ansprechen eines Systems über seinen *Fully Qualified Domain Name (FQDN)*, anstatt über seine IP Adresse. Das *Domain Name System (DNS)* bietet basierend auf einem verteilten Namensverzeichnisbaums die Funktionalität an, FQDNs auf IP Adressen (und umgekehrt) abzubilden. Wünschenswert wäre es also auch, die Einträge des zuständigen DNS Servers mit dem FQDN des Clients zu aktualisieren.

Diese Möglichkeit bietet DHCP mit dem *Dynamic DNS* Verfahren an, in der die Einträge in den entsprechenden Dateien entweder durch den Server oder den Client aktualisiert werden können.

Auf weiteres Eingehen in die Fülle der verschiedenen DHCP Optionen und das Kommunikationsprotokoll zwischen Client und Server, wird an dieser Stelle verzichtet und auf die Spezifikation [RFC2131] verwiesen. Im Rahmen von Fortgeschrittenenpraktika wurden am Lehrstuhl ein Java/CORBA-Managementagent für DHCP-Server [Demm98] sowie ein Java-basierter Testclient für DHCP-Server [May98] entwickelt. Das auf DHCP basierende Management von nomadischen Systemen in Intranets wurde ebenfalls in [Heil97] und [Rieg96] untersucht.

2.2.2 Mobile IP

Im vorhergehenden Kapitel wurde u.a. betrachtet, wie einem NoS automatisch mittels DHCP eine neue IP Adresse zugeteilt werden kann, wenn dieses innerhalb eines Subnetzes neu angeschlossen wird. Damit ist primär die Voraussetzung für eine Kommunikation mit anderen Netzkomponenten erfüllt. Oft liegt aber folgendes Szenario vor:

> *Die mobilen Einheiten sind fest in einem Subnetz integriert und dienen den Usern als Arbeitsplatzrechner. Temporär werden diese Einheiten aus diesem Netz entfernt und in einem anderen Subnetz, z.B. über eine Funkverbindung, angeschlossen. Es wird gewünscht, daß trotzdem die mobilen Einheiten weiterhin so agieren können, als ob diese sich noch immer in ihrem Heimatsubnetz befinden.*

Es liegt auf der Hand, daß dies mit der Zuteilung einer neuen IP Adresse nicht ohne weiteres bewerkstelligt werden kann. Ein Grund dafür wäre, daß die Systeme innerhalb von LANs meist in der Art konfiguriert sind, daß nur die in diesem LAN ansässigen Systeme auf die vorhandenen Ressourcen zugreifen können. Grundsätzlich wäre dieses Szenario aber realisierbar, wenn die mobile Einheit die ihm einmal zugewiesene IP-Adresse behält, völlig unabhängig von der Tatsache, in welchem Subnetz dieser wahrhaftig angeschlossen wird. Dann aber müßten, bei Abwesenheit des entsprechenden Systems im Heimat-LAN, alle an diese Einheit adressierten IP-Datagrammpakete an dessen neuen Standort weitergeleitet werden. Für die umgekehrte Richtung gilt selbstverständlich dasselbe, so daß der durchgeführte Ortswechsel für alle übrigen Systeme völlig transparent vollzogen wird. Dieser Ansatz wird mit *MobileIP* verfolgt und wurde in [RFC2002] spezifiziert. Darin werden die folgenden Begriffe definiert:

Mobile Node
: Darunter werden Systeme wie Laptops, Router, etc. gezählt, die jeweils die Möglichkeit haben, in verschiedenen Netzen angeschlossen zu werden. Ein mobiler Knoten kann, ohne die Notwendigkeit seine IP Adresse zu ändern, seinen Standort wechseln. Jedem mobilen Knoten wird eine Langzeit-IP-Adresse im Heimatnetz zugewiesen. Entfernt sich dieser aus dem Heimatnetz und wird in einem fremden Netz angeschlossen, so wird der mobile Knoten mit einer *Care-of-address* assoziiert. Der mobile Knoten quittiert weiterhin alle die von ihm gesendeten Nachrichten mit seiner Heimat-IP-Adresse. Um die Tunnelung der Datagramme kümmern sich jeweils der *Home* und der *Foreign Agent*.

Home Agent
: Dies ist ein Router im Heimatnetz des mobilen Knoten, der Datagramme an diesen weiterleitet, falls der Heimatstandort verlassen wurde. Um diese Aufgabe zu bewäl-

tigen, verwaltet der Home Agent unterdessen auch die nötige Information über den derzeitigen Anschlußpunkt des mobilen Knoten.

Foreign Agent
Der Foreign Agent ist der eigentliche Empfänger der vom *Home Agent* gesendeten Datagramme an einen mobilen Knoten, falls sich dieser zeitweise in dem vom Foreign Agent verwalteten Netz befindet. Der mobile Knoten registriert sich zunächst bei dem Foreign Agent eines besuchten Netzes, der anschließend in Verbindung mit dessen Home Agent tritt, um die Datagramm-Tunnelung verwirklichen zu können.

In Zusammenarbeit mit *DHCP* ist es durch die DHCP-Option *Mobile IP Home Agent* möglich, DHCP Clients die Adressen mehrerer Home Agents mitzuteilen. MobileIP wird oft für die *wireless connection* (Funk- und Infrarotverbindung) von mobilen Einheit eingesetzt.

2.2.3 Produktliste

Basierend auf den beiden in den vorhergehenden Kapiteln vorgestellten Konzepten, sind wegen ihres standardisierten Status zahlreiche Produkte für nahezu alle Betriebsysteme erschienen. Im folgenden werden einige, mittels einer Kurzbeschreibung, vorgestellt:

Network Registrar (für SUN Solaris, Windows NT)
Diese Software von der *American Internet Corporation* ([AIC97a], [AIC97b]) beinhaltet sowohl einen DHCP Server als auch einen DNS Server und ist allgemein für das IP Management in Unternehmen ausgelegt. Parametereinstellungen für die beiden Server können jeweils nur durch ausgewählte Administratoren durchgeführt werden, die sich zuvor einer Authentifizierungsprozedur unterzogen haben. Die Software bietet für diese Tätigkeit eine übersichtliche graphische Benutzeroberfläche (GUI) an. Als besonderes Merkmal ist die Möglichkeit mittels Logdateien die durchgeführten Schritte der beiden Server zu protokollieren, um diese für eine spätere Analyse heranziehen zu können. Zudem ist es möglich, sich jederzeit den augenblicklichen Status und Zustand der Server anzeigen zu lassen. Leider fehlt auch hier eine standardisierte Managementschnittstelle, wie z.B. ein SNMP-Interface, um diese Statusinformationen durch ein zentrales Managementsystem abfragen und integrieren zu können. So muß diese relevante Managementinformation durch den Administrator immer separat betrachtet werden.

Meta IP/Manager
Der *Meta IP/Manager* von *Metainfo* [Meta98] bietet dieselbe Funktionalität an wie die Software *Network Registrar.* Auch hier sind sowohl ein DHCP- als auch ein DNS-Server im Paket enthalten. Der einzige Unterschied liegt in der graphischen Benutzeroberfläche, die in diesem Fall Java-basierend realisiert worden ist. Dadurch ergibt sich der entscheidende Vorteil der System- und Ortsunabhängigkeit der Java-Oberfläche der Server, die von jedem Punkt des zu administrierenden Netzes aus angesprochen und verwaltet werden können. Auch hier fehlt wieder eine standardisierte Managementschnittstelle.

Windows NT 4.0 Server
Im *Microsoft Windows NT 4.0 Server* Betriebssystem sind standardmäßig ein DHCP- und ein DNS-Server enthalten, die durch eine Windows-gewohnte Oberfläche administriert werden können. Wie bei den beiden vorhergehend beschriebenen

Produkten, ist auch hier keine standardisierte Managementschnittstelle eingeschlossen. Dafür beinhaltet die *Windows NT 4.0 Client* Version des Betriebsystems einen DHCP-Client. In Tests [McCl96] wurde jedoch festgestellt, daß der DHCP-Server in einem heterogenen Netzumfeld nur unzureichende Ergebnisse liefert.

Telxon MobileIP Solution
Bei dieser Software [Telx95] handelt es sich um eine Implementierung von *MobileIP* für DOS/Windows-Systeme. Die Zielsetzung, die hierbei verfolgt wird, ist eine Grundlage für *virtuelle Büros* zu schaffen. Es ist sowohl für die drahtlose Verbindung als auch für Verbindungen, die im Rahmen eines Netzwerks stattfinden, gedacht.

Linux MobileIP
Für das Betriebssystem *Linux* sind viele Implementierungen von *MobileIP* erhältlich. Eine Liste der verschiedenen Softwareprodukte ist auf der MobileIP-Homepage [GDL96] zu finden. Die Produkte sind meist frei beziehbar, da diese von wissenschaftlichen Einrichtungen verschiedener Universitäten zu Testzwecken implementiert worden sind.

Für Policy-basiertes Management ist ebenfalls vor kurzem folgendes Produkt auf den Markt gekommen:

Orchestream Service Provisioning System (SPS)
Diese Software von *Orchestream* [Orch98], die sowohl für Windows- als auch Unix-Systeme erhältlich ist, konzentriert sich auf die Festlegung von Policies bezüglich *Quality of Service (QoS)*-Parametern im WAN-Bereich. Sie ist v.a. für den Einsatz in denjenigen Netzwerken ausgelegt, deren Nutzung von Service Providern angeboten wird, um diesem zu ermöglichen, seinen Kunden eine bestimmte Bandweite zu festgelegten Zeiten anbieten zu können. Die Managementarchitektur sieht eine zentrale Stelle für die Spezifizierung der Policies vor, die, nach deren Aktivierung, auf *Policy Server* verteilt werden, die sich (lokal) um deren Durchsetzung kümmern. Diese Policy Server agieren auf den *Managementobjekten (MO)*, bei denen es sich ausschließlich um WAN-Endgeräte wie Router handelt, über Proxy-Agenten, die jeweils die herstellerspezifischen Managementprotokolle der MOs unterstützen. Wie aus dieser Kurzbeschreibung erkennbar ist, eignet sich dieses Produkt, aufgrund dessen Zielsetzung, nicht für das Management nomadischer Systeme.

2.3 Zusammenfassung

Anhand des Bibliotheksszenarios wurden sowohl die verfolgten Managementziele als auch erstmals die Anforderungen an das zu entwickelnde Managementsystem skizziert. Diesem Szenario und der daraus resultierenden Wunschvorstellung für das Managementsystem wurde das gegenwärtige NoS-Management basierend auf DHCP und MobileIP gegenübergestellt. Aus Sicht des integrierten Netz- und Systemmanagements wurde dies als unzureichend bewertet, da einerseits eine standardisierte Managementschnittstelle in allen vorgestellten Softwareprodukten fehlte und andererseits die Beschränkung auf die reine IP Adressenverwaltung bzw. Datagrammweiterleitung nur eine Facette der formulierten Managementziele ist. Es wurde aber auch in dem vorgestellten Beispielszenario deutlich, daß diese standardisierten Konzepte durchaus in das noch zu entwickelnde MS integriert werden können und sollten, da die Tatsache, daß zahlreiche Implementierungen dieser Spezifikationen für alle denkbaren Betriebssysteme existieren, v.a. in einem heterogenen Netzumfeld nicht zu vernachlässigen ist. Voraussetzung hierfür bleibt aber, daß die entsprechenden Implementierungen eine Managementschnittstelle vorweisen können. Für DHCP Server wurde bereits im Rahmen einer Diplomarbeit [Rieg96] eine SNMP-Schnittstelle und die dazugehörige MIB entwickelt.

Kapitel 3

Policy-Management

In diesem Kapitel werden die bisherigen Ergebnisse des Policy-Managements verteilter Systeme betrachtet und auf ihre Anwendbarkeit für das Management nomadischer Systeme hin untersucht. Es wird im folgenden aufgrund der Vielzahl an Veröffentlichungen auf diesem Gebiet darauf verzichtet, alle Konzepte miteinander zu vergleichen. Vielmehr konzentriert sich dieses Kapitel auf die Ergebnisse derjenigen Arbeiten, die für das NoS-Management am vielversprechendsten sind. Basierend auf diesen Erkenntnissen, werden die unternommenen Weiterentwicklungen beschrieben, die sich v.a. in einer Anpassung der *Policy Description Language (PDL)* äußert. Eine ausführliche Gegenüberstellung der verschiedenen Policy-Ansätze und deren Realisierung bietet D.A. Marriot in seiner Dissertation [Mari97], so daß der interessierte Leser auf diese verwiesen wird. Weiterhin wird ein Überblick über Konfliktlösungsstrategien dargeboten sowie, zum Ende dieses Kapitels, typische Policies hinsichtlich nomadischer Systeme vorgestellt.

3.1 Einführung

Im folgenden werden zunächst die Begriffe, die in Zusammenhang mit Policies in der Literatur ([Mari97], [Koch96], [MMS94], [Wies95b]) verwendet werden und für das NoS-Management relevant sind, zunächst definiert und deren Bedeutung erklärt. Bei der Betrachtung der PDL für operationale Policies in Abschnitt 3.5, werden einem Teil dieser Begriffe neue bzw. dem NoS-Management angepaßte Definitionen zugewiesen:

(Management) Policy

Sowohl Koch (in [Koch96]) als auch Wies (in [Wies95b]) liefern folgende (nicht wörtlich übernommene) Definition für *Management Policies*:

Policies beschreiben das gewünschte Verhalten von Systemen und Applikationen in einem heterogenen Netzumfeld.

Wies betrachtet weiterhin Policies als die Verbindung zwischen dem Unternehmens- und Technologiemanagement. Eine tatsächliche Umsetzung dieser so formulierten, abstrakten *High-level-Policies*, wird durch weitere Verfeinerungen erreicht. Ab einer bestimmten Abstraktionsebene der Policies, können diese als aufgestellte Regeln betrachtet werden, die Anweisungen enthalten, wie auf ein bestimmtes, eingetretenes

Ereignis reagiert werden soll. Die durch Verfeinerung erhaltenen *Low-level-Policies* beinhalten i.d.R. eine Menge von Objekten (*Subjekte*), die auf einer Menge von Managementobjekten (*Zielobjekte, Targets*) bestimmte Aktionen ausführen dürfen (*authorization*) oder müssen (*obligation*).

Subjekte
Subjekte sind eine Menge von Objekten, die die innerhalb einer Policy bestimmten Operationen ausführen.

Zielobjekte (Targets)
Dies stellt diejenige Menge von Objekten dar, die von den ausgeführten Aktionen einer Policy betroffen sind.

Modalität
Eine Policy kann bezüglich ihres Modus als *Autorisierungs-* oder *Verpflichtungspolicy* agieren. Eine Autorisierungspolicy berechtigt bzw. verbietet das Ausführen der formulierten Aktionen, während die Verpflichtungspolicy die Subjekte zur Ausführung der spezifizierten Aktionen verpflichtet.

Aktionen
Als Policy-Aktionen werden die auszuführenden Operationen bezeichnet. Je nach Verfeinerungsstufe der Policy kann es sich hierbei um ausdrückliche Managementoperationen handeln oder, wie bei High-level-Policies üblich, um die in Prosa formulierten, verfolgten Ziele der Policy.

Vorbedingungen (Preconditions)
Dies ist die Prämisse, unter welcher die formulierten Aktionen ausgeführt werden dürfen. Oft wird als Vorbedingung für die Feuerung einer Managementoperation das erfolgreiche Abschließen der vorhergehenden Operationen gesetzt.

Constraints
Constraints beschreiben allgemein die Bedingungen für die Anwendbarkeit einer Policy.

Wie eingangs schon darauf hingewiesen wurde, zeigen die meisten Veröffentlichungen auf diesem Gebiet eine der oben dargestellten, ähnlichen Struktur der Policy-Beschreibung. Neben dieser Struktur, existieren auch andere Ansätze Policies zu spezifizieren und zu definieren, die für das allgemeine Management von verteilten Systemen u.U. besser geeignet sind. Ein vielversprechender Ansatz wurde in [Goh98] entwickelt, in dem Policies als eine Menge von *Constraints* (Einschränkungen und Beschränkungen) angesehen werden, die ein Objekt erfüllen muß. Um das Erfüllen dieser Constraints überprüfen zu können, wird jedem Managementobjekt eine Menge von Zuständen zugewiesen. Verläßt ein Objekt seinen augenblicklichen Zustand und verletzt in seinem neuen Zustand einen Constraint, so werden die nötigen Operationen durchgeführt, um das Objekt in einen Zustand überzuführen, der die aufgestellten Constraints nicht verletzt. Wie sich während des Studiums der Literatur und den ersten Skizzen des Prototypen zeigte, erschien dieser, neben anderen hier unerwähnt gebliebenen Ansätzen, als zu mächtig für das Management nomadischer Systeme. Es wurde bewußt die obige Struktur gewählt, da dessen Umsetzung in konkrete Implementierungen, wie z.B. in [Koch96] und [Mari97], schon gelungen war und deshalb die Fertigstellung eines Prototypen für die Policy Interpretation am vielversprechendsten wirkte. Dennoch flossen einige Ideen dieser Ansätze v.a. in die Definition der *Policy Description Language*.

Oben wurde ebenfalls schon eine Klassifizierung von Policies durch eine Unterscheidung zwischen *Autorisierungspolicies* und *Verpflichtungspolicies* angedeutet. Beide lassen sich wiederum in eine *positive* und *negative* Auslegung aufteilen. Die *positive Autorisierungspolicy* erlaubt den Subjekten die Ausführung der in der Policy angegebenen Aktionen auf den Zielobjekten. Die *negative Autorisierungspolicy* verbietet ausdrücklich deren Ausführung durch die Subjekte. Eine *positive Verpflichtungspolicy* erzwingt die Durchführung der in der Policy angegebenen Anweisungen durch das Subjekt. Eine *negative Verpflichtungspolicy* hingegen verpflichtet zur Ausführung aller Operationen, die nicht spezifiziert worden sind. An dieser Stelle sei darauf hingewiesen, daß diese sehr einfache Klassifizierung von Policies, die für das Management nomadischer Systeme im Rahmen dieser Arbeit als ausreichend bewertet wird, in [Wies95b] um eine viel verfeinerte Einteilung durch Einführung weiterer Kriterien (Life-Time einer Policy, Managementszenario, Funktionalität, etc.) erweitert wurde.

Nach diesen einführenden Worten und Begriffsklärungen werden in den nachfolgenden Kapiteln das *Refinement* von Policies, basierend auf einer Policy Hierarchie, sowie die *Policy Description Language (PDL)* vorgestellt, die es ermöglicht, maschineninterpretierbare und damit durchsetzbare Policies für das NoS-Management zu formulieren.

3.2 Die Policy Hierarchie

Im vorhergehenden Kapitel wurde bereits festgestellt, daß Policies der unterschiedlichsten Abstraktionsebenen existieren. Ein Ziel muß es deswegen sein, diese Ebenen zu spezifizieren und deren Abstraktionsgrad genau festzulegen, um den Entwicklungsprozeß von einer *High-level-Policy* zu einer *Low-level-Policy* beschreiben zu können. In [Koch96] wird zwischen *drei Hierarchiestufen* unterschieden, das durchaus als zweckmäßig zu bewerten ist. Mehr Stufen führen zu einer erschwerten Abgrenzung und einem erhöhten Aufwand der Überführung der Policies von der obersten zur untersten Ebene. Weniger Stufen, d.h. in diesem Fall nur zwei bzw. eine Stufe, erschweren hingegen das Widerspiegeln des tatsächlichen Entwicklungsprozesses und damit die Formulierung der Policies durch den Systemadministrator.

In Abbildung 3-1 sind die drei Hierarchiestufen der Policies dargestellt, die zwischen den folgenden *Policy Typen* differenzieren lassen:

Strategische Policy

Eine *strategische Policy* enthält eine informelle Aussage über das verfolgte und zu erreichende Ziel des Netz- und Systemmanagements. Sie dient der ersten, meist recht ungenauen Spezifizierung der Intention des Unternehmensmanagements. Für eine Automation der Herstellung dieser Policies bzw. ihrer Überführung in Policies der nächst niedrigeren Stufen, sind diese jedoch aufgrund ihres in Prosa verfaßten Inhalts ungeeignet.

Zielorientierte Policy

In den *zielorientierten Policies* werden bereits die Subjekte, Zielobjekte, Aktionen und Constraints konkretisiert. Obwohl die zielorientierten Policies, ähnlich wie die strategischen Policies, in einer eher umgangssprachlichen Art und Weise formuliert werden und nicht an eine feste Syntax gebunden sind, bietet es sich an, Templates für diese anzulegen (siehe auch Kapitel 3.4), die entsprechende Felder für die zu spezifizierenden Objekte aufweisen. In diesem Fall würde der Systemadministrator die Möglichkeit haben, zielorientierte Policies zu generieren, indem eine Art Formular ausgefüllt wird. Die tatsächliche Belegung der Felder wird jedoch weder interpretiert noch überprüft und kann demzufolge mit beliebigen Werten besetzt werden.

Operationale Policy

Im Gegensatz zu den Policies der vorhergehenden Hierarchiestufen, werden die *operationalen Policies* in einer festgelegten *Policy Description Language (PDL)* spezifiziert. Durch die feste Syntax und Semantik der in PDL formulierten Policies wird deren Interpretation durch eine Managementanwendung ermöglicht. Gegenüber den zielorientierten Policies werden alle Angaben in der Weise präzisiert, daß die Policy auf ihre Anwendbarkeit überprüft werden kann. Die Policy-Aktionen sind z.B. konkrete Operationsaufrufe der Managementschnittstelle der Zielobjekte.

I.d.R. durchlaufen Policies alle drei Hierarchiestufen, so daß prinzipiell zu jeder operationalen Policy ihre zielorientierte und strategische Eltern-Policy festgestellt werden. Weiterhin sei darauf hingewiesen, daß es sich bei der Überführung von Policies in die nächst niedrigere Stufe, immer um 1:1 bzw. um 1:n Beziehungen handelt, so daß sowohl der Weg von der strategischen zur operationalen Policy als auch dessen Umkehrung immer eindeutig ist.

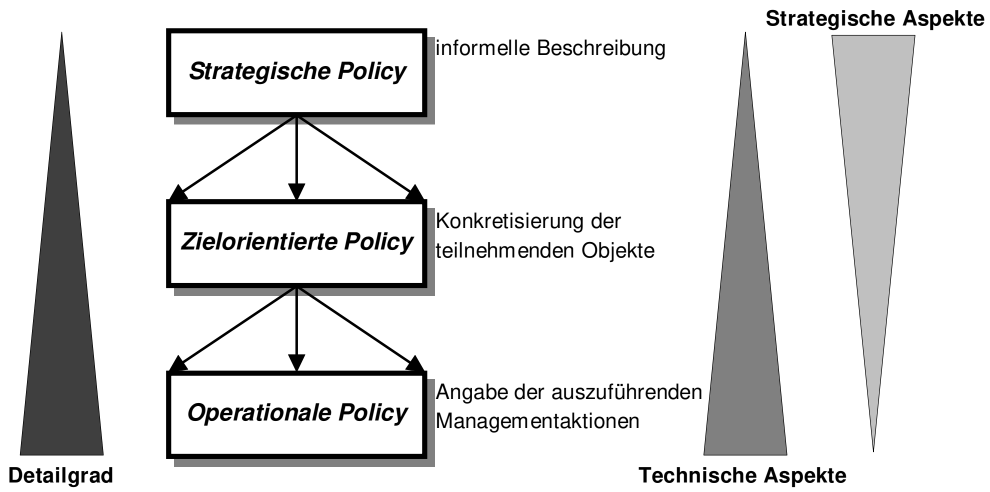

Abbildung 3-1: Die Policy Hierarchie (nach [Koch96])

3.3 Der Policy Entwicklungsprozeß

Der Entwicklungsprozeß einer Policy ist grundsätzlich schon durch die im vorhergehenden Kapitel beschriebene Policy Hierarchie determiniert. In Abbildung 3-2 ist der angestrebte Ablauf von der Generierung einer strategischen Policy über deren Überführung in operationale Policies bis zur deren Interpretierung durch ein Managementsystem veranschaulicht und lautet wie folgt:

(1) Der Systemadministrator generiert zunächst eine strategische Policy, indem informell das zu erreichende Ziel formuliert wird.

(2) Mit Hilfe des zur Verfügung gestellten Policy-Systems wird der *Refinement*-Prozeß durchgeführt, dessen Endprodukt schließlich die in PDL spezifizierten operationalen Policies sind.

(3) Die operationalen Policies werden, nach einer eventuellen Überprüfung ihrer Anwendbarkeit, persistent angelegt, indem sie in eine Datenbank (DB) eingefügt werden.

(4) Wird die operationale Policy durch den Systemadministrator aktiviert, so kümmert sich ein Managementsystem um deren Durchsetzung, indem die Policy interpretiert wird.

(5) Indem das Managementsystem auf bereits existierende *Managementagenten* aufsetzt, werden die spezifizierten Zielobjekte beobachtet. Bei Auftreten relevanter Events, werden die in der operationalen Policy angegebenen Managementoperationen ausgeführt.

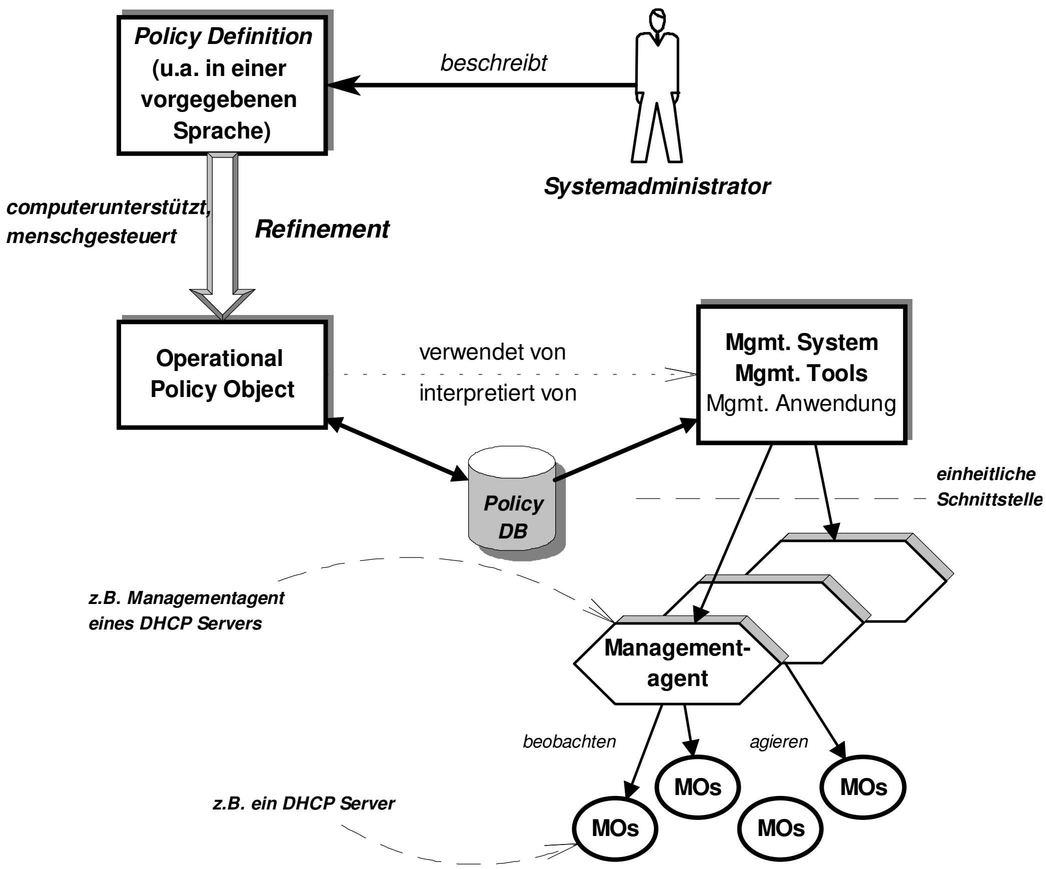

Abbildung 3-2: Der Policy Entwicklungsprozeß (nach [Wies95b])

Die tatsächliche Schwierigkeit bei der Entwicklung von operationalen Policies liegt ohne Zweifel im *Refinement* der Policies. An dieser Stelle ist es aufgrund der Komplexität des Refinement-Prozesses bisher nicht gelungen, zufriedenstellende Ergebnisse bei der Computerunterstützung zu erreichen. Neben der reinen Überführung der Policies in die nächst niedrigere Hierarchiestufe, fällt in den Refinement-Prozeß ebenfalls die nicht triviale Aufgabe der Konfliktaufdeckung zwischen den neu generierten und bereits existierenden Policies. Folglich ist der Systemadministrator meist bei dieser Aufgabe völlig auf sich selbst gestellt. Um hier dennoch dem Policy-Ersteller eine Hilfestellung geben zu können, bietet es sich an, Vorgehensmethoden und -modelle des *Software-Engineerings* anzuwenden. Das wohl bekannteste und weitverbreitetste Modell ist das *Wasserfallmodell*, das in Abbildung 3-3 veranschaulicht ist. Jede Phase wird durch eine Validierung abgeschlossen, in der u.a. die vorhergehende Phase auf die dort festgelegten Aspekte überprüft wird. Werden gewisse Differenzen zwischen zwei Phasen festgestellt, so erfolgen Anpassungen, die durchaus auch vorhergehende Phasen mit einbeziehen können, bis diese Differenzen behoben sind. So ergibt sich in der Phase der Generierung der operationalen Policies die Möglichkeit bei Aufdeckung eines Konflikts, die zielorientierten Policies neu zu formulieren. Ist der Konflikt damit nicht zu beheben, so muß an der strategischen Policy angesetzt werden, usw. Damit wird aber die Entdeckung eines Konflikts bereits vorausgesetzt. Oft liegt aber genau hier die Schwierigkeit. Deswegen wird diese Problematik gesondert in Kapitel 3.6 betrachtet.

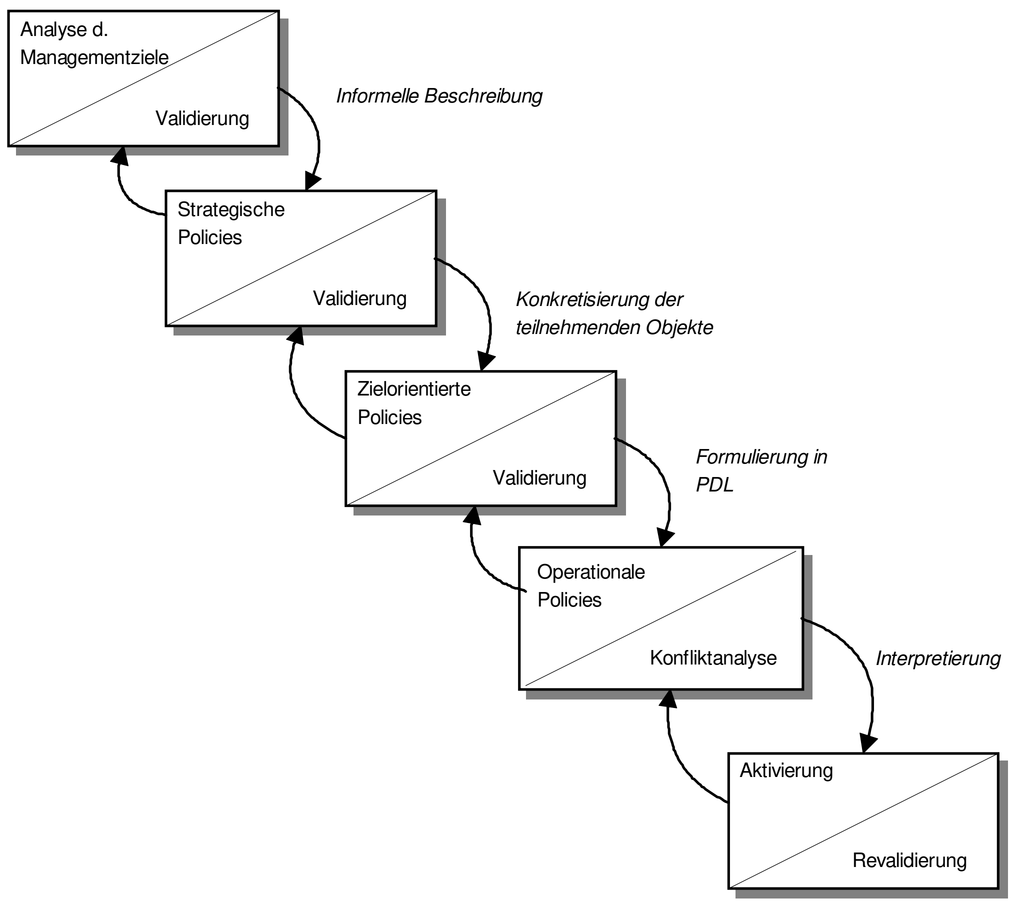

Abbildung 3-3: Vorgehensmodell bei der Policy-Erstellung

In Abbildung 3-4 ist zusätzlich der Lebenszyklus einer operationalen Policy in Form eines Petrinetzes dargestellt. Wird die operationale Policy als ein unabhängiges Objekt angenommen und nicht nur als ein in PDL formulierter Satz, so wäre es denkbar, daß es selbständig die Zielobjekte beobachtet und auf diesen agiert, indem es als Objekt direkt auf den *Managementagenten* aufsetzt. Das Managementsystem wäre daraufhin nur noch für die Verwaltung der Policy Objekte zuständig. Dieser Gedanke wird in Kapitel 4 durch Betrachtung einer konkreten Umsetzung weiterverfolgt.

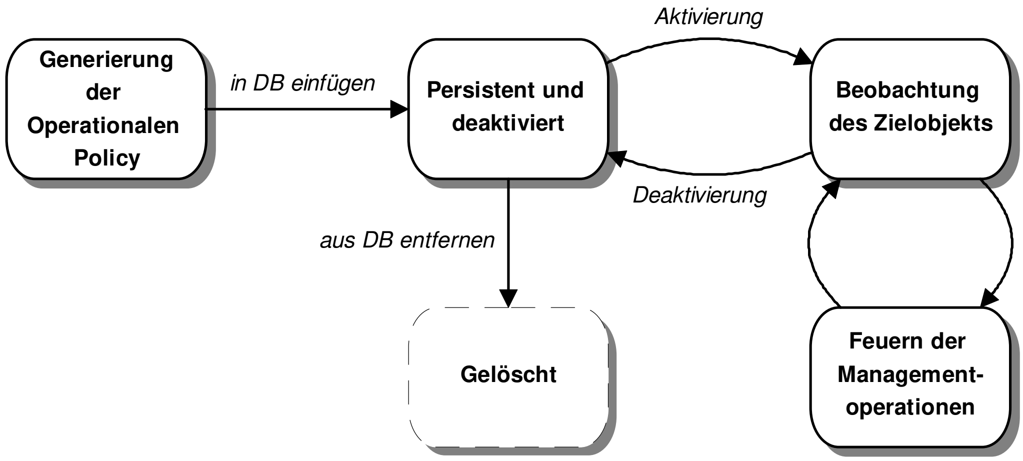

Abbildung 3-4: Der Lebenszyklus einer operationalen Policy

3.4 Templates für zielorientierte Policies

In Kapitel 3.2 wurde bereits festgestellt, daß die Entwicklung von *Templates* für zielorientierte Policies trotz ihrer noch z.T. informell, d.h. umgangssprachlich getroffenen Aussagen, durchaus ihre Berechtigung haben, um v.a. dem Policy-Ersteller die Generierung der Policies durch eine semi-formale Syntax zu erleichtern.

Ein Policy-Template besteht grundsätzlich aus einer Anzahl von *Feldern*. Jedes Feld ist zweigeteilt: der erste Teil enthält den *Namen* des Feldes, während der zweite Teil für die *Belegung* durch den Policy-Ersteller freigehalten wird. Jedes Feld kann mit einer beliebigen Zeichenfolge besetzt werden. Es ist dennoch ratsam, gewisse syntaktische Regeln einzuführen, um einerseits das Erstellen von Policies möglichst durch die Eindeutigkeit der verwendeten Felderbelegungen zu erleichtern und andererseits die Möglichkeit offen zu halten, zielorientierte Policies eventuell zu einem späteren Zeitpunkt computerunterstützt zu verarbeiten. Das Befolgen der syntaktischen Regeln ist und bleibt aber trotzdem freiwillig, auch wenn dies wünschenswert ist.

Es stellt sich natürlich die Frage, welche Template-Felder überhaupt zu definieren sind. Ein Teil dieser Felder ergeben sich schon aus den in Kapitel 3.1 analysierten Bestandteilen einer verfeinerten Policy. Dazu zählen die *Modalität*, die *Subjekte*, die *Zielobjekte*, die *Aktionen* und das *Constraint* der Policy. Zusätzlich kommt der *Name* der Policy sowie *Domänenbezeichnungen* für die Subjekte und Zielobjekte hinzu. Eine *Domäne* ist hierbei eine Menge verwalteter Objekte, die vorher, z.B. durch Nutzung eines entsprechenden Dienstes, gebildet worden ist. Ein verwaltetes Objekt kann zu mehreren Domänen gehören, so daß durch geeignete Domänenbildung einerseits die Struktur des vorliegenden Netzes widergespiegelt werden kann, als auch andererseits unterschiedliche Sichtweisen auf die verwalteten Objekte realisiert werden können. Mit der Domänenbildung ist es folglich auch möglich *Rollen* für die Objekte zu modellieren. Die Konstruktion der Domänenstruktur spielt demzufolge eine wesentliche Rolle, um den Kreis der von einer Policy betroffenen Objekte identifizieren und v.a. korrekt spezifizieren zu können.

Das im folgenden beschriebene Template für zielorientierte Policies basiert auf dem in [Koch96] entwickelten Template, wurde aber um die besagten Domänenangaben sowie weiteren Formalismen bezüglich der Felderbelegung erweitert und erstmals in [Heil98] veröffentlicht.

Nachstehend werden die Felder, deren Bedeutung sowie die syntaktischen Vorgaben in Form einer Tabelle vorgestellt:

Feldname	***Feldbedeutung***	***Syntax***
Name	der Name der zielorientierten Policy	beliebige Zeichenfolge
Modality	die Modalität der Policy	entweder *Permission/Prohibition* oder *Obligation*
Subject	der Typ derjenigen Objekte, die als Subjekte der Policy auftreten	Es können Variablen definiert werden, die, jeweils durch ein Komma getrennt, mit einem Doppelpunkt dem *Typ* vorangestellt werden. Dadurch können die so definierten Variablen in anderen Feldern wieder verwendet werden. Die *Objekttypen* sind nicht festgelegt, sollten aber verständlich deklariert sein. Beispiel: *s: nos*
SDomain	die Domäne der Subjekte	beliebige Zeichenfolge, wobei auch hier die Möglichkeit der Variablendeklarierung besteht Beispiel: *sd: private*
Target	der Typ derjenigen Objekte, die als Zielobjekte der Policy auftreten	Syntax wie beim *Subject*-Feld Beispiel: *t: dhcp*
TDomain	die Domäne der Zielobjekte	Syntax wie beim *SDomain*-Feld Beispiel: *td: lounge*
Action	die Aktionen der Policy, die von den Subjekten aufgerufen werden dürfen bzw. müssen	beliebige Zeichenfolge für die Aktionsnamen, wobei mehrere Aktionen durch ein Semikolon getrennt werden und vorhergehend definierte Variablen als Funktionsparameter oder -aufrufer vorkommen können Beispiel: *s.Use(t)*
Constraint	die im Constraint spezifizierten Bedingungen müssen vor der Anwendung einer Policy erfüllt sein	geklammerter, logischer Ausdruck, der sowohl Funktionsaufrufe als auch Variablen enthalten kann Beispiel: *(sd.authenticated(s) && s.hasAccount(td))*

Es ist ausschließlich bei den zielorientierten Policies möglich, diese auf derselben Hierarchieebene zu verfeinern, um präzisere zielorientierte Policies zu erhalten. Die grobgranulareren

Policies sollten in diesem Fall aber gelöscht werden, um die direkte 1:1 bzw. 1:n Beziehung zwischen strategischen und zielorientierten Policies beizubehalten und Verwirrungen zu vermeiden.

In Kapitel 3.7 sind Beispiele für die Entwicklung von strategischen Policies beschrieben.

3.5 PDL für operationale Policies

3.5.1 Anforderungen

Eine *operationale Policy* ist, im Gegensatz zu den Policies höherer Hierarchiestufen, in der Weise formuliert, daß diese in konkrete Managementanweisungen umgesetzt werden kann. Um die Interpretierung der Policy zu vereinfachen, enthält eine operationale Policy bereits eine Sequenz von *Managementoperationen*, die bei Auftreten eines Events ausgeführt werden.

Bevor jedoch eine maschineninterpretierbare Sprache für operationale Policies spezifiziert werden kann, muß zunächst eine Anforderungsanalyse stattfinden, um den Sprachumfang abschätzen zu können. Aus den bereits festgelegten *Templates* für zielorientierte Policies können bereits ein Teil der Elemente der *Policy Description Language (PDL)* angegeben werden. Zweifelsohne müssen auch bei den operationalen Policies *Subjekte*, *Zielobjekte*, *Domänen, Aktionen* und *Constraints* deklariert werden können. Ein Ziel bei der Definition der PDL ist es selbstverständlich, deren Grammatik und Wortschatz so einfach wie möglich zu gestalten. Dadurch wird einerseits ein Overhead an Informationen vermieden und andererseits ein möglichst intuitiver Umgang mit der Sprache erhofft. Dies verlangt aber jedoch nach einer präzisen Analyse der bisher identifizierten Komponenten der Sprache sowie der vorliegenden *Managementarchitektur (MA)*. Die Managementarchitektur wird ausführlich in Kapitel 4 dargestellt, so daß an dieser Stelle nur ein einführender Blick getätigt wird. Aus dem Bibliotheksszenario in Kapitel 2 wurde bereits in Abbildung 3-2 eine erste Skizze der MA angefertigt. Die MA setzt sich aus der Managementanwendung, den Managementagenten und den Managementobjekten zusammen. Die Managementanwendung interpretiert die operationale Policy und setzt diese durch, indem sie auf den Agenten agiert. Die Agenten stellen einen einheitlichen Zugriff auf ihre Managementschnittstelle zur Verfügung, das in diesem Fall durch eine *CORBA IDL Schnittstelle* realisiert wird. Die gesamte Kommunikation der Managementanwendung mit den Agenten findet in einer CORBA Umgebung statt, so daß die durch CORBA zur Verfügung stehenden *CORBA Services*, wie der *Naming Service* und der *Event Service*, genutzt werden.

Für die bereits identifizierten Komponenten der Sprache, gelten die folgenden Feststellungen:

- *Subjekte*: Im Template für zielorientierte Policies wird im Subjektfeld der *Typ* des Objekts angegeben. Im vorliegendem Fall treten als Subjekte ausschließlich *nomadische Systeme* auf, so daß implizit bei den hier vorliegenden operationalen Policies immer von dieser Tatsache ausgegangen werden kann.
- *Subjektdomäne*: Da es sich, wie schon festgestellt wurde, bei den Subjekten immer um nomadische Systeme handelt, ist die interessierende Subjektdomäne die *Heimatdomäne* der NoS.

- *Zielobjekte (Targets)*: Wie schon aus dem Bibliotheksszenario aus Kapitel 2 gefolgert werden kann, handelt es sich bei den Zielobjekte immer ausschließlich um die *Managementagenten* derjenigen *Ressourcen*, die ein NoS nutzen könnte. Wie in der zielorientierten Policy, werden ebenfalls in der operationalen Policy nur Zielobjekte genau *eines* Managementagententyps angegeben.
- *Domäne der Zielobjekte*: Konsequenterweise wird die *Domäne der Managementagenten* der Ressource angegeben. Dies sollte auch zugleich die Domäne sein, in der das NoS versucht die Ressource zu nutzen.
- *Aktionen*: Es werden die, bei Eintritt eines Events, auszuführenden Operationen der Managementschnittstelle der Agenten notiert. Um eine Vereinfachung der Sprache zu erreichen, wird die Forderung aufgestellt, daß die Agenten einen einheitlichen Zugriff auf ihre bereitgestellte Schnittstelle anbieten. Im vorliegenden Fall müssen also alle Agenten eine CORBA IDL-Schnittstelle aufweisen.
- *Event*: Wie zu Anfang dieses Abschnitts bereits festgestellt wurde, werden die in einer operationalen Policy angegebenen Anweisungen bei Eintritt eines Events ausgeführt. Im vorliegenden Fall ist dieses Event immer der *Versuch eines NoS eine Ressource zu nutzen*. Der Managementagent der Ressource leitet dieses Event an das MS weiter und fügt zusätzliche Managementinformationen hinzu, wie z.B. die Identität des NoS, die Funktionalität der Ressource, die das NoS nutzen will, etc. Die Struktur dieser, von den Agenten verschickten Information, ist jeweils von Agent zu Agent verschieden und muß folglich in der Policy angegeben werden.
- *Constraint*: Der Constraint ist, ähnlich wie bei der zielorientierten Policy, eine aussagenlogische Formel, die zum Zeitpunkt des Auftretens eines Events ausgewertet wird. Nur falls die Auswertung positiv ausfällt, werden die in der Policy angegebenen Anweisungen ausgeführt.

Offensichtlich fehlt in den hier analysierten operationalen Policies die Spezifizierung der *Modalität* der Policy. Der Grund hierfür liegt darin, daß der Aktionsteil der Policy nicht mehr Operationen enthält, die vom Subjekt ausgeführt werden *müssen* oder *dürfen*, sondern Operationen, die an der Managementschnittstelle von Agenten aufgerufen werden, nachdem das Event *'NoS versucht Ressource zu nutzen'* aufgetreten ist und das Constraint positiv ausgewertet wurde. D.h. insbesondere, daß bei der Überführung einer zielorientierten Policy in operationale Policies, eine *Verpflichtung* bzw. *Autorisierung* des Subjekts in geeignete Managementanweisungen an die Agenten der betroffenen Ressourcen überführt werden muß.

3.5.2 Die Syntax der PDL

Aus der Anforderungsanalyse läßt sich bereits eine kontextfreie Grammatik für die formale Sprache PDL angeben. Für die Darstellung der PDL wird die weitverbreitete *Extended Backus-Naur-Form (EBNF)* [Krieg94] eingesetzt, für die folgendes gilt:

- *Terminalsymbole* werden in Anführungszeichen notiert, z.B. „POLICY“
- *Nichtterminalsymbole* werden zur besseren Unterscheidung von den Terminalsymbolen in spitze Klammern gesetzt, also z.B. <PolicyDeclaration>

- es gilt folgende Meta-Syntax:

  = „ist definiert als ...“

  . . . | . . . genau eine Alternative aus der Liste muß stehen

  [. . .] Inhalt der Klammer kann stehen oder nicht

  {. . .} Inhalt der Klammer kann n-fach stehen ($n \geq 0$)

In Abbildung 3-5 sind die reservierten Schlüsselwörter definiert. Desweiteren sind in Abbildung 3-6 die verwendeten Interpunktionszeichen dargestellt.

POLICY	FOR	ON	DELIVERS
ACTION	CONSTRAINT	Subject	long
string	boolean	true	false

Abbildung 3-5: Die reservierten PDL-Bezeichner

{	}	(	)
:	;	,	.
<	>	=	! (Negation)
‖ (Disjunktion)	&& (Konjunktion)		

Abbildung 3-6: Die PDL-Interpunktionszeichen

In Abbildung 3-7 sind allgemeine Grammatikregeln spezifiziert. Insbesondere gilt für einen allgemeinen *Namensbezeichner* (identifier), daß dieser mit einem Buchstaben anfangen muß, ansonsten aber sowohl Ziffern als auch Bindestriche und Unterstriche enthalten kann. Neben dem allgemeinen Namensbezeichner wurde zusätzlich ein *Domänenbezeichner* (xfnidentifier) definiert, der jeweils mehrere Namensbezeichner mit einem Schrägstrich voneinander trennt. In diesem Kontext repräsentieren jeweils die einzelnen Namensbezeichner eine Domäne, so daß der gesamte Ausdruck ein Objekt bzw. eine Objektmenge eindeutig identifiziert. Der *erweiterte Domänenbezeichner* (extended_xfn) ist speziell für den Einsatz mit dem CORBA Naming Service spezifiziert worden. Innerhalb des Naming Service ist ein Name zweigeteilt, bestehend aus einem *identifier*- und einem *kind*-Attribut, der von der Struktur her einem UNIX-Dateinamen ähnlich ist. Diese beiden Attribute eines CORBA-Namens werden durch einen Punkt getrennt.

In Abbildung 3-8 ist schließlich die komplette Syntax der PDL in EBNF veranschaulicht.

<table>
<tr><td><TQLQUERY></td><td>=</td><td>„TQL“ „ : “ {<LETTER> | <DIGIT> | „ “ | „ - “ | „ _ “ | „ . “ | „ / “ | „ ″ “ | „ (“ | „) “ | „ { “ | „ } “ | „ [“ | „] “}</td></tr>
<tr><td><NSQUERY></td><td>=</td><td>„NS“ „ : “ <EXTENDED_XFN></td></tr>
<tr><td><XFNIDENTIFIER></td><td>=</td><td>„ / “ <IDENTIFIER> {„ / “ <IDENTIFIER>} [„ / “]</td></tr>
<tr><td><EXTENDED_XFN></td><td>=</td><td>„ / “ <IDENTIFIER> „ . “ <IDENTIFIER> {„ / “ <IDENTIFIER> „ . “ <IDENTIFIER>} [„ / “]</td></tr>
<tr><td><IDENTIFIER></td><td>=</td><td><LETTER> {<LETTER> | <DIGIT> | „ - “ | „ _ “}</td></tr>
<tr><td><STRINGCONSTANT></td><td>=</td><td>„ ″ “ {<LETTER> | <DIGIT> | „ - “ | „ _ “ | „ : “} „ ″ “</td></tr>
<tr><td><SCALAR></td><td>=</td><td><DIGIT> {<DIGIT>}</td></tr>
<tr><td><LETTER></td><td>=</td><td>„a“ | „b“ | „c“ | . . . | „z“ | „A“ | „B“ | „C“ | . . . | „Z“</td></tr>
<tr><td><DIGIT></td><td>=</td><td>„0“ | „1“ | „2“ | . . . | „9“</td></tr>
</table>

Abbildung 3-7: Allgemeine Grammatikregeln

<table>
<tr><td><inputLine></td><td>=</td><td><PolicyDeclaration> <DeliversDeclaration> <ActionDeclaration> <ConstraintDeclaration></td></tr>
<tr><td><PolicyDeclaration></td><td>=</td><td>„POLICY“ <IDENTIFIER> „FOR“ <XFNIDENTIFIER> „ON“ <DomainExpression></td></tr>
<tr><td><DomainExpression></td><td>=</td><td>„<“ (<TQLQUERY> | <NSQUERY>) „>“</td></tr>
<tr><td><DeliversDeclaration></td><td>=</td><td>„DELIVERS“ <StructEventDecl> {„ ; “ <StructEventDecl>}</td></tr>
<tr><td><StructEventDecl></td><td>=</td><td><IDENTIFIER> „{“ <StructAttributeDeclaration> { „ ; “ <StructAttributeDeclaration>} „}“</td></tr>
<tr><td><StructAttributeDeclaration></td><td>=</td><td>(„string“ | „long“ | „boolean“) <IDENTIFIER></td></tr>
<tr><td><ActionDeclaration></td><td>=</td><td>„ACTION“ „{“ <Action> {„ ; “ <Action> } „}“</td></tr>
<tr><td><Action></td><td>=</td><td><Method> | <FunctionDesignator></td></tr>
<tr><td><ConstraintDeclaration></td><td>=</td><td>„CONSTRAINT“ [„ConstraintExpressions“]</td></tr>
<tr><td><ConstraintExpressions></td><td>=</td><td>„(“ <ConstraintExpression> „)“
| „!“ „(“ <ConstraintExpression> „)“</td></tr>
<tr><td><ConstraintExpression></td><td>=</td><td><ConstraintExpressions> („&&“ | „||“) <ConstraintExpressions>
| <SimpleExpression></td></tr>
<tr><td><SimpleExpression></td><td>=</td><td>(<EventDesignator> | <FunctionDesignator>) <OperatorExpression></td></tr>
<tr><td><OperatorExpression></td><td>=</td><td>(„<“ | „>“) (<SCALAR> | <EventDesignator> | <FunctionDesignator>)
| „=“ (<Constants> | <EventDesignator> | <FunctionDesignator>)</td></tr>
<tr><td><FunctionDesignator></td><td>=</td><td><Object> „ . “ <Method></td></tr>
<tr><td><EventDesignator></td><td>=</td><td><IDENTIFIER> „:“ <IDENTIFIER></td></tr>
<tr><td><Object></td><td>=</td><td>„Subject“ | <EXTENDED_XFN></td></tr>
<tr><td><Method></td><td>=</td><td><IDENTIFIER> „(“ Arguments „)“</td></tr>
<tr><td><Arguments></td><td>=</td><td>[<Argument> {„ , “ <Argument> }]</td></tr>
<tr><td><Argument></td><td>=</td><td><Constants> | <EventDesignator> | <FunctionDesignator></td></tr>
<tr><td><Constants></td><td>=</td><td>(<SCALAR> | <STRINGCONSTANT> | <TRUE> | <FALSE>)</td></tr>
</table>

Abbildung 3-8: Die PDL-Syntax

3.5.3 Die Semantik der PDL

Für die Erläuterung der einzelnen syntaktischen Einheiten der PDL, werden die folgenden Variablen verwendet:

- *name*, *event*, *idl_operation, domain* und *attribute* sind *einfache Namensbezeichner* aus der Wortmenge <IDENTIFIER>
- *subject* ist ein *einfacher Domänenbezeichner* aus der Wortmenge <XFN_IDENTIFIER>
- *target* ist ein Suchausdruck aus der Wortmenge <NSQUERY> oder <TQLQUERY>
- *object* ist ein *erweiterter Domänenbezeichner* aus der Wortmenge <EXTENDED_XFN>

In Abbildung 3-9 ist der schematische Aufbau einer operationalen Policy dargestellt, aus der bereits die vier Teile einer Policy hervorgehen. Im weiteren werden diese vier Teile separat betrachtet und erklärt.

```
POLICY name FOR subject ON  < target >
DELIVERS event {string | long | boolean  attribute}
ACTION {idl_operation( ) | Subject.idl_operation( ) | object.idl_operation( )}
CONSTRAINT (... && ... | ... | | ...)
```

Abbildung 3-9: schematischer Aufbau einer operationalen Policy

Die erste syntaktische Einheit, deren schematischer Aufbau in Abbildung 3-10 dargestellt ist, deklariert neben dem *Namen* der Policy ebenfalls den *Subjektbereich* und den *Zielobjektbereich*. Wie schon am Anfang dieses Abschnitts bereits erwähnt, wird implizit angenommen, daß es sich bei den Subjekten ausschließlich um nomadische Systeme handelt, so daß der Subjektbereich die *Heimatdomäne* der NoS eingrenzt. Es wird ebenfalls implizit angenommen, daß es sich bei den Zielobjekten um Managementagenten handelt, die beobachtet werden sollen. Um die sich gegenwärtig im Netz befindenden Managementagenten ausfindig machen zu können, wird ein Suchausdruck angegeben, der die Menge der Zielobjekte zurückgibt. Wird der *CORBA Naming Service* für die Deklaration des Suchausdrucks verwendet, so wird diesem „NS: “ vorangestellt, ansonsten wird der *CORBA Topology Service* verwendet, dessen Suchausdruck mit „TQL: “ beginnt. Mit der Verwendung des *Naming Service* muß beachtet werden, daß der Suchausdruck immer entweder auf genau einen Agenten oder auf eine Menge von Agenten verweist, die sich genau auf der nächsten Domänenebene befinden.

```
POLICY name FOR subject ON  < NS: domain/ . . . /(domain | object_name)/  |
                                          TQL: tql_query >
```

Abbildung 3-10: schematischer Aufbau der Policy-Deklaration

Die zweite syntaktische Einheit deklariert die von den Agenten gesendeten Events und ist in Abbildung 3-11 schematisch veranschaulicht. Jeder Agent sendet festgelegte, agentenspezifische Events, die der Policy-Ersteller kennen muß. Es werden nur diejenigen Events notiert, bei deren Eintreten die spezifizierten Aktionen ausgeführt werden sollen. Ein Event wird eindeutig durch seinen Namen identifiziert, der aus dem Wortbereich <IDENTIFIER> gebildet wird. Desweiteren besteht ein Event aus mehreren Attributen, die weiterführende Informationen über die Eventursache, den Eventverursacher, etc. enthalten. Es müssen nur diejenigen Attribute eines Events deklariert werden, die in einer der weiteren syntaktischen Einheiten zur Weiterverwertung als Operationsparameter oder Constraintvergleichswert verwendet werden. Jedes Attribut ist innerhalb eines Events eindeutig durch seinen Namen identifizierbar, der aus dem Wortbereich <IDENTIFIER> stammt. Neben dem Namen muß auch der Typ des Attributs angeführt werden, wobei nur zwischen *string*, *long* und *boolean* unterschieden wird.

```
DELIVERS event1 {string | long | boolean  attribute1;
                 string | long | boolean  attribute2;
                 . . .
                 string | long | boolean  attributeN };
         event2  { . . . };
         . . .
         eventM { . . . }
```

Abbildung 3-11: schematischer Aufbau der Event-Deklaration

In der dritten syntaktischen Einheit, die in Abbildung 3-12 schematisch dargestellt ist, werden die, beim Eintreten eines deklarierten Events, auszuführenden Aktionen spezifiziert. Es handelt sich bei den Aktionen ausschließlich um *IDL Operation*, die an den *IDL Interfaces* der entsprechenden CORBA Server Objekte aufgerufen werden. Wird bei einer deklarierten Operation kein Objekt ausdrücklich angegeben, so wird diese an der Schnittstelle des Managementagenten aufgerufen, der das auslösende Event gesendet hat. Andernfalls wird der Objektname aus dem Wortbereich <XFNIDENTIFIER> der Operation vorangestellt. Es besteht ebenfalls die Möglichkeit, eine Operation an der Schnittstelle des *betroffenen Subjekts* aufzurufen, das durch das Voranstellen des Wortes „Subject“ angezeigt wird. Jedes Subjekt, das in diesem Fall immer ein nomadisches System ist, wird durch ein CORBA Objekt repräsentiert, das eine einheitliche IDL Schnittstelle aufweist. Dieses Objekt wird beim erstmaligen Auftreten des nomadischen Systems kreiert und spiegelt dessen augenblicklichen Zustand wider. Durch entsprechende Operationsaufrufe, kann dieser Zustand geändert werden. Als Parameter für die Operationen sind skalare Werte, Strings, boolesche Werte sowie weitere Operationsaufrufe gestattet. Es ist ebenfalls möglich sich auf Attribute des gesendeten Events zu beziehen. In diesem Fall wird der adäquate Eventname und der Attributname, die durch einen Doppelpunkt getrennt sind, angeführt. Nur diejenigen Eventattribute sind zugelassen, die im DELIVERS-Abschnitt deklariert worden sind, da ansonsten eine korrekte Verarbeitung nicht möglich ist.

```
ACTION { idl_operation( event:attribute, true, 2, . . ., "foo" );
         Subject.idl_operation( object.idl_operation( . . . ), . . ., false );
         object.idl_operation( Subject.idl_operation(. . . ), . . .)
        }
```

Abbildung 3-12: schematischer Aufbau der Action-Deklaration

Die letzte syntaktische Einheit, deren schematische Struktur in Abbildung 3-13 veranschaulicht ist, beinhaltet die Deklaration des Constraints der Policy. Nur wenn die Evaluierung des Constraint-Ausdrucks positiv ausfällt, werden die spezifizierten Aktionen gefeuert. Der Basisausdruck eines Constraints ist immer ein Vergleich. Als Operanden der üblichen Vergleichsoperatoren (=, <, >), können Konstanten, Operationsaufrufe sowie Eventattribute notiert werden. Der Basisausdruck steht immer in Klammern. Mehrere Basisausdrücke können durch die booleschen Operatoren „&&“ und „| |“ miteinander verbunden werden. Bei der Auswertung des Constraints wird *nicht* die übliche Regel „Und-Bindung vor einer Oder-Bindung“ angewendet, statt dessen ist nur die Klammerung der Ausdrücke ausschlaggebend.

```
CONSTRAINT ( (event:attribute =  "foo") && (Subject.idl_operation( . . . ) < 2 )
              | | !(object.idl_operation( . . . ) = true)
             && (Subject.idl_operation( . . . ) > object.idl_operation( . . . ) )
              &&  . . . )
```

Abbildung 3-13: schematischer Aufbau der Constraint-Deklaration

In Kapitel 6 wird die Umsetzung der hier beschriebenen PDL in eine konkrete Implementierung durch Einsatz des *Java Compiler Compilers (JavaCC)* beschrieben.

3.6 Konfliktlösungsstrategien

In Abschnitt 3.3 wurde bereits die Problematik der Konfliktlösung zwischen Policies angesprochen. Nachdem die PDL für operationale Policies im vorhergehenden Abschnitt vorgestellt wurde, ist es nun möglich einige Strategien zur Lösung von Policykonflikten zu entwikkeln, die der Semantik der PDL angepaßt sind. Bereits in [LuSl97] werden die möglichen Konflikte zwischen Policies analysiert und Lösungsansätze betrachtet. Die nun folgende Analyse ist zum größten Teil eine Anpassung der in dieser Arbeit vorgestellten Ergebnisse.

Bevor Strategien zur Lösung dieser Problematik erarbeitet werden können, muß zunächst festgestellt werden, unter welchen Bedingungen Konflikte zwischen operationalen Policies auftreten, da die eigentliche Problematik in der Entdeckung der Konflikte liegt. Grundsätzlich tritt ein Konflikt immer dann auf, wenn zwei Policies eine *gegensätzliche Bedeutung* und damit auch eine *gegensätzliche Wirkung* beinhalten. Ein notwendiges, aber nicht hinreichendes Kriterium für einen Konflikt ist, daß sich die *Subjekt-* und *Zielobjektbereiche* zweier Policies *überlappen*. Aber erst wenn ebenfalls die spezifizierten Aktionen eine gegensätzliche Wirkung bei ihrer Ausführung haben, sowie das auslösende Event und die deklarierten Constraints gleich bzw. ähnlich sind, kann auch tatsächlich ein Konflikt bei ihrer Aktivierung auftreten.

Im folgenden werden die einzelnen Bestandteile einer in PDL formulierten operationalen Policy separat daraufhin untersucht, inwiefern sich Überschneidungen automatisch feststellen lassen, und an welchen Stellen zusätzlich die leitende Hand des Policy-Erstellers benötigt wird:

Subjektbereich

Der Subjektbereich schränkt die Heimatdomäne derjenigen nomadischen Systeme ein, die von der Policy betroffen sind. Wird davon ausgegangen, daß ein NoS zu mehreren Domänen gehören kann, so muß eine Auswertung des Subjektbereichs stattfinden, um die Menge der betroffenen NoS, und damit eine mögliches Überschneiden mit dem Subjektbereich anderer Policies, feststellen zu können. Dies kann ohne weiteres automatisch überprüft werden, jedoch hängt die Auswertung des Subjektbereichs vom Zeitpunkt ihrer Durchführung ab, so daß Überschneidungen durch Änderungen der Domänenzuteilung an ein NoS auftreten können, die vorher nicht entdeckt werden konnten. Dies trifft insbesondere immer dann auf, wenn neue NoS dem Managementsystem hinzugefügt werden. Um trotzdem in diesem Fall Überschneidungen feststellen zu können, müssen bei Domänenänderungen von NoS immer alle Policies auf Überschneidungen hin überprüft werden. Hierbei ist O(n) eine obere Schranke für den Suchaufwand.

Zielobjektbereich

Für den Zielobjektbereich gilt dasselbe wie für den Subjektbereich. Auch hier kann nur durch eine Evaluierung der Domänendeklaration eine Überschneidung bemerkt werden.

Eventdeklaration

An dieser Stelle reicht ein einfacher Vergleich der spezifizierten Eventnamen, um eine Überschneidung festzustellen. Dies kann, mit einer Komplexität von O(n), automatisch durchgeführt werden.

Constraint

Bei der hier festgelegten, gültigen Syntax für das Constraint, handelt es sich bei diesem Teil der PDL um einen Ausdruck der Aussagenlogik. Um aussagenlogische Formeln miteinander vergleichen zu können, bietet sich deren Konvertierung in die Klauselform, wie z.B. in die *disjunktive Normalform*[3], an. Die disjunktive Normalform hat das folgende Aussehen:

$$\left(\bigvee_{i=1}^{n}\left(\bigwedge_{j=1}^{m_i} L_{i,j}\right)\right)$$, wobei $L_{i,j}$ eine atomare Formel ist.

Eine atomare Formel wäre im vorliegenden Fall ein Vergleich zwischen z.B. einem Operationsrückgabewert und einer Konstante.

Das Constraint ist demzufolge immer dann wahr, wenn eine der Konjunktionen $L_{i.}$ wahr ist. Liegt nach der Umwandlung in die disjunktive Normalform der Fall vor, daß der gesamte Constraintausdruck Teil des Constraints einer anderen Policy wäre, so würden die Evaluierung beider Constraints denselben Wert zurückliefern und so-

[3] Jede aussagenlogische Formel läßt sich in die äquivalente *disjunktive Normalform* umformen (z.B. [Scho92]).

mit ein potentieller Konflikt eintreten. Liegt jedoch der Fall vor, daß jeweils nur ein Teil der atomaren Formeln in den Konjuktionen zweier Constraints gleich ist, so kann keine Aussage über deren Auswertung im voraus getroffen werden. Ebenfalls können zwei miteinander zu vergleichende Constraints in ihren atomaren Formeln verschieden sein und trotzdem immer denselben Wert zurückliefern. Folglich ist eine automatische Überprüfung der Constraints nur z.T. möglich und benötigt deswegen einer zusätzlichen Untersuchung (semantischer Art) durch den Policy-Ersteller.

Aktionen
Die bei Auftreten eines Events durchzuführenden Aktionen müssen durch den Policy-Ersteller überprüft werden, um eine gegensätzliche Wirkung der Aktionen feststellen zu können, da weder die Signatur noch die Semantik der Operationen, die von den Agenten angeboten werden, im vorliegendem Fall standardisiert sind. Es besteht demzufolge keine Möglichkeit diesen Vorgang zu automatisieren. Erst durch eine vorherige Festlegung von Standardschnittstellen und der Forderung, daß die Agenten genau eine Untermenge dieser Schnittstellen implementieren müssen, ist es möglich, den Aktionsbereich automatisch zu überprüfen.

Wie aus dieser Beschreibung zu ersehen ist, muß sowohl bei den Constraints als auch bei den Aktionen der Policy-Ersteller eingreifen. Ein entsprechender *Policy Service* kann eine Vorselektion der betroffenen Policies durchführen, indem die oben beschriebenen Überschneidungen festgestellt werden. Ein tatsächlicher Konflikt tritt nur genau dann ein, wenn eine Überschneidung in *allen* Bereichen festzustellen ist. Dabei ist die obige Reihenfolge bei den (automatischen) Vergleichen unbedingt einzuhalten, um möglichst früh einen mögliches Überlappen, und damit einen potentiellen Konflikt, ablehnen zu können.

Basierend auf diesem selektierendem Verfahren können folgende *Konfliktlösungsstrategien* auf Policies mit überschneidenden Bereichen angewendet werden:

„Negative" Policies haben höhere Priorität
Eine Policy, die Aktionen enthält, die einem Subjekt das Recht auf die Nutzung einer bestimmten Ressource einschränkt, hat immer eine höhere Priorität als eine Policy, die dem Subjekt die Nutzung der Ressource erlaubt.

Exakter spezifizierte Policies haben höhere Priorität
Ist in einer Policy der Subjekt- und/oder der Zielobjektbereich präziser angegeben, d.h. ist der eine Bereich eine echte Teilmenge des korrespondierenden Bereichs der zweiten Policy, so hat diese Policy immer eine höhere Priorität.

Explizite Zuweisung von Prioritäten
Ebenso besteht die Möglichkeit den Policies explizit Prioritäten zuzuweisen, um einen möglichen Konflikt bei deren Durchsetzung aufzulösen.

Aufhebung der Überlappung
Die trivialste Lösung eines Konflikts zwischen Policies, ist die Aufhebung der Überlappung der Policybereiche durch Neuformulierung der Policies. Dabei wird einfach die Schnittmenge der sich überschneidenden Bereiche bei einer der beiden Policies weggelassen. Sind die beiden Bereiche jedoch identisch, so müssen die Eltern-Policies der höheren Hierarchiestufen untersucht und ggf. neu spezifiziert werden.

Bei den ersten drei Lösungsstrategien spielen Prioritäten eine wesentliche Rolle. Eine Policy mit einer höheren Priorität soll sich gegenüber einer Policy mit einer niedrigeren Priorität durchsetzen. Umgesetzt könnte diese Strategie dadurch, daß bei einem tatsächlichen Eintreten eines Konflikts nur die Policy mit der höheren Priorität durchgesetzt wird. Jedoch wird bei näherer Betrachtung der Struktur einer operationalen Policy deutlich, daß zwei Policies sich prinzipiell nur in dem sich überlappenden Aktionsbereich widersprechen können. Demzufolge ist das Zurückhalten der Ausführung des sich nicht überschneidenden Bereichs weder erwünscht noch beabsichtigt, so daß zur Lösung dieses Dilemmas, die sich widersprechenden Aktionen eine gesonderte Markierung benötigen, um dies zu vermeiden.

Die hier vorgestellten Strategien konnten in dem, während dieser Arbeit entwickelten Prototypen, noch nicht verwirklicht werden, so daß an dieser Stelle für spätere Erweiterungen angesetzt werden kann, um den *Policy Service* zu verbessern.

3.7 Policies für nomadische Systeme

Im folgenden wird anhand des Bibliotheksszenarios die Entwicklung operationaler Policies dargestellt.

Beispiel 1

Aus der Beschreibung des Beispielszenarios des 2. Kapitels, kann u.a. folgende *strategische Policy* entnommen werden:

> *Nur die Mitarbeiter der Bibliotheken dürfen die Ressourcen im nichtöffentlichen Bereich benützen.*

Jetzt bieten sich dem Policy-Ersteller mehrere Möglichkeiten an, diese strategische Policy in *zielorientierte Policies* umzuwandeln. Einerseits könnten allen Ressourcen ein Verbot ausgesprochen werden, daß diese durch Nichtmitarbeiter genutzt werden. Ein besserer Weg ist jedoch, die Verbindung zur Ressource *Switch* im nichtöffentlichen Bereich nur für Mitarbeiter zu erlauben, da dadurch auch die Kommunikation mit anderen Netzkomponenten und -ressourcen eingeschlossen wird. Es ergibt sich also folgende zielorientierte Policy:

Name	LibrarySwitchService
Modality	Authorization
Subject	n: nos
SDomain	PrivateArea
Target	s: Layer2Switch
TDomain	d: PrivateArea
Action	Use
Constraint	d.authenticated(n)

Diese zielorientierte Policy muß nun in operationale Policies überführt werden. Wie schon in Abschnitt 3.5 festgestellt wurde, wird die Modalität in einer operationalen Policy nicht mehr ausdrücklich angeführt, sondern muß durch entsprechende Managementanweisungen ausgedrückt werden. Dies kann u.U. schwierig ausfallen und erst über Umwege erreicht werden, so daß oft im nachhinein die Intention einer operationalen Policy nicht mehr nachvollziehbar ist. In einem derartigen Fall bieten die Eltern-Policies der höheren Hierarchiestufen eine Hilfestellung, so daß deren Speicherung und Verbindung zu den operationalen Policies im normalen Betrieb nicht vernachlässigbar ist. Im folgenden wird angenommen, daß die Ports eines Switchs im privaten Bereich grundsätzlich abgeschalten sind und nur durch eine explizite Anweisung geöffnet werden. Weiterhin wird davon ausgegangen, daß bereits ein VLAN besteht, indem alle zugänglichen Ressourcen miteinander verbunden sind.

Es ergibt sich folgende *operationale Policy*:

```
POLICY SwitchCreateVLAN
FOR /libraries/ ON <NS: /libraries/our_library/private/sw1.pc_switch >
DELIVERS NewNodeAttached {string port; string NewNodeHardwareAddress}
ACTION { openPort(port);
        addToVLAN(/libraries/our_library/mail1.mail_hub/.get_VLAN_ID(),
                NewNodeAttached:port); }
CONSTRAINT (Subject.getState() = authentified)
```

Beispiel 2

Ebenfalls läßt sich aus dem Bibliotheksszenario folgende strategische Policy formulieren:

> *Nur authentifizierten Systemen wird die Nutzung des Druckers erlaubt. Bei Nutzung des Druckers im privatem Bereich, muß eine Abrechungsmöglichkeit mit der Heimatdomäne existieren. Bei Nutzung des öffentlichen Druckers, darf die Druckerquota nicht überschritten sein.*

Daraus ergeben sich die folgenden strategischen Policies:

Name	PublicPrinter
Modality	Permission
Subject	n: nos
SDomain	VisitorDomain
Target	p : printer
TDomain	d: VisitorDomain
Action	Print
Constraint	d.authenticated(n) && p.quota(n) > 0

Name	PrivatePrinter
Modality	Permission
Subject	n: nos
SDomain	*sd: abroad*
Target	p: printer
TDomain	td:PrivateDomain
Action	Print
Constraint	td.authenticated(n) && td.accountExistsFor(sd)

Bei näherer Betrachtung dieser zielorientierten Policies kann festgestellt werden, daß die *Permission*-Policies auch als *Prohibition*-Policies formuliert werden können. Es hat sich aber bei der hier festgelegten PDL herausgestellt, daß sich der Vorgang der Überführung in operationale Policies von einer *Permission-Policy* aus einfacher gestaltet. Eine *Meta-Policy* für zielorientierte Policies könnte also lauten:

> *Ziehe die Formulierung einer Permission immer der einer Prohibition vor und versuche statt dessen die Negation durch Constraints auszudrücken.*

Die zielorientierten Policies ergeben die folgenden operationalen Policies:

```
POLICY CreateVLANPublicPrinter
FOR /libraries/our_library/public/
ON <NS: /libraries/our_library/public/sw1.pc_switch >
DELIVERS NewNodeAttached {string port; string NewNodeHardwareAddress}
ACTION { openPort(port);
         addToVLAN(/our_library/public/hpIIIsi.printer/.get_VLAN_ID(),
                   NewNodeAttached:port); }
CONSTRAINT ( (Subject.getState() = authentified) &&
 (/our_library/public/hpIIIsi.printer/.getQuotaFor (Subject.getID()) > 0) )
```

```
POLICY CreateVLANPrivatePrinter
FOR /libraries/ ON <NS: /our_library/private/sw1.pc_switch >
DELIVERS NewNodeAttached {string port; string NewNodeHardwareAddress}
ACTION { openPort(port);
        addToVLAN(/libraries/our_library/private/hpIIIsi.printer/.get_VLAN_ID(),
                  NewNodeAttached:port); }
CONSTRAINT ( (Subject.getState() = authentified) &&
 (/libraries/our_library/accounting/.accountExistsFor(Subject.getDomain() ) ) )
```

Wie aus diesen Beispielpolicies zu ersehen ist, wurde die Überführung der strategischen Policies in die operationalen Policies nicht dadurch erreicht, daß Operationen an der Managementschnittstelle des Druckers bzw. der Druckerwarteschlange aufgerufen wurden, sondern das entsprechende NoS dem VLAN des Druckers zugewiesen wurde. Das bedeutet, daß in einer zielorientierten Policy i.d.R. zunächst die unmittelbaren Zielobjekte spezifiziert werden, diese aber nicht unbedingt in die daraus resultierenden operationalen Policies übernommen werden müssen. In der operationalen Policy entscheidet der Policy-Ersteller meist intuitiv nach den tatsächlich in dem Netz vorliegenden Fakten. Im konkreten Fall erweist sich dieser Weg als der bessere, da gleich zu Beginn, beim Anschluß des Endgeräts an den Switch, entschieden wird, ob es den Drucker nutzen darf oder nicht. Es erfolgt also nur eine einmalige Überprüfung des Constraints, der, wäre die Policy mit der Druckerwarteschlange als Zielobjekt formuliert worden, im Gegensatz dazu, jedesmal ausgewertet werden müßte, wenn ein neuer Druckauftrag hinzukommt.

Dieser Ansatz läßt sich prinzipiell uneingeschränkt auf alle Ressourcen verallgemeinern. Wird angenommen, daß jede Ressource ihre eigene VLAN-ID besitzt und Endsysteme mehreren VLANs angehören können, so kann eine neue *Meta-Policy* zur Erstellung operationaler Policies formuliert werden:

> *Soll die Nutzung einer Ressource ganzheitlich erlaubt werden, so sollte das NoS dem VLAN der Ressource, falls möglich, zugeordnet werden. Nur falls die Funktionalität einer Ressource je nach Subjekt eingeschränkt werden soll, ist ein direkter Eingriff über die Managementschnittstelle der Ressource nötig.*

Wie aus diesen zwei Beispielen gefolgert werden kann, erleichtern *Meta-Policies* die Erstellung von Policies. Eine Meta-Policy ist, ganz allgemein gesehen, eine Regel, wie mit einer Policy umgegangen werden sollte. Im vorliegenden Fall beinhalten die Meta-Policies Hilfestellungen und Regeln für den Policy-Entwicklungsprozeß. Diese Meta-Policies ergeben sich meist aus der Erfahrung der Policy-Ersteller. Die Weiterverfolgung dieses Aspekts würde jedoch den Rahmen dieser Diplomarbeit sprengen und sollte demzufolge in nachfolgenden Arbeiten als Ansatz dienen. Am Lehrstuhl wird bereits in einer Diplomarbeit [Avit98] der Einsatz von Meta-Policies zur Verwaltung von Policies untersucht.

3.8 Zusammenfassung

In diesem Kapitel wurden zunächst die Grundbegriffe des Policy-basierten Netz- und Systemmanagements dargestellt, um darauf aufbauend Anpassungen in Hinblick auf das Konfigurationsmanagement von nomadischen Systemen durchzuführen. Dies äußerte sich u.a. in der Einführung neuer syntaktischer Vorgaben für die Templates der zielorientierten Policies und v.a. in der Entwicklung einer neuen Notation für die Beschreibung maschineninterpretierbarer operationaler Policies im Rahmen der *Policy Description Language*.

Bei der Betrachtung des Policy Entwicklungsprozesses, wurde ebenfalls die Problematik des Refinements von Policies deutlich. Vor allem die Aufdeckung und Lösung von Konflikten stellt eine nicht triviale Aufgabe an den Policy Service und den Policy-Ersteller. Es wurden aber bereits erste, der hier vorgestellten PDL angepaßten Strategien zur Entdeckung und Lösung von Konflikten vorgestellt. Hier bedarf es aber durchaus weiterer Untersuchungen, um eine Verbesserung der hier beschriebenen Ansätze zu erreichen sowie neue zu entwickeln.

Anhand des Bibliotheksszenarios wurden einige Beispielpolicies entwickelt, um die Vorgehensweise bei der Formulierung der Policies nochmals zu unterstreichen. Es wurde deutlich, daß mehrere Möglichkeiten existieren, um *High-Level-Policies* auf operationale Policies abzubilden. Einerseits wurde dadurch die Relevanz der Formulierung und Speicherung dieser High-Level-Policies begründet, um zu gewährleisten, daß die Intention der operationalen Policies zu jedem Zeitpunkt von einem Systemadministrator nachvollziehbar bleibt. Andererseits zeigt dies auch die Notwendigkeit der Formulierung von Meta-Policies, um dem Policy-Ersteller die Kreierung der Policies zu erleichtern. Dieser Aspekt konnte aufgrund der Zielsetzung dieser Arbeit nicht weiter vertieft werden, so daß dies in nachfolgenden Arbeiten weiterverfolgt werden sollte.

Kapitel 4

Die Managementarchitektur

In diesem Kapitel wird die entwickelte Managementarchitektur (MA) vorgestellt. Zunächst werden die Anforderungen an die MA formuliert, um anschließend deren konkrete Umsetzung in einer CORBA-Umgebung aufzuzeigen. Zudem wird die prototypische Implementierung der Komponenten der Gesamtarchitektur des Managementsystems (MS) dargestellt und ihre Funktionsweise erklärt. Es werden ebenfalls die Anforderungen an die Schnittstelle eines Managementagenten zusammengestellt, um eine Zusammenarbeit mit dem entwickelten Managementsystem zu ermöglichen. Abschließend wird ein typischer Managementablauf zwischen NoS und MS anhand von Diagrammen veranschaulicht.

4.1 Anforderungen

In der nun folgenden Anforderungsanalyse der Managementarchitektur, gilt es erste Anhaltspunkte für deren Realisierung festzustellen. Die MA sollte in der Weise konzipiert sein, daß die nachstehenden Forderungen erfüllt sind:

- das Managementsystem ist von der zugrundeliegenden Infrastruktur so unabhängig wie möglich
- das MS ist skalierbar
- der Managementbereich ist durch einfaches Hinzufügen neuer Komponenten erweiterbar
- die Funktionalität des MS läßt sich ebenfalls ohne weiteres durch neue Komponenten erweitern

Die MA sollte demzufolge modular wie ein Baukasten beschaffen sein, in der sich die einzelnen Bestandteile zusammenfügen lassen, um das Gesamtsystem zu erweitern und aufzubauen. Diese Forderungen lassen sich grundsätzlich dann durchsetzen, wenn die MA in die folgenden drei Basisbestandteile getrennt wird, die ebenfalls in Abbildung 4-1 veranschaulicht sind: die *Managementanwendung*, die *Managementagenten* und die *Managementobjekte (Managed Objects, MO)*.

Die Managementanwendung, die ein Policy-basiertes Management durch eine entsprechende Implementierung ermöglicht, agiert auf den Managementagenten durch Verwendung einer einheitlichen Kommunikationsschnittstelle, die von allen Managementagenten innerhalb des MS unterstützt wird. Die Agenten wiederum agieren auf den MOs über diejenige Schnittstelle, die diese unterstützen. Als MOs werden nicht nur wie erwartet die nomadischen Systeme angesehen, sondern insbesondere auch die zu managende Infrastruktur, deren Komponenten in Kapitel 1.3 als *Netzressourcen* bezeichnet wurden. Über die Beeinflussung der Netzressourcen wird das gewünschte Ziel der Steuerung des Verhaltens der nomadischen Systeme erreicht.

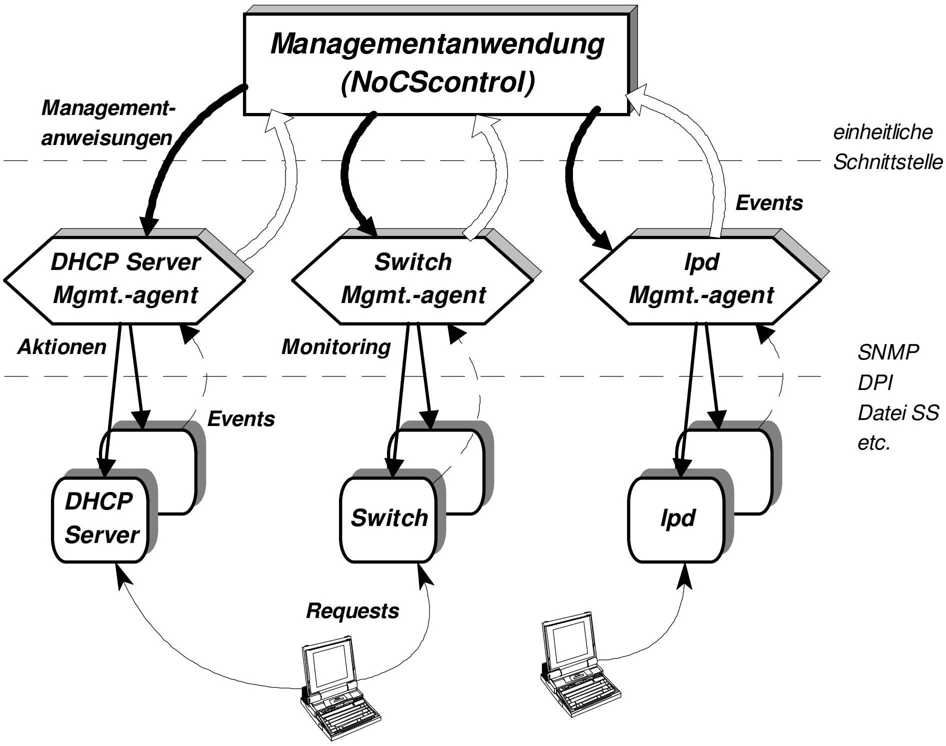

Abbildung 4-1: Die Komponenten der MA

Die Managementanwendung bildet zusammen mit den verschiedenen Managementagenten den tatsächlichen *Manager* der MOs. Die Managementagenten nehmen im Gesamtsystem eine bivalente Rolle ein. Aus Sicht der übergeordneten Managementanwendung (hier mit *NoCScontrol* bezeichnet) agieren die Agenten in einer eher passiven Rolle, da diese Anweisungen entgegennehmen und Events in Richtung der Managementanwendung versenden. Aus dem Blickwinkel der MOs agieren die Managementagenten jedoch in der Rolle des Managers, da in diesem Fall Managementaktionen in Richtung der MOs versendet werden und Events, sei es in Form von SNMP-traps oder über eine Dateischnittstelle, von dieser Seite erhalten werden. Die zu modellierende Managementanwendung agiert also nicht direkt auf den MOs, sondern über den Umweg der Managementagenten, die hier demzufolge als eine Art *Proxy* fungieren. Die Kommunikation zwischen Managementagent und MO fällt, abhängig von dem unterstützten Managementinterface der zu managenden Netzressource, natürlich unterschiedlich aus. Ideal wäre eine integrierte, echte Managementschnittstelle der Ressource z.B. in Form eines SNMP-Interfaces. Der Umweg über die Agenten entfällt, wenn bereits das MO die einheitliche Schnittstelle der Managementanwendung unterstützt.

Durch die Wahl die MA in dieser Form aufzuteilen, sind einige der am Anfang aufgestellten Forderungen bereits erfüllt. Jeder Agent, wie z.B. ein Managementagent eines DHCP-Servers, agiert für sich gesehen selbständig auf den für ihn bestimmten MOs und ist für einen zu managenden Teilbereich zuständig und ausgerichtet. Auf diese Weise besteht sowohl die Möglichkeit bereits existierende Agenten in die MA miteinzugliedern, als auch später hinzukommende Managementbereiche, die derzeit noch nicht berücksichtigt sind, in die MA mit einzubeziehen, indem die dafür notwendigen Agenten entwickelt und implementiert werden. Damit ist die Forderung nach einem möglichst einfachen Weg den zu managenden Bereich zu erweitern erfüllt, ebenso wie die Forderung nach Skalierbarkeit und nach einer gewissen Unabhängigkeit von der in einem Netz tatsächlich etablierten Infrastruktur.

Es bleibt noch die Forderung nach einer einfachen Erweiterung der *Funktionalität* des MS übrig. Um dies verwirklichen zu können, müßte die Managementanwendung in der Weise umgesetzt sein, daß bei einer tatsächlichen Erweiterung (oder auch Änderung) nicht die ganze Anwendung, wie z.B. durch ein erneutes Übersetzen, davon betroffen ist. Dies läßt sich nur dadurch realisieren, daß die Anwendung selber wieder aus einzelnen Komponenten aufgebaut ist, die eine bestimmte Teilfunktionalität der Anwendung verwirklichen und diese wieder an einer Schnittstelle den anderen Komponenten anbieten. Um die verschiedenen Komponenten der Anwendung bestimmen zu können, muß folglich zunächst eine Analyse der Funktionalität der Anwendung durchgeführt werden, um anschließend Teilfunktionen nach Kriterien wie Lokalität, Kommunikationsbedarf, etc. zusammenlegen zu können.

Aus Kapitel 3 lassen sich die folgenden Teilfunktionen der Anwendung angeben, um ein Policy-basiertes Management von nomadischen Systemen zu ermöglichen:

- Kreieren, Löschen und persistentes Anlegen von Policies
- Refinement von Policies, das auch die Lösung von Konflikten beinhaltet
- Durchsetzen einer aktivierten operationalen Policy, indem das Zielobjekt beobachtet wird und bei Eintreten eines Events die spezifizierten Managementaktionen gefeuert werden
- Strukturierung eines Netzes durch Domänenbildung

Nach dieser ersten Identifizierung der Teilfunktionen der Anwendung, läßt sich bereits erkennen, welche dieser Teilfunktionen jeweils zusammen in einer Komponente realisiert werden sollten, um v.a. auch die Managementanwendung mit der Forderung nach Skalierbarkeit verwirklichen zu können. Die einzelnen Komponenten erbringen den jeweils anderen Komponenten an einer wohldefinierten, öffentlichen Schnittstelle einen *Dienst*. Diese Dienste sind in Abbildung 4-2 veranschaulicht und werden nun im folgenden erklärt:

Policy Service

Der *Policy Service* bietet dem Policy-Ersteller die Möglichkeit, strategische, zielorientierte und operationale Policies zu verfassen und persistent anzulegen. Die Formulierung der Policies soll durch Computerunterstützung erleichtert werden, indem u.a. Konflikte bereits während der Policy-Erzeugung angezeigt werden und, wenn möglich, eine Lösung des Konflikts vorgeschlagen wird. Weiterhin dient der Policy Service zur Aktivierung und Deaktivierung von operationalen Policies. Bei der Aktivierung einer operationalen Policy, wird mit Hilfe eines *Policy Interpreters* die Durchsetzung der in PDL formulierten Policy erreicht. Ebenso bietet der Policy Service die Möglichkeit, persistente Policies wieder zu löschen. Bei dieser Operation muß darauf Acht gegeben werden, daß die entsprechenden abgeleiteten Policies ebenfalls gelöscht und deaktiviert werden.

Domain Service

Um den Gültigkeitsbereich von Policies angeben zu können sowie die Netzressourcen und NoS geeignet gruppieren und aufteilen zu können, wird ein Domain Service benötigt. Neben der Strukturierung eines Netzes durch *Domänenbildung* und der entsprechenden Zuweisung von Netzkomponenten, liegt die Hauptaufgabe des *Domain Services* in der *Beantwortung* und Auflösung von *Domänenanfragen* seitens der anderen Services. Zwei typische Anfragen sind die der Feststellung der Domäne eines MOs sowie die Rückgabe aller MOs, die einer bestimmten Domäne zugeteilt sind. Der hier benötigte Domain Service unterscheidet sich gegenüber den Domain Services, die in den MAs für nicht nomadische verteilte Systeme entwickelt worden sind, in einem wesentlichen Punkt: es müssen zusätzlich Anfragen ermöglicht werden, die die *Heimatdomäne von NoS* zurückliefern und deren gegenwärtigen Aufenthaltsort feststellen. Es liegt auf der Hand, daß dieser Aspekt der Anfragen für sich nicht bewegende verteilte Systeme irrelevant ist.

Monitoring Service

Der *Monitoring Service* bietet die Möglichkeit der Überwachung der in einer Policy angegebenen *Target Objects*. Tritt eines der in der Policy angegebenen Events ein, so wird der *Enforcement Service* benachrichtigt, der sich um die Durchsetzung der entsprechenden Policy kümmert.

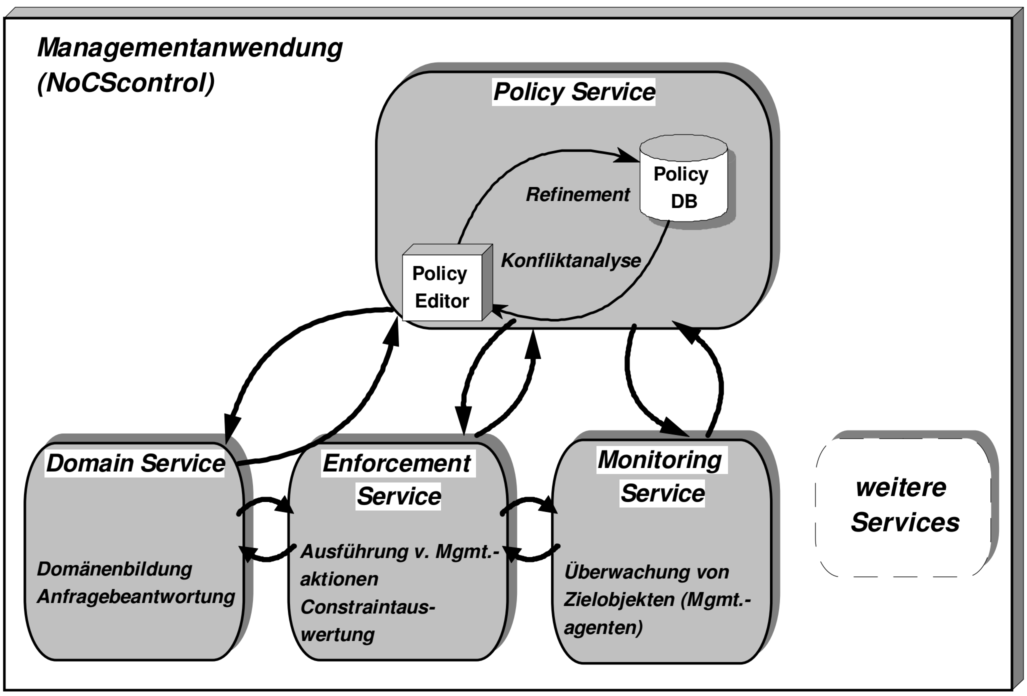

Abbildung 4-2: Die Komponenten der Managementanwendung

Enforcement Service

Mit der Aktivierung einer operationalen Policy werden die Dienste des *Enforcement Services* in Anspruch genommen, der sich daraufhin um die tatsächliche Durchsetzung der Policy kümmert. Zunächst abonniert sich der Enforcement Service beim Monitoring Service für die relevanten, die Policy betreffenden Events. Bei Eintreten eines Events wird, nach Prüfung des in der Policy angegebenen Constraints, die Managementaktionen gefeuert und damit die Policy tatsächlich durchgesetzt.

Die Umsetzung der hier vorgestellten ersten Skizze der MA in eine konkrete Prototypimplementierung wird in den nun folgenden Kapiteln gezeigt.

4.2 Umsetzung in einer CORBA-Umgebung

Die in Unterkapitel 4.1 skizzierten Anforderungen an die MA zeigten an zwei Stellen die Notwendigkeit der Verteilung von Objekten in einem Netz:

- die *Managementagenten* sind, abhängig von Einsatzort und Funktionalität, verteilt über das ganze Netz aufzufinden
- die einzelnen *Komponenten der Managementanwendung* sollen aus Gründen der Skalierbarkeit und der Ausfallsicherheit, ebenfalls so unabhängig wie möglich innerhalb einer verteilten Anwendung agieren

Um die Kommunikation zwischen den einzelnen Bestandteilen der MA in einer heterogenen und vernetzten Systemumgebung zu ermöglichen, bietet sich der Einsatz einer *Middleware* an. Zu den Aufgaben einer Middleware gehört es, eine Kommunikationsgrundlage zu etablieren, die die Verteilung der Objekte in einem Netz transparent erscheinen läßt. Aus Sicht eines Objekts gestalten sich die Schritte für den Zugriff auf die Schnittstelle anderer Objekte immer gleich und vollkommen unabhängig von der Tatsache, an welchem Ort die Objekte wahrhaftig instanziiert worden sind. Dadurch bietet sich der Vorteil, sich bei der Implementierung der Objekte auf das wesentliche, nämlich auf die nach außen hin anzubietenden Schnittstellen, beschränken zu können.

Derzeit werden mehrere leistungsfähige Middleware-Konzepte angeboten, wobei CORBA einerseits durch die Standardisierung des Rahmenwerks und aller hinzukommenden Erweiterungen und andererseits durch die nun zu erwartende hohe Verbreitung durch die Integration in das neue *Java Development Kit 1.2 (JDK1.2)* von *Sun* [Sun98] hervorsticht. CORBA bietet auch den entscheidenden Vorteil konkret auf eine heterogene Systemumgebung ausgerichtet zu sein. Auch existieren seit mehr als fünf Jahren CORBA-Implementierungen von mehreren Herstellern, so daß inzwischen Erfahrungen gesammelt und Fehler beseitigt werden konnten. Für die Implementierung des Prototypen wurde die CORBA-Implementierung *Visibroker 3.0 for Java* von *Visigenic* [Visi97] verwendet. Auch in diesem Fall wird durch den Verkauf von Visigenic an *Borland/Inprise* und deren Integration des Visibrokers in den neuen *Borland/Inprise Jbuilder 2*, eine Zunahme der Verbreitung dieser CORBA-Implementierung erhofft. Im folgenden werden die Grundlagen von CORBA erklärt, um anschließend darauf aufbauend die Bestandteile, sowie deren Funktionsweise, des Prototypen erklären zu können. Für weiterführende Informationen über CORBA wird auf die einschlägige Literatur [OHE96] verwiesen.

4.2.1 Einführung in CORBA

Durch den Einzug der *Objekt-Orientierung* (OO) in die Programmierung und Softwareentwicklung, ergaben sich die folgenden Vorteile:

Datenkapselung
Die Datenstruktur eines Objekts sowie dessen Realisierung, wird eindeutig festgelegt und von der Datenstruktur anderer Objekte abgekapselt.

Datenverbergung
Die Kommunikation mit einem Objekt kann jeweils nur über die definierten, durch Objektmethoden und -attribute zur Verfügung gestellten Schnittstellen geschehen.

Vererbung
Durch die Möglichkeit der Vererbung von Eigenschaften eines Objekts an andere Objekte, ist die Möglichkeit der Wiederverwendung bereits existierenden Codes gegeben.

Verteilte Objekte sind nun das entsprechende Komplement im Bereich der verteilten Systeme, mit dem Zusatz, daß sie selbständige, unabhängige Einheiten darstellen, die dadurch nicht nur im Rahmen eines Programmes sichtbar sind [OHE 96]. Verteilte Objekte können von jedem Punkt innerhalb eines Netzwerks angesprochen werden und damit ihre Funktionalität über die Grenzen des Systems auf dem sie installiert sind, anbieten. Um dies verwirklichen zu können, wird ein Rahmenwerk benötigt, der die Einzelheiten wie Kommunikationsprotokolle, erlaubte Datenstrukturen, etc. festlegt.

Die *Object Management Group* (OMG) ist ein Konsortium bestehend aus über 800 Mitgliedern aus der IT-Branche, die sich die primäre Aufgabe gesetzt hat, einen offenen, nicht-proprietären *Bus* zu spezifizieren, der das Konzept der in einem Netzwerk verteilten Objekte ermöglicht. Der dabei entwickelte Standard wird als *Common Object Request Broker Architecture* (CORBA) bezeichnet. Die Beschreibung der zu Grunde liegenden Architektur [OMG97a] wurde erstmals 1990 im *Object Management Architecture Guide* (OMA Guide) veröffentlicht.

In Abbildung 4-3 ist die Architektur von CORBA aufgezeichnet, die aus den folgenden vier Hauptelementen besteht:

Object Request Broker (ORB)
Der ORB ist das Kernstück von CORBA und definiert den oben schon erwähnten Bus, der die Kommunikation zwischen verteilten Objekten ermöglicht.

Common Object Services
Die Common Object Services definieren systemnahe Dienste, die die Funktionalität des ORBs erweitern.

Common Facilities
Mit den Common Facilities werden *Application Objects* mit einer speziellen Funktionalität definiert, die direkt von anderen Application Objects genutzt werden können.

Application Objects
Die Application Objects sind die tatsächlichen Nutznießer des ORBs und seiner zur Verfügung stehenden Dienste.

CORBA ist, um eines ihrer Hauptmerkmale zu beschreiben, eine *Client-Server-Architektur*. Zusammengefaßt ist das mit CORBA verfolgte Ziel, die Zusammenarbeit von verteilten Systemen zu ermöglichen, indem die Unabhängigkeit von Programmiersprachen und Betriebssystemen gewahrt und eine Ortstransparenz von Client und Server erlangt wird.

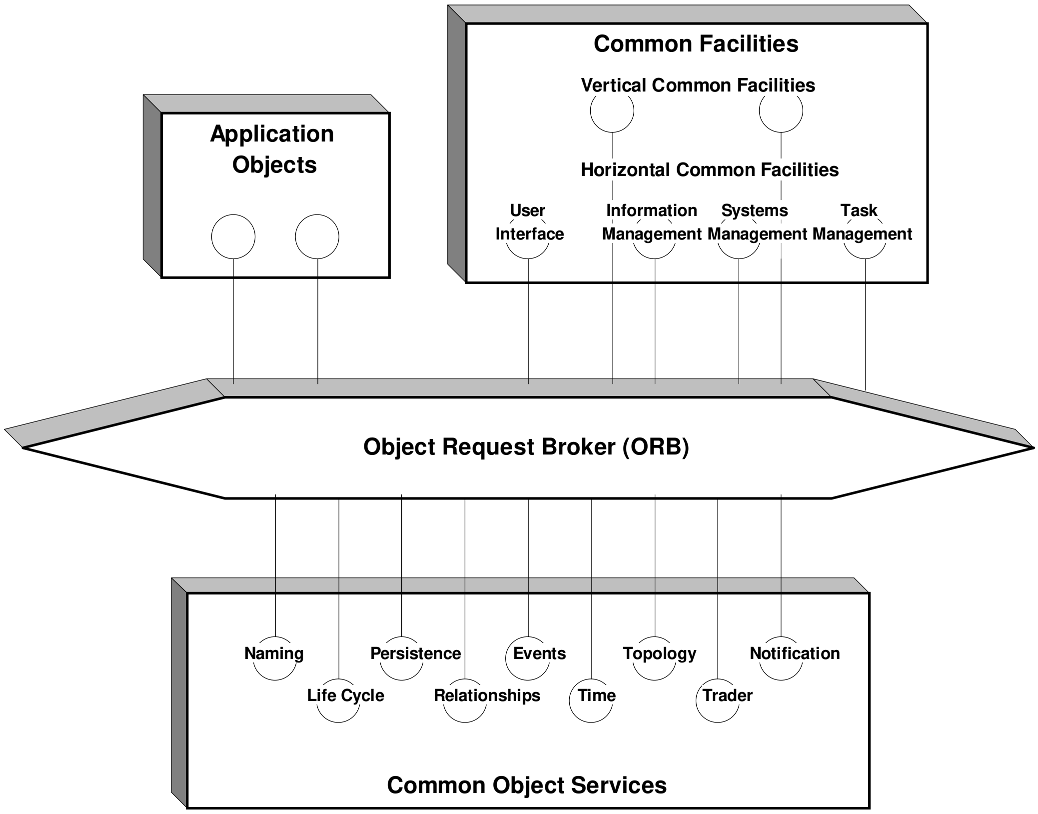

Abbildung 4-3: Die OMG OMA

4.2.2 Der CORBA 2.0 ORB

Aus Abbildung 4-4 ist die Struktur des ORBs sowie dessen Zusammenspiel mit einem Client und einem Server-Objekt zu ersehen. Die Grundvoraussetzung, die einen Datenaustausch von Client und Server erst ermöglicht, ist die Spezifikation der Schnittstelle des Servers. Alle Server-Schnittstellen werden in der programmiersprachenunabhängigen *IDL (Interface Definition Language)* spezifiziert. Mit einem Pre-Compiler, wie z.B. *idl2java*, ist es möglich, die in IDL spezifizierten Schnittstellen in entsprechende Interfaces der eingesetzten Programmiersprache zu übersetzen.

Auf *Client-Seite* sind die folgenden CORBA-Bestandteile zu finden:

Client IDL Stubs

Von der Client-Seite aus gesehen, agieren die Client IDL Stubs wie ein lokaler *Proxy* für die Schnittstelle des Server-Objekts. Dies bedeutet, daß auf Client-Seite die entsprechende Operation des Server-Objekts wie ein lokaler Methodenaufruf in den eigenen Code eingefügt wird. Die Client IDL Stubs werden zusammen mit dem

Client-Code übersetzt und enthalten die notwendigen Kodierungs- und Dekodierungsfunktionen für den entfernten Operationsaufruf. Um in der CORBA-Terminologie zu bleiben, wird in diesem Fall von einem *statischen Binden* an ein Server-Objekt gesprochen, da schon während bzw. vor dem Übersetzungsvorgang bekannt ist, welche Operationen auf der Server-Seite aufgerufen werden.

Dynamic Invocation Interface (DII)
Mit Hilfe dieser API ist es möglich, während der Laufzeit eines Client-Objekts Informationen über die Schnittstelle eines Server-Objekts zu gewinnen, um im nächsten Schritt entsprechende Operationen des Servers aufzurufen. In diesem Fall wird vom *dynamischen Binden* an ein Server-Objekt gesprochen.

Interface Repository (IR)
Die Interface Repository enthält die Beschreibung aller Schnittstellen von Server-Objekten in IDL. Mit Hilfe der IR ist es z.B. möglich, an die relevanten Informationen zu gelangen, die ein dynamisches Binden an ein Server-Objekt bewerkstelligen.

ORB Interface
Das ORB Interface bietet den Objekten einige nützliche APIs an, wie z.B. eine Objektreferenz in einen String umzuwandeln und umgekehrt.

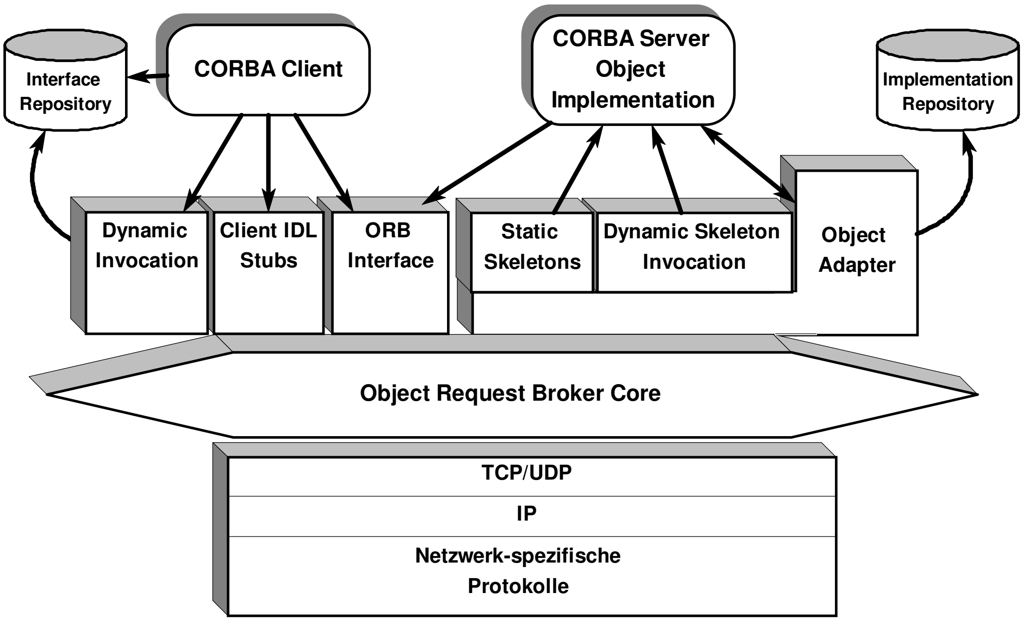

Abbildung 4-4: Die Struktur des CORBA 2.0 ORBs

Auf *Server-Seite* sind die folgenden Komponenten vorzufinden:

Static Skeletons
Static Skeletons sind *Server IDL Stubs*, die die Schnittstellen des Server-Objekts statisch zur Verfügung stellen. Wird ein entfernter Operationsaufruf durch den *Object Adapter* registriert, so wird der entsprechende Aufruf durch diese IDL Stubs an das Server-Objekt weitergereicht, der die tatsächliche Implementierung der Operation enthält.

Dynamic Skeleton Interface (DSI)

Das Dynamic Skeleton Interface bietet die Möglichkeit zur Laufzeit erhaltene Methodenaufrufe an Server-Objekte weiterzureichen, die *keine* IDL-basierten Skeletons zur Verfügung stellen. Es wird versucht anhand der Parameterwerte das richtige Server-Objekt sowie die entsprechende Operation auszuwählen und aufzurufen. Das DSI ist v.a. in Hinblick auf die Implementierung von Inter-ORB-Brücken relevant.

Object Adapter (OA)

Der Object Adapter ist der eigentliche Empfänger von Client-Requests, die einen Zugriff auf das Interface eines Server-Objekts enthalten. Die Requests werden an das entsprechende Server-Objekt weitergeleitet. Der OA spielt dabei die Rolle eines Managers, indem es sich um die Objektreferenz des Servers, den Einträgen in die *Implementation Repository* und v.a. um das scheduling von eintreffenden Requests kümmert. Um die Implementierung von Server-Objekten zu vereinfachen, ist der *Basic Object Adapter (BOA)* standardmäßig in jedem CORBA ORB enthalten, der jedes CORBA Server-Objekt unterstützt. So beschränkt sich die eigene Implementierung eines OAs für ein Server-Objekt auf einzelne Spezialfälle, wie z.B. wenn von vornherein mit einer hohen Anzahl von gleichzeitig eintretenden Requests gerechnet wird und deswegen eine spezielle Prioritätenstrategie unterstützt werden soll.

Implementation Repository

Die Implementation Repository ist eine Laufzeitdatenbank die u.a. die derzeit auf dem ORB erreichbaren Server-Objekte enthält. Sie wird auch dazu genutzt, zusätzliche Informationen über die tatsächliche ORB-Implementierung und dessen CORBA-services-Unterstützung zu halten.

4.2.3 Die CORBA Services

Wie aus Abbildung 4-3 zu ersehen ist, sind eine Vielzahl von verschiedenen *Common Object Services* (kurz: CORBA Services) [OMG97b] spezifiziert worden, die jeweils teilnehmenden Objekten eine *Grundfunktionalität* anbieten. Daraus resultiert meist eine erleichterte Handhabung mit anderen Objekten, wobei sich dies auch durch unterstützende Maßnahmen bei der Objektimplementierung auswirken kann. Im folgenden werden nur diejenigen Spezifizierungen von CORBA Services näher betrachtet, die in Zusammenhang mit dem zu entwickelten Managementsystem als nützlich erscheinen. Dazu zählen der *Naming Service*, der *Topology Service*, der *Event Service* und der *Notification Service*.

4.2.3.1 Der Naming Service

Die primäre Funktion des *Naming Services* ([OMG97b], S.3-1 ff) ist die eindeutige Zuweisung von Namen an CORBA-Objekte, um so eine eindeutige Identifizierung dieser Objekte allein durch die Angabe des Namens zu ermöglichen.

Folgende Bezeichnungen und Beziehungen werden innerhalb der Spezifikation festgelegt (siehe hierzu auch Abbildung 4-5):

name binding

Als *name binding* wird die Zuordnung eines Namens zu einem Objekt bezeichnet. Diese Zuweisung erfolgt immer relativ zu einem *naming context*.

naming context
Ein *naming context* ist ein Objekt, das eine Menge von *name bindings* enthält, deren Namen innerhalb des *naming contexts* eindeutig sein müssen.

Names
Eine *Name* ist eine geordnete Liste von *components.* Besteht ein Name nur aus einem *component*, so wird er als *simple name* bezeichnet, ansonsten als *compound name*. Um die Struktur eines *Names* zu verdeutlichen, wird folgende Notation verwendet:

```
< component 1 ; component 2 ; component 3 >
```

Jeweils der letzte *component* eines *Names* ist an ein Objekt gebunden und wird als Name des Objekts gewertet. Die vorhergehenden *components* werden zur Identifizierung von *naming contexts* verwendet.

(name) component
Das Basisgebilde des Naming Services ist der *name component*, der aus zwei Attributen besteht: das *identifier attribute* und das *kind attribute*. Beide Attribute werden als Strings dargestellt. Der *identifier* wird syntaktisch, wie der Name schon sagt, zur Identifizierung von Objekten eingesetzt, während das *kind* Attribut als Beschreibung des *identifiers* oder des damit assoziierten Objekts benützt werden kann. Als Belegung für das *kind* Attribut wären z.B. *executable*, *postscript*, etc. denkbar.

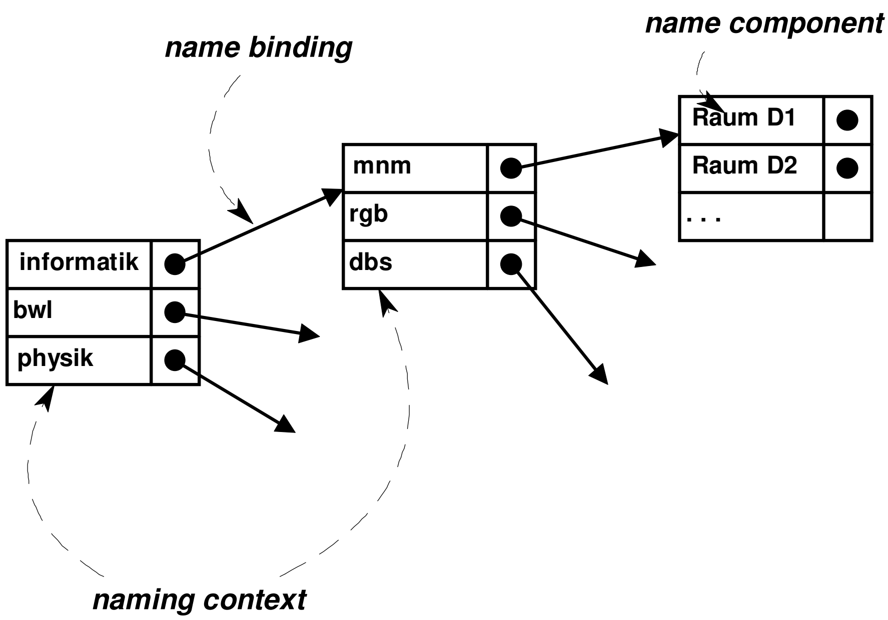

Abbildung 4-5: Bestandteile des Naming Service

Da ein *naming context* ein Objekt ist, kann es selber innerhalb eines anderen *naming contexts* an einen Namen gebunden werden. Dadurch ist es möglich, einen *naming graph* aufzubauen (siehe hierzu auch Abbildung 4-6), der dazu genutzt werden kann, ein Netz und dessen Komponenten zu strukturieren. Damit bietet sich der *Naming Service* dafür an, als *Domain Service* der in Kapitel 4.1 beschriebenen Managementarchitektur eingesetzt zu werden.

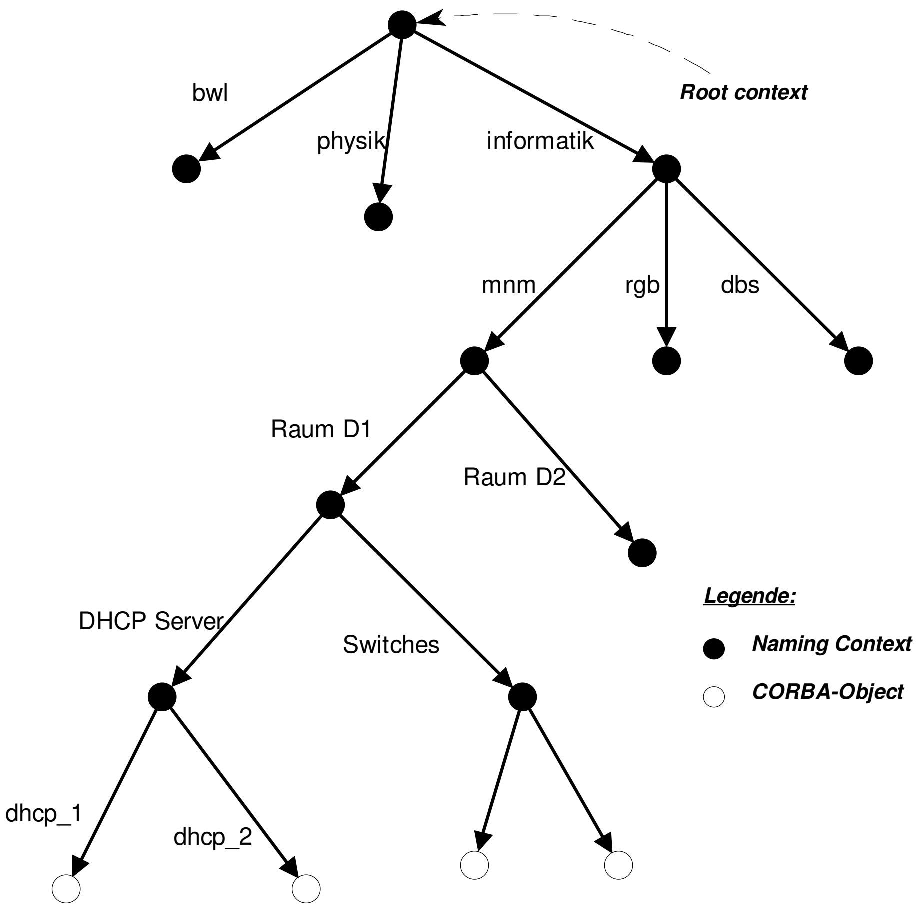

Abbildung 4-6: Beispiel für einen naming graph

Der Naming Service bietet grundsätzlich vier verschiedene Operationen an: *bind*, *resolve, list* und *destroy*. Mit *bind* ist es möglich, Objekte an einen Namen relativ zu einem *naming context* zu binden. Der Operation *resolve* wird als Parameter ein *compound name* übergeben, woraufhin versucht wird, das entsprechende, zu diesem Namen gehörende Objekt zurückzugeben. Mit *destroy* werden *name bindings* wieder aufgelöst und *list* bietet die Möglichkeit, sich innerhalb eines *naming context* alle *name components* zurückgeben zu lassen.

Der Naming Service ist bewußt einfach gehalten, um diesen für alle möglichen Einsatzszenarien offen zu halten. Hier liegt aber auch der große Nachteil des Naming Services. Es ist z.T. sehr umständlich Objektanfragen in Form von *compound names* zu formulieren. Ebenso gestaltet sich das Suchen von Objekten in einem *naming graph* sehr schwerfällig, da dies nur durch wiederholte Anwendung der Operation list möglich ist. Ein weiterer Nachteil ist, daß es nicht möglich ist bestimmte Regeln an das *name binding* auszudrücken, um nur bestimmte Beziehungen zu erlauben, wie z.B. daß nur dhcp-MOs an den DHCPServer-Ast in Abbildung 4-5 angehängt werden dürfen. Wie im nächsten Kapitel zu sehen sein wird, werden vom *Topology Service* diese speziellen Anforderungen erfüllt. Der Vorteil des Naming Services liegt darin, daß aufgrund seiner Einfachheit bereits Implementierungen dieser Spezifikation, u.a. von *Visigenic*, existieren.

4.2.3.2 Der Topology Service

Um mit CORBA auch das *Netz- und Systemmanagement* zu erfassen, wurde der *Topology Service* spezifiziert, dessen Funktionalität es ermöglicht, topologische Beziehungen zwischen Objekten zu definieren. Der Topology Service versetzt CORBA-Clients in die Lage, in einem Netzwerk miteinander verbundene verteilte Systeme zu modellieren sowie ihre Abhängigkeiten untereinander darzustellen, und damit die für das Netz- und Systemmanagement relevanten Informationen aufzubereiten.

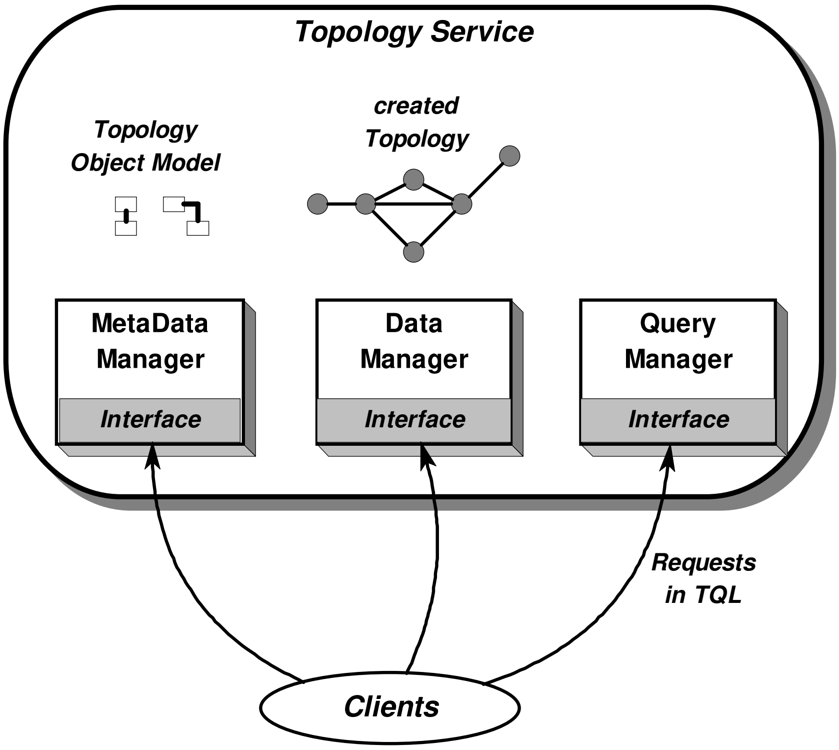

Abbildung 4-7: Die Architektur des Topology Service

Im RFP des Topology Service [HePa97] wird die in Abbildung 4-7 gezeigte Architektur spezifiziert, die aus den folgenden Bestandteilen zusammengesetzt ist:

MetaData Manager

Der *Topology MetaData Manager* verwaltet die Regeln und die Semantik einer Topologie. Über das *MetaData Interface* wird Client-Objekten die Möglichkeit gegeben Regeln aufzustellen, die die Art der Beziehungen zwischen den verschiedenen Topology-Typen betreffen. Es wird festgelegt welche Topology-Typen miteinander eine Beziehung eingehen dürfen und wieviele Instanzen eines Typs in Beziehung zueinander stehen dürfen.

Data Manager

Die Verwaltung der tatsächlich angelegten Topologie übernimmt *der Topology Data Manager*. Am *Data Manager Interface* wird Client-Objekten ermöglicht, eine Topologie aufzubauen, die den, mit Hilfe des MetaData Managers aufgestellten *Topology rules* genügen.

Query Manager

Der Topology Query Manager übernimmt die Aufgabe der Beantwortung von Anfragen an eine bestimmte Topologie, die jeweils in der *Topology Query Language (TQL)* formuliert sind.

Das Basisgebilde des Topology Service ist die Entität (*entity*), das durch einen *Identifier* und einen Typen (*topological type*) charakterisiert wird. Eine Topologie wird definiert, indem Typen spezifiziert und deren Beziehung untereinander durch Festlegung von Regeln (*topology rules*) beschrieben werden. In diesen Regeln werden neben der Kardinalität, d.h. der Anzahl der Instanzen eines Typs, auch die erlaubten Assoziationen zwischen den Typen festgesetzt. Der Aufbau einer Topologie geschieht schließlich durch das tatsächliche Bilden von Beziehungen zwischen Typinstanzen, die sich nach den aufgestellten Regeln richten. Eine Regel wird durch das folgende 8-Tupel beschrieben:

(type1, role1, min-cardinality1, max-cardinality1, type2, role2, min-cardinality2, max-cardinality2)

Weiterhin ist die Möglichkeit gegeben, zwischen Typen Vererbungshierarchien aufzubauen, in der ebenfalls die jeweils festgelegten Regeln an den Subtypen weitergegeben werden. Um eine Enthaltenseinshierarchie darzustellen, wird ein *aggregate entity* gebildet. Werden Entitäten als Teil einer größeren Ressource angesehen, so gehen diese einzelnen Entitäten eine *tie*-Verbindung ein und bilden eine aggregate entity. Als Einschränkung wurde im RFP des Topology Services festgelegt, daß jeweils nur eine Instanz eines Typen in genau einem aggregate entity vorkommen darf. Als AE könnte z.B. ein Computer angesehen werden, der aus den *entities* vom Typ Netzwerkkarte, Grafikkarte, Prozessor und DNS-Name besteht. Als *topology rule* könnte z.B. eine Assoziation zwischen mehreren Netzwerkkarten und einem Hub festgelegt werden, um ein LAN-Segment zu beschreiben.

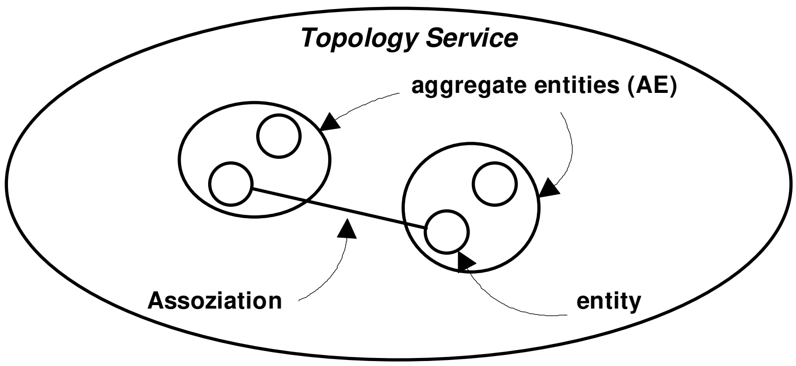

Abbildung 4-8: Basisgebilde des Topology Service

Mit dem Aufbau einer Topologie durch die Bildung von Assoziationen zwischen Entitäten und damit den aggregate entities (AE) in denen sie enthalten sind, wird ein *Graph* konstruiert, dessen *Kanten* die Assoziationen und dessen *Knoten* die aggregate entities sind. Um beim Beispiel zu bleiben, würden 10 Netzwerkkarten (= Entität), die miteinander verbunden sind, ein LAN-Segement beschreiben, in der 10 Computer (= *aggregate entity*) miteinander kommunizieren können. Eine Suche innerhalb einer Topologie äußert sich durch die Angabe eines *Navigationspfades* (*navigation path*) durch diesen Graphen. Ein Navigationspfad besteht aus einem oder mehreren *AE-Mustern* (*AE-patterns*), die zusammen mit *Assoziationsmustern* (*association-patterns*) formuliert werden. AE-Muster können zusammengesetzt werden aus einem *AE identifier*, bzw. Teile des identifiers vermischt mit wildcard-Symbolen, oder einem

Entscheidungskriterium für die gültigen topology types. Die Assoziationsmuster enthalten Entscheidungskriterien, die die Kanten des Graphen während der Suche betreffen. Vor Beginn jeder Suche, wird ein AE als Startpunkt ausgewählt. Die Suche, und damit das AE-Muster kombiniert mit dem Assoziationsmuster, wird in der *Topology Query Language (TQL)* ausgedrückt, deren Syntax in Abbildung 4-9 an einem Beispiel dargestellt ist.

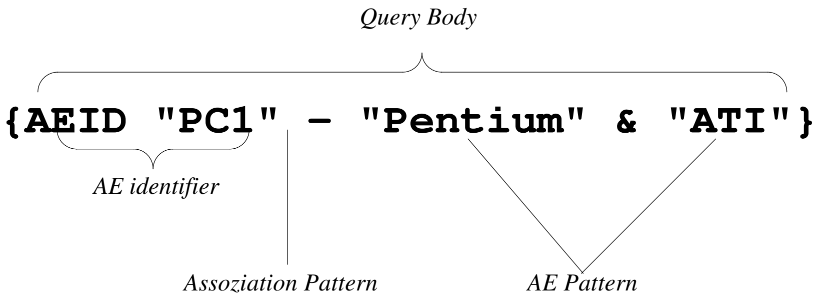

Abbildung 4-9: TQL Syntax

Die komplette Suchanweisung befindet sich in geschweiften Klammern. AE-Muster werden in Hochkommata gesetzt. Handelt es sich dabei um einen identifier, so wird das Schlüsselwort *AEID* vorangestellt, ansonsten wird angenommen, daß ein topological type deklariert wurde. Assoziationen zwischen AE-Muster werden durch einen Bindestrich gekennzeichnet. In dem Beispiel aus Abbildung 4-9 werden vom AE „PC1“ als Startpunkt alle AE gesucht, die Entitäten vom Typ „Pentium“ und „ATI“ enthalten und eine Verbindung zu „PC1“ haben. Es wird bei diesem Beispiel angenommen, daß jeweils „Pentium“ ein Subtyp von „Prozessor“ und „ATI“ ein Subtyp von „Grafikkarte“ ist. Werden AEs als geschlossene Einheiten betrachtet, so kann der obige Suchausdruck folgendermaßen umgangssprachlich ausgedrückt werden: *Suche alle Computer mit einem Pentium-Prozessor und einer ATI-Grafikkarte, die vom PC1 aus erreichbar sind.* Auf tieferes Eingehen in die Syntax der TQL wird an dieser Stelle verzichtet und statt dessen auf die TQL-Spezifikation im RFP des Topology Service ([HePa97], S.72 ff) verwiesen.

Trotz der sehr einfach gehaltenen Syntax der TQL, erweist sich die Sprache als sehr mächtig. Durch Kombination aus mehreren AE- und Assoziationsmustern lassen sich in knapper Form sehr komplexe Anfragen konstruieren. Durch die Entstehungsgeschichte des Topology Service und dem Netz- und Systemmanagement als dessen Entstehungsgrund, eignet sich der Topology Service hervorragend dafür, eine Momentaufnahme der gegenwärtigen Netztopologie zu präsentieren. Damit ist der Topology Service auch als Domain Service der in Kapitel 4.1 beschriebenen MA einsetzbar, und eignet sich für diese Aufgabe besser als der *Naming Service*. Da noch keine Implementierung des Topology Service zum Zeitpunkt der Implementierung der Komponenten der MA vorhanden war, werden beide Services als Domain Service unterstützt, wobei der Naming Service bei den Testläufen eingesetzt wurde. Parallel dazu wird in einem Fortgeschrittenenpraktikum [Roel98] am Lehrstuhl der Topology Service implementiert, so daß nach dessen Fertigstellung dieser dem Naming Service vorzuziehen ist.

4.2.3.3 Der Event Service

Der *CORBA Event Service* ([OMG97b], S.4-1ff; [OHE96], S.119ff) ermöglicht Objekten sich dynamisch für das Auftreten bestimmter Events zu registrieren. Das Vorkommnis eines internen Ereignisses in einem Objekt wird als *Event* bezeichnet, falls davon ausgegangen werden kann, daß bei anderen Objekten das Interesse bestehen könnte von diesem Ereignis zu erfahren. Dies bedeutet auch, daß jedes Objekt selber die Events definiert, für die sich andere Objekte registrieren können. Die Nachricht bei Auftauchen eines Events, die von einem Objekt an alle registrierten Objekte geschickt wird, wird als *Notification* bezeichnet. Der Event Service kümmert sich dabei um die korrekte Verteilung der Notification an alle dafür registrierten Objekte, d.h. implizit auch, daß die jeweiligen Event-Produzenten keine Kenntnis über die registrierten Objekte haben müssen. Dadurch wird ein gewisser Grad an Unabhängigkeit zwischen den Event-Produzenten und den registrierten Objekten gewahrt.

Wie aus dem vorhergehenden Abschnitt geschlossen werden kann, werden innerhalb des Event Service grundsätzlich zwischen zwei teilnehmende Parteien unterschieden:

- *Event Supplier*: Im Event Supplier treten die Ereignisse auf, für deren Benachrichtigung sich andere Objekte registrieren können.
- *Event Consumer*: Der Event Consumer registriert sich für die ihn interessierenden Events.

Um die oben beschriebene Unabhängigkeit zwischen Supplier und Consumer ermöglichen zu können, fungiert ein *Event Channel* als Black Box: der Supplier schickt bei Auftreten eines Events eine Nachricht an den Event Channel, der diese Nachricht an alle Consumer weiterleitet, die sich bei ihm registriert haben. Dies bedeutet auch, falls sich mehrere Supplier an einen Event Channel gebunden haben, so werden dementsprechend *alle* Notifications an alle registrierten Consumer weitergeleitet, auch wenn sich einige darunter befinden, die sich nur für die Events eines bestimmten Suppliers interessieren. Ebenso erhalten Consumer die Notifications aller Events eines Suppliers, auch wenn das Interesse nur für ganz bestimmte Events dieses Suppliers besteht. Daraus kann demzufolge geschlossen werden, daß die Event-Filterung in den Consumern stattfinden muß. Diese zusätzliche Funktionalität wird, wie im nächsten Kapitel gezeigt wird, durch den *Notification Service* abgedeckt.

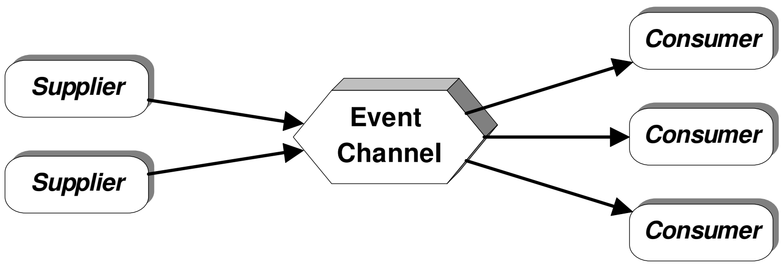

Abbildung 4-10: Funktionsweise des Event Channels

Um den obigen Event Channel zu realisieren, bindet sich der Supplier nicht direkt an den Event Channel, sondern an einen *Proxy Consumer*. Komplementäres gilt für den Consumer, der sich tatsächlich an einen *Proxy Supplier* bindet. Der Event Channel kümmert sich daraufhin um die Verbindungen zwischen den Proxy Suppliern und den Proxy Consumern.

Die Spezifikation des Event Services sieht zwei verschiedene Modelle der Event-Benachrichtigung, und damit der Notification-Weiterleitung, vor (siehe auch Abbildung 4-11):

Push Modell

Im Push Modell müssen die Consumer zwei Methoden implementieren: *push()* und *disconnect_push_consumer()*. Der *Push Supplier* bindet sich innerhalb des Event Channels an einen Proxy Push Consumer. Bei Auftreten eines Events wird vom Supplier die push-Operation des Proxy Consumers mit einer Nachricht als Argument aufgerufen. Diese wird an den Proxy Push Supplier der tatsächlichen Consumer weitergeleitet, die daraufhin deren push-Methode aufrufen. Die von den Push-Consumern implementierte push-Methode enthält die auf dieses Event hin auszuführenden Operationen. In den meisten Fällen wird aber zunächst die mitgeschickte Nachricht ausgewertet, um festzustellen, ob tatsächlich das für den Consumer relevante Event aufgetreten ist.

Pull Modell

In diesem Fall müssen vom Supplier zwei Methoden implementiert werden: *pull()* und *disconnect_pull_supplier()*. Der Consumer führt ein Polling gegenüber dem Supplier aus, indem – meist in regelmäßigen Abständen – die *try_pull*-Methode des Proxy Suppliers aufgerufen wird. Ist auf der Seite des Pull Suppliers tatsächlich ein Event aufgetreten, so kann die pull-Methode vom Consumer aufgerufen werden. Die pull-Methode liefert als Rückgabewert schließlich die Notification des Suppliers, die anschließend ausgewertet werden kann.

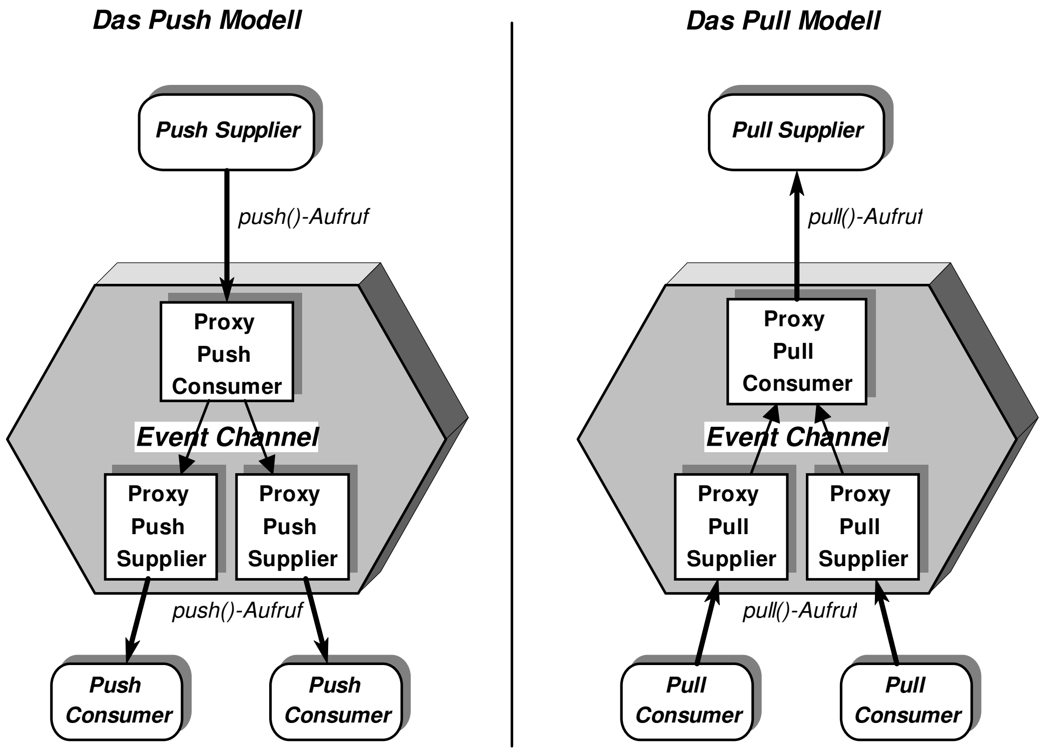

Abbildung 4-11: Push- und Pull-Modell

Die Notification, die vom Supplier in Form eines Methodenarguments bzw. als Rückgabewert einer Operation an den Consumer geschickt wird, muß vom Typ *CORBA Any*[4] sein. Daraus folgt, daß jeder Consumer im voraus den Aufbau der zu erhaltenden Nachricht kennen muß. Deswegen ist es ratsam, die Struktur einer empfangenen Notification zu determinieren, um den Consumern eine einheitliche Handhabung zu ermöglichen.

[4] Es ist ebenfalls ein strukturierter Datentyp mit dem Namen *TypedEvent* spezifiziert worden, der aber von der vorliegenden Implementierung von *Visigenic* nicht unterstützt wird und deshalb im weiteren unbeachtet bleibt.

Wie aus der Beschreibung des Event Services zu ersehen ist, eignet sich v.a. das Push Modell als Monitoring Service der Managementanwendung, die in Kapitel 4.1 skizziert wurde. Von *Visigenic* existiert ebenfalls eine Implementierung des Event Service, so daß sich dessen Einsatz im MS anbieten würde.

4.2.3.4 Der Notification Service

Der *Notification Service* basiert auf dem *Event Service*, der im vorhergehenden Kapitel beschrieben worden ist, und enthält die folgenden Erweiterungen, die in dessen Submission [BEA98] spezifiziert worden sind:

- Die gesendeten und empfangenen Events entsprechen einer wohldefinierten, festgelegten Datenstruktur.
- Clients können durch Filtereinstellungen am Proxy-Supplier des Event Channels genau festlegen, welche Events sie empfangen wollen.
- Supplier können feststellen, welche Clients an den Event Channel angebunden sind und welche Event-Typen von diesen verlangt werden, um so ein Verschicken von Events zu vermeiden, falls kein Interesse dafür besteht.
- Clients können im Gegenzug feststellen, welche Event-Typen von den Suppliern eines Event Channels unterstützt werden, um sich so dynamisch für neue Events abonnieren zu können.
- Es können gegenüber dem Event Channel, den Proxy-Objekten, als auch gegenüber den Events, *quality of Service*parameter definiert und eingestellt werden.
- Optional existiert eine *Event-Type-Repository*, die es Clients ermöglicht, anhand der zur Verfügung gestellten Informationen Filtereinstellungen zu tätigen.

Da im Zeitraum der Entwicklung der Managementanwendung keine Implementierung des Notification Services verfügbar war und die Architektur des Notification Service der des Event Service sehr ähnlich ist, wird an dieser Stelle auf die Darstellung der kompletten Architektur verzichtet und auf die Spezifikation [BEA98] verwiesen. Statt dessen beschränkt sich die weitere Betrachtung des Notification Service auf die Darstellung der wohldefinierten Event-Datenstruktur, deren Festlegung, wie schon in Kapitel 4.2.3.3 festgestellt wurde, auch für die in Kapitel 4.1 skizzierten MA von Bedeutung ist. Ebenfalls kann natürlich diese Datenstruktur, unabhängig von einer tatsächlichen Implementierung des Notification Services, zusammen mit dem Event Service zum Austausch von Daten zwischen Supplier und Consumer verwendet werden.

Als *Structured Event* wird die wohldefinierte Event-Datenstruktur bezeichnet, die von den Suppliern gesendet und von den Consumern empfangen wird. Der Aufbau eines Structured Events ist in Abbildung 4-12 dargestellt und setzt sich aus den folgenden Bestandteilen zusammen:

Event Header

Der Event Header besteht aus einem festen (*Fixed Header*) und einem variablen Part (*Variable Header*). Durch diese Aufteilung soll überflüssiger Overhead v.a. bei kurzen Nachrichten vermieden werden. Der Fixed Header wiederum wird in die String-Felder *domain_type*, *event_type* und *event_name* aufgegliedert. Der *domain_type* enthält die vertikale Industriezugehörigkeit und könnte mit Werten wie z.B. *Telekommunikation*, *Finanzwesen*, etc. belegt werden. Der *event_type* bestimmt die Art des Events, wobei hier Feldbelegungen wie *ShutDown*, *ServerUp*, etc. denkbar wären. Das *event_name*-Feld sollte mit dem eindeutigen Namen bzw. dem eindeutigen Identifier der Instanz des Events belegt werden. I.d.R. wird an dieser Stelle der *compound name* der Event-Instanz erwartet, falls der *Naming Service* eingesetzt wird. Der *Variable Header* setzt sich aus 0 oder mehr Attributnamen (*ohf_name*) und deren Wertebelegungen (*ohf_value*) zusammen.

Event Body

Der Event Body ist zweigeteilt, bestehend aus dem *Filterable Body* und dem *Remaining Body*. Der *Filterable Body* enthält die Daten, nach dem es den Event-Consumern ermöglicht werden soll, Filtereinstellung vorzunehmen. Jede Position besteht aus einem Attributnamensfeld (*fd_name*) und dessen Belegung (*fd_value*). Der Remaining Body hingegen enthält Daten, die sich wegen ihrer Größe nicht zum Filtern eignen, aber zu einer eventuellen Auswertung durch den Consumer genutzt werden können. Denkbar wären an dieser Stelle z.B. *core*-Dateien, die mehr Informationen über ein Programmabsturz enthalten.

Der Structured Event kann also ohne weiteres zusammen mit dem Event Service zur strukturierten Darstellung des Nachrichtenaustauschs zwischen Supplier und Consumer eingesetzt werden. Dies würde den weiteren Vorteil ergeben, daß bei einer später hinzukommenden Implementierung des Notification Service, diese wohldefinierte Datenstruktur bereits verwendet werden würde und dadurch u.a. die Filtermöglichkeiten des Notification Services sofort genutzt werden können, ohne daß die Supplier und Consumer nachträglich angepaßt werden müßten.

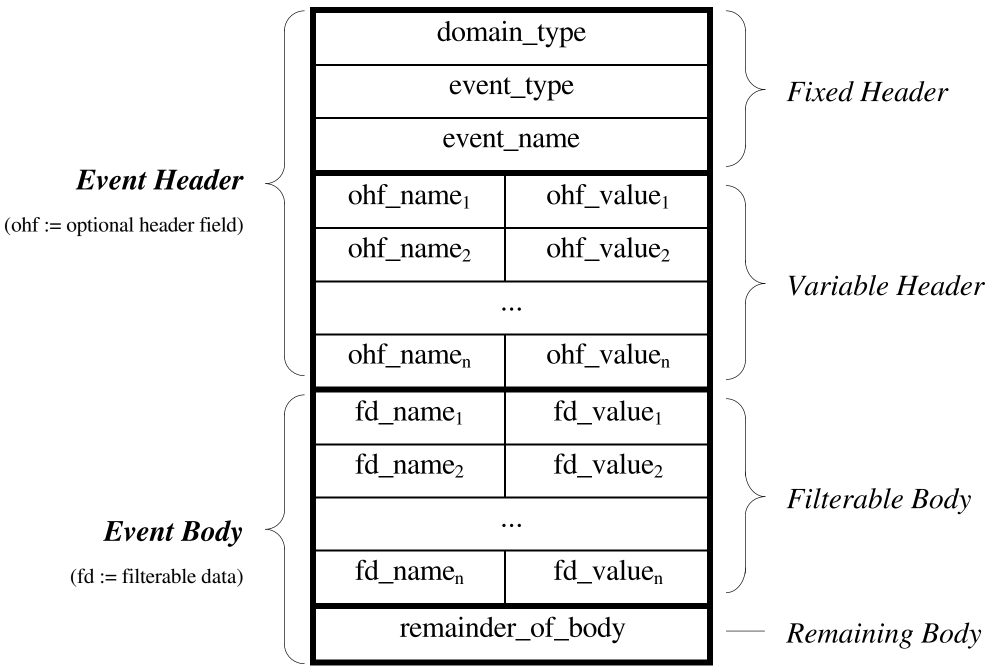

Abbildung 4-12: Die Struktur des Structured Events

4.2.4 Zusammenfassung

Wie dieser kurze Überblick über CORBA und dessen Services gezeigt hat, eignet sich diese Middleware hervorragend zur Implementierung der in 4.1 skizzierten Managementanwendung. Ein Teil der Komponenten der Anwendung, insbesondere der *Policy Service* und der *Enforcement Service*, würden als CORBA-Objekte realisiert werden. Beide würden aber innerhalb des MS sowohl als Server als auch als Client agieren. In der Rolle des Server-Objekts würden sie gegenüber den anderen Komponenten der MA mit einer wohldefinierten IDL-Schnittstelle ihre Funktionalität anbieten, während sie als Client-Objekt die Funktionalität der anderen MA-Bestandteile nutzen würden. Durch die Ortstransparenz von CORBA-Objekten ist das Ziel der Skalierbarkeit der Architektur ebenfalls erreichbar. Zusätzlich ergibt sich mit den CORBA Services die Möglichkeit, die restlichen MA-Komponenten durch diese zu ersetzen. Es bietet sich an, den *Monitoring Service* durch Nutzung der Funktionalität des *CORBA Event Services* zu substituieren, wie dies im Falle des *Domain Service* zusammen mit dem *CORBA Naming Service* bzw. des *CORBA Topology Service* möglich wäre. In dem nun folgenden Kapitel 4.3 wird ausführlich die in einer CORBA-Umgebung realisierte Managementarchitektur erörtert.

4.3 Gesamtarchitektur

Alle implementierten Komponenten der MA sind als CORBA-Objekte in der Programmiersprache *Java (Version 1.1.6)* [Sun98] realisiert worden. Auf Implementierungsgesichtspunkte, d.h. auf den Entwicklungsprozeß der CORBA-Objekte von der IDL-Spezifikation der Schnittstelle bis zur Implementierung in *Java* sowie auf die eingesetzte CORBA-Implementierung *Visibroker 3.0 von Visigenic* [Visi97], wird in Kapitel 6 ausführlich eingegangen. Im folgenden werden, nach einem ersten Überblick, alle Bestandteile der Managementanwendung, deren nach außen sichtbaren Schnittstellen sowie ihr Zusammenspiel untereinander, detailliert erklärt. Desweiteren gelten im folgenden die nachstehenden Begriffsbedeutungen (siehe hierzu auch Abbildung 3-7) :

- *erweiterter Domänenbezeichner* ist aus dem Wortbereich <EXTENDED_XFN>
- *einfacher Domänenbezeichner* ist aus der Wortmenge <XFN_IDENTIFIER>

4.3.1 Überblick

In Abbildung 4-13 sind die Elemente der Managementanwendung sowie eine grobe Darstellung der Kommunikationswege, skizziert. Um das Ziel der Unabhängigkeit der verschiedenen entwickelten Services zu verwirklichen, sind die Komponenten folgendermaßen aufgebaut:

Service

Der Service bietet in seiner Gesamtheit eine bestimmte Funktionalität an, die durch eine öffentliche Schnittstelle genutzt werden kann. Ein Service ist grundsätzlich zweigeteilt in eine *Factory* und den von dieser Factory produzierten *Objekte*. Außer beim *Persistence Service* sind sowohl die Factory als auch die Objekte CORBA Server-Objekte und bieten für sich eine eigene Schnittstelle an. Wie schon in Kapitel 4.2.4 allgemein festgestellt wurde, können aber sowohl Factory als auch die produzierten Objekte gegenüber anderen Services als Clients agieren und dadurch deren Funktionalität nutzen.

Mobile Agent
Jeder *Service* der Managementanwendung wird tatsächlich durch einen *Mobile Agent* verwirklicht. Wird ein Mobile Agent hochgefahren, so ist der ihm zugewiesene Service von CORBA Clients an der öffentlichen Schnittstelle nutzbar. Ein Mobile Agent wird auf einem *Agentensystem* kreiert, das die Kontrolle über alle auf ihm instanziierten Agenten hat. Ein Mobile Agent hat gegenüber einem *Stationary Agent* die zusätzliche Fähigkeit, während des laufenden Betriebs das Agentensystem zu wechseln, indem es auf einen anderen *migriert*. Durch diese Art der Verwirklichung eines Services, ist dessen geforderte Unabhängigkeit und Ortstransparenz grundsätzlich gewahrt und ermöglicht.

Agentensystem
Das *Agentensystem* stellt den Lebensraum der auf ihm kreierten Agenten zur Verfügung, d.h. im konkreten Fall wird durch dessen Hochfahren eine eigene *Java Virtual Machine (VM)* gestartet. Pro *Host* in einem Netzwerk, wird höchstens ein Agentensystem instanziiert. Ist es nötig, daß z.B. ein Host neu gebootet werden muß oder eine bessere Lastverteilung im Netz erreicht werden soll, so besteht die Möglichkeit, daß alle *Mobile Agents* auf ein anderes Agentensystem im Netzwerk wechseln. Dadurch wird die Ausfallsicherheit der Agenten und damit der Services, die diese realisieren, erhöht. Das Agentensystem bietet zusätzlich eine eigene GUI zur Steuerung der Agenten an.

Agenten GUI
Jeder Agent hat seine eigene GUI, mit dem sowohl (eingeschränkt) die Operationen der IDL-Schnittstelle des auf ihm realisierten Services als auch agentenspezifische Funktionen, wie das Initiieren der Migration, aufgerufen werden können. Über die GUI des Agentensystems ist es möglich an die GUIs der Agenten zu gelangen.

Das hier vorgestellte Konzept entspricht der *Mobile Agent System Interoperability Facilities Specification (MASIF)* [OMG97c], die ihm Rahmen einer Diplomarbeit [Kemp98] am Lehrstuhl umgesetzt wurde.

Eine zusätzliche GUI ermöglicht ebenfalls, den *CORBA Naming* und *Topology Service* direkt als Client anzusprechen, um einen Überblick des gegenwärtigen Netzzustands darzustellen.

Im folgenden wird noch explizit eine Zuordnung zwischen den Komponenten aus dem Architekturbild in der Abbildung 4-2, das die Anforderungen an die MA darstellte, und der tatsächlich realisierten, verteilten Anwendung aus Abbildung 4-13 getätigt:

Policy Service
Der Policy Service wird durch den *Policy Factory Mobile Agent (PFMA)* realisiert. Er beinhaltet die gesamte Funktionalität des Policy Service, die bei der Anforderungsanalyse an diesen gestellt wurde, bis auf das persistente Anlegen der Policies, das nun durch den *Persistence Service Mobile Agent* separat erfüllt wird.

Enforcement Service
Der Enforcement Service wird durch den *Enforcement Object Factory Mobile Agent (EOFMA)* verwirklicht, der die ihm in der Anforderungsanalyse zugewiesene Funktionalität vollständig erfüllt.

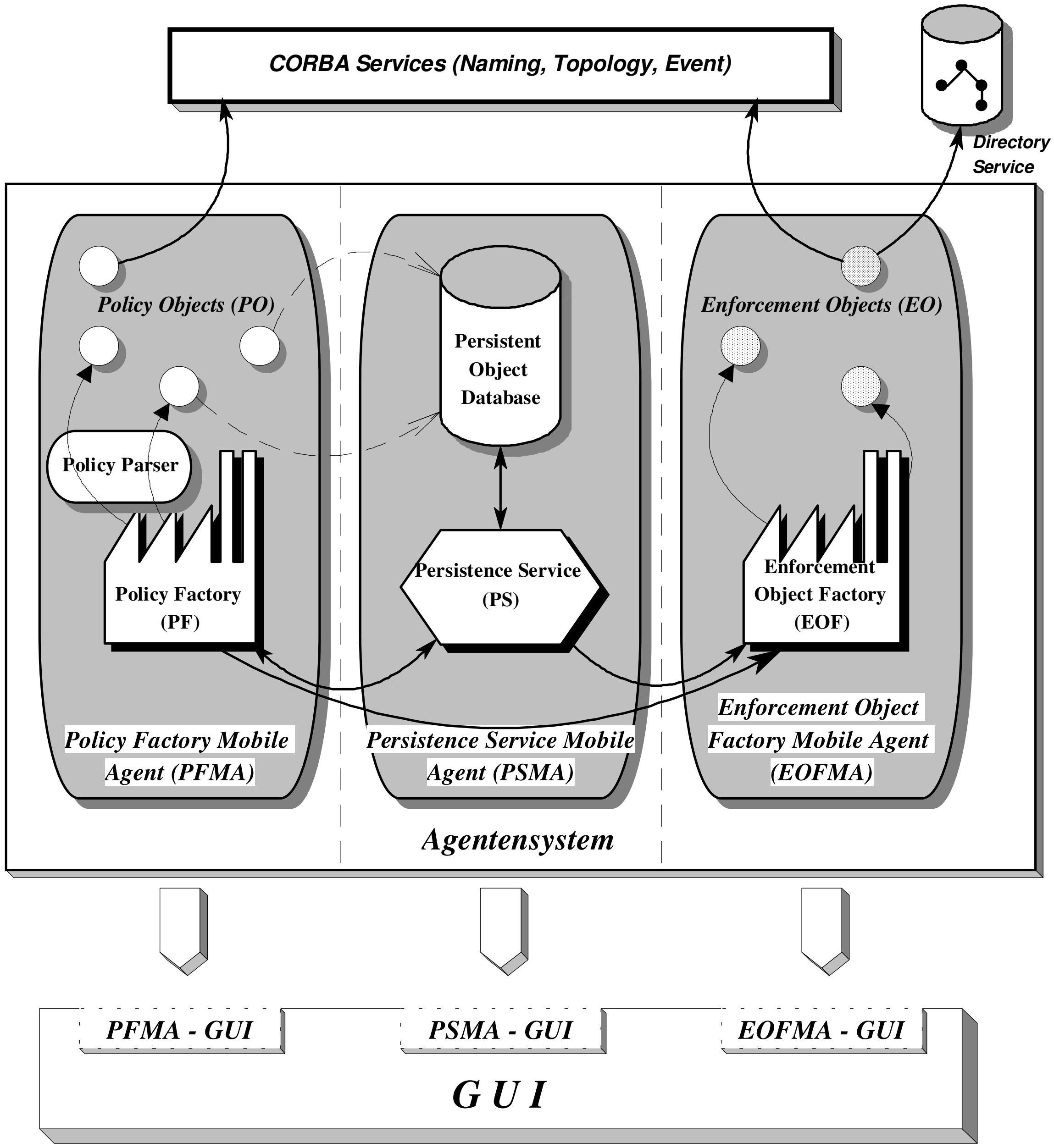

Abbildung 4-13: Gesamtüberblick der Komponenten der Managementanwendung

Monitoring Service

Der Dienst des Monitoring Services wird durch die CORBA Services *Event* und *Notification Service* erbracht. Zum Zeitpunkt der Implementierung des Prototypen stand nur eine Implementierung des *Event Services* von *Visigenic* [Visi97b] zur Verfügung.

Domain Service

Der Domain Service ist im vorliegenden Fall zweigeteilt. Er wird einerseits durch die CORBA Services *Naming* und *Topology Service* realisiert, um das Netz in Domänen zu strukturieren und Anfragen, die die gegenwärtige Situation im zu managenden Netz betreffen, zu beantworten. Andererseits kommt ein *Directory Service* [X.500] für die Modellierung der Heimatdomäne nomadischer Systeme zum Einsatz, da sich dieser durch seine Zielsetzung besser als die CORBA Services dafür eignet. Im Laufe dieses Abschnitts wird diese Thematik detaillierter erläutert.

4.3.2 Der Persistence Service

In der Anforderungsanalyse in Kapitel 4.1 wurde beim Policy Service festgestellt, daß die generierten Policies sowie deren Zustand in geeigneter Weise gespeichert werden müssen, um sie zu gegebener Zeit wieder herstellen zu können. Zwei Möglichkeiten boten sich während der Implementierung der MA, die Persistenz der Policies zu wahren:

- der Policy Service achtet selbständig und intern auf die Persistenz der kreierten Policy Objekte
- es wird ein unabhängiger Service realisiert, der nicht nur Policies sondern CORBA-Objekte allgemein persistent anlegen kann

Die Entscheidung fiel zugunsten eines eigenen unabhängigen Services, den *Persistence Service*, um so auch bei späteren Erweiterungen der MA dessen Funktionalität nutzen zu können. Der implementierte Persistence Service richtet sich in den Grundzügen seines Aufbaus nach der Spezifikation des *CORBA Persistent Object Service* ([OMG97], S.5-1 ff). Es handelt sich hierbei jedoch nicht um eine echte Untermenge des CORBA Services, sondern es wurde vielmehr das Konzept der Spezifikation übernommen. Die spezifizierten Teile des *CORBA Persistent Object Services* wurden vereinfacht und in einer statt in mehreren Komponenten vereinigt, um möglichst schnell eine lauffähige Version des Services zu erhalten. Es handelt sich also hier um eine auf das Wesentliche beschränkte Version des CORBA Services.

In Abbildung 4-14 sind die Bestandteile des Persistence Services skizziert. Der Persistence Service teilt sich in die *Persistent Object Factory*, die eine API zum Erstellen von *Persistent Objects* beinhaltet, und dem eigentlichen *Persistence Service*, der als CORBA Server realisiert ist und an seinem IDL-Interface Methoden zum speichern, wiederherstellen, einfügen und entfernen von Persistent Objects bietet. Der Persistence Service wird tatsächlich durch den *Persistence Service Mobile Agent (PSMA)* realisiert und verwaltet intern die eingefügten Persistent Objects in einer Datenbank.

Um den Persistence Service sowohl für alle möglichen CORBA Objekte offen zu halten als auch diese in einer Datenbank halten und später speichern zu können, bedarf es einer besonderen, nämlich serialisierten Darstellung dieser Objekte. Dies wird erreicht, indem die CORBA Objekte zunächst in *Persistent Objects* umgewandelt werden. Ein *Persistent Object* besteht aus den folgenden zwei Feldern (siehe auch IDL Spezifikation in Abbildung 4-13):

- *PID-Feld:* Der eindeutige PID (**P**ersistent Object **Id**entifier) wird als Schlüsselwert für die Datenbank und zur Identifizierung des Persistent Objects verwendet
- *Obj-Feld:* Dieses *ByteArray*-Feld enthält die serialisierte Darstellung des CORBA-Objekts

Die *Persistent Object Factory* ist eine Java API, die Methoden sowohl zum Kreieren von Persistent Objects als auch zum Extrahieren eines CORBA Objekts aus einem Persistent Object, beinhaltet.

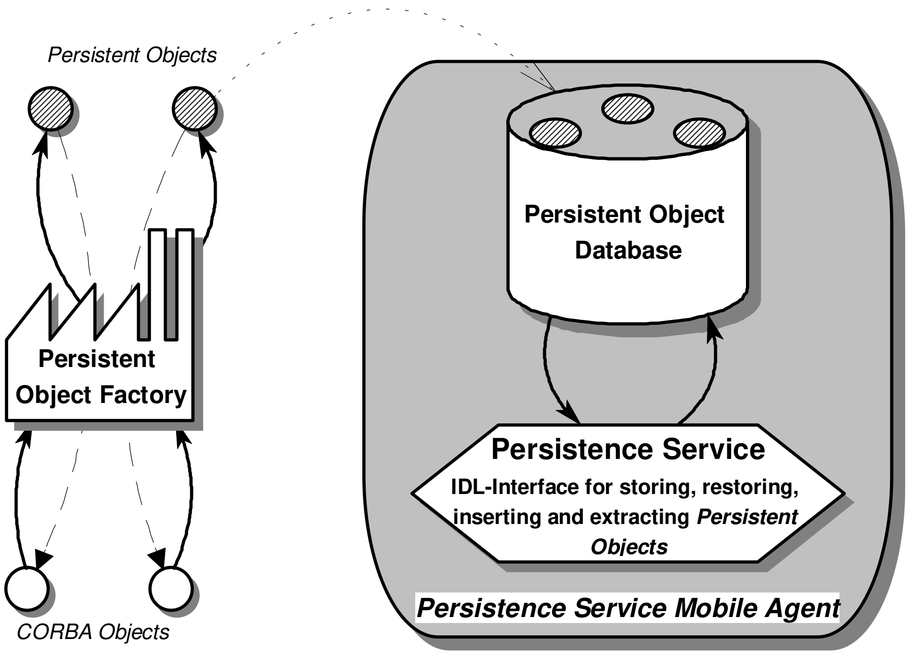

Abbildung 4-14: Die Bestandteile des Persistence Service

In Abbildung 4-15 sind die Datentypen, die innerhalb des Persistence Service verwendet werden, und die Exceptions spezifiziert, die von den einzelnen Operationen des IDL-Interfaces geworfen werden.

```
module persistence {
 exception UserDBCorrupted{string reason;};
 exception UserExists{string reason;};
 exception UserNotFound{string reason;};
 exception ObjectNotFound {string reason;};
 exception DBAlreadyInUse{string reason;};
 exception NoDBAllocated{string reason;};
 exception StoreFailure{string reason;};
 excpetion RestoreFailure {string reason;};

  typedef sequence<octet> ByteArray;
  typedef sequence<PersistentObject> POlist;
  typedef sequence<string> IDlist;

  struct PersistentObject  { ByteArray Obj;
                             string PID;};
  };
```

Abbildung 4-15: Allgemeine Datentypen des Persistence Service

In Abbildung 4-16 ist die IDL Spezifikation der Operationen des *Persistence Service* zu ersehen. In der vorliegenden Implementierung ist pro Instanz des Persistence Service genau eine Datenbank von Clients ansprechbar. Wird von verschiedenen Clients zur gleichen Zeit der Zugriff auf mehrere Datenbanken benötigt, so müssen entsprechend viele Instanzen des PS kreiert werden. Im folgenden werden die einzelnen, durch die IDL-Schnittstelle sichtbaren Operationen des Persistence Service detailliert erklärt:

void connect(string userName)
: Diese Operation meldet einen neuen Client des Persistence Service mit dem Namen *userName* an. Die vom PS angebotenen Operationen sind erst nach durchgeführter Anmeldung erfolgreich ausführbar. Intern werden die Namen aller Clients in einer Liste gehalten.

void disconnect(string userName)
: Es wird der Client mit dem Namen *userName* wieder abgemeldet. Ist dies der letzte User einer vorher allokierten Datenbank, so wird die Verbindung zu dieser Datenbank wieder freigegeben. Alle in der Datenbank enthaltenen Daten gehen verloren, falls nicht vorher diese mit der Operation *store()* gesichert worden sind.

void allocateNewDB(string userName)
: Eine neue Datenbank wird allokiert. Falls bereits eine Verbindung zu einer Datenbank besteht und noch mindestens ein weiterer Client als User angemeldet ist, schlägt diese Operation fehl.

void store(string fileName, string userName)
: Die DB, zu der im Augenblick des Operationsaufrufs eine Verbindung besteht, wird unter dem Namen *fileName* gesichert, indem diese als ganzes serialisiert und gepackt als Datei gespeichert wird.

void restore(string fileName, string userName)
: Die DB mit dem Namen *fileName* wird wieder hergestellt, indem diese aus der Datei mit diesem Namen deserialisiert und entpackt wird. Diese Operation schlägt fehl, falls zum Zeitpunkt des Operationsaufrufs bereits eine Verbindung zu einer Datenbank besteht und mindestens ein weiterer User angemeldet ist.

void addElement(PersistentObject newObject, string userName)
: Das Persistent Object *newObject* wird in die Datenbank, mit der PID als Schlüssel, eingefügt. Diese Operation schlägt fehl, falls keine Verbindung zu einer Datenbank zum Zeitpunkt des Aufrufs besteht. Ist bereits ein Objekt mit dieser PID in der DB enthalten, so wird aufgrund der Eindeutigkeit der PID angenommen, daß es sich um eine Aktualisierungsmaßnahme handelt, so daß das bereits enthaltene Objekt durch das neue ersetzt.

void deleteElement(string PID, string userName)
: Durch Aufruf dieser Operation wird das Persistent Object mit der übergebenen PID aus der DB gelöscht.

PersistentObject getElement(string PID, string userName)
: Das Persistent Object mit der ID *PID* wird in der Datenbank gesucht und im Erfolgsfall zurückgeliefert.

string[] listAllID(string userName)
: Mit dieser Operation ist es möglich, sich die PIDs aller Persistent Objects aus der aktuellen DB zurückgeben zu lassen.

PersistentObject[] getAllObjects(string userName)
Alle in der derzeit aktuellen DB enthaltenen Persistent Objects werden innerhalb eines Arrays zurückgeliefert.

boolean DB_allocated()
Es wird festgestellt, ob bereits eine Verbindung zu einer Datenbank besteht. Falls dies wahr ist, kann mit der Operation *getDBName()* der Name dieser Datenbank ermittelt werden. Es tritt häufig der Fall ein, daß ein Client eine ganz bestimmte Datenbank benötigt. Um Herauszufinden ob sich das Anmelden bei diesem Persistence Service lohnt, können diese Operationen vorher zur Klärung verwendet werden.

string getDBName()
Der Name der Datenbank, zu der zum Zeitpunkt des Operationsaufrufs eine Verbindung besteht, wird zurückgeliefert. Handelt es sich dabei um eine neu angelegte Datenbank, die noch nicht gespeichert wurde, so wird '*not set*' zurückgegeben. Ebenfalls ist '*not set*' das Ergebnis dieser Operation, falls keine Verbindung zu einer Datenbank besteht.

Weiterhin weist die API der *Persistent Object Factory* die folgenden zwei Operationen auf:

PersistentObject createPO(org.omg.CORBA.Object obj, String PID)
Ein Persistent Object wird aus dem übergebenen CORBA-Objekt kreiert, indem es serialisiert und als Byte-Array dargestellt wird. Damit wird der augenblickliche Zustand des Objekts persistent, in Form des Persistent Objects, angelegt. Der übergebene String wird als PID des neu kreierten Persistent Objects gewertet. Um ein intuitives Suchen in der Datenbank zu ermöglichen, empfiehlt es sich als PID den tatsächlichen Object Identifier zu übergeben, der i.d.R. der *compound name* des Objektes ist.

org.omg.CORBA.Object extractObject(PersistentObject PO)
Das übergebene Objekt wird wieder deserialisiert und in das entsprechende CORBA-Objekt umgewandelt. Dabei wird der Zustand des Objekts bei dessen Serialisierung wiederhergestellt.

```
module persistence {
  interface PersistenceService {
     void connect(in string userName)
          raises (UserExists, UserDBCorrupted);
     void disconnect(in string userName)
          raises (UserNotFound, UserDBCorrupted);
     void allocateNewDB(in string userName)
          raises (UserNotFound, DBAlreadyInUse);
     void store(in string fileName, in string userName)
          raises (UserNotFound, NoDBAllocated, StoreFailure);
     void restore(in string fileName, in string userName)
          raises (UserNotFound, DBAlreadyInUse, RestoreFailure);
     void addElement(in PersistentObject newObject, in string userName)
          raises (UserNotFound, NoDBAllocated);
     void deleteElement(in string PID, in string userName)
          raises (UserNotFound, NoDBAllocated, ObjectNotFound);
     PersistentObject getElement(in string PID, in string userName)
          raises (UserNotFound, ObjectNotFound);
     IDlist listAllID(in string userName)
          raises(UserNotFound, NoDBAllocated);
     POlist getAllObjects(in string userName)
          raises(UserNotFound, NoDBAllocated);
     boolean DB_allocated();
     string getDBName(); };
};
```

Abbildung 4-16: IDL Spezifikation des Persistence Service

4.3.3 Der Policy Service

Der *Policy Service* bietet CORBA Clients die Möglichkeit, strategische, zielorientierte und operationale Policies anzulegen. Wie aus Abbildung 4-17 zu ersehen ist, besteht der Policy Service aus einer *Policy Factory* und den *Policy Objects*, die jeweils auf dem *Policy Factory Mobile Agent* (*PFMA*) instanziiert werden. Beide Komponenten sind als CORBA Server realisiert und offerieren somit ihre Funktionalität an einer wohldefinierten IDL-Schnittstelle. Die GUI des *PFMA* bietet ebenfalls die Möglichkeit die Operationen der *Policy Factory* aufzurufen und damit Policies zu kreieren. Um die Persistenz der kreierten Policy Objects zu wahren, wird der in Abschnitt 4.3.2 beschriebene *Persistence Service* verwendet. Der Policy Service ist eng mit dem *Policy Enforcement Service* verbunden, der in Abschnitt 4.3.4 ausführlich betrachtet wird. Wird eine operationale Policy aktiviert, so wird mit Hilfe des *Policy Interpreters* deren *Policy Description* interpretiert und die benötigten *Enforcement Objects (EO)* kreiert. Während der Interpretation einer operationalen Policy, wird jeweils immer nur der momentane Netzzustand mit Hilfe des *Topology* bzw. des *Naming Service* ausgewertet. Um Zustandsänderungen durch Hinzukommen bzw. Entfernen von Managementagenten zu bemerken, bindet sich jede *aktive* operationale Policy an den bekannten Event Channel *AgentInit-Channel*, in den u.a. jeweils beim Hoch- bzw. Herunterfahren der Managementagenten eine Notification gesendet wird.

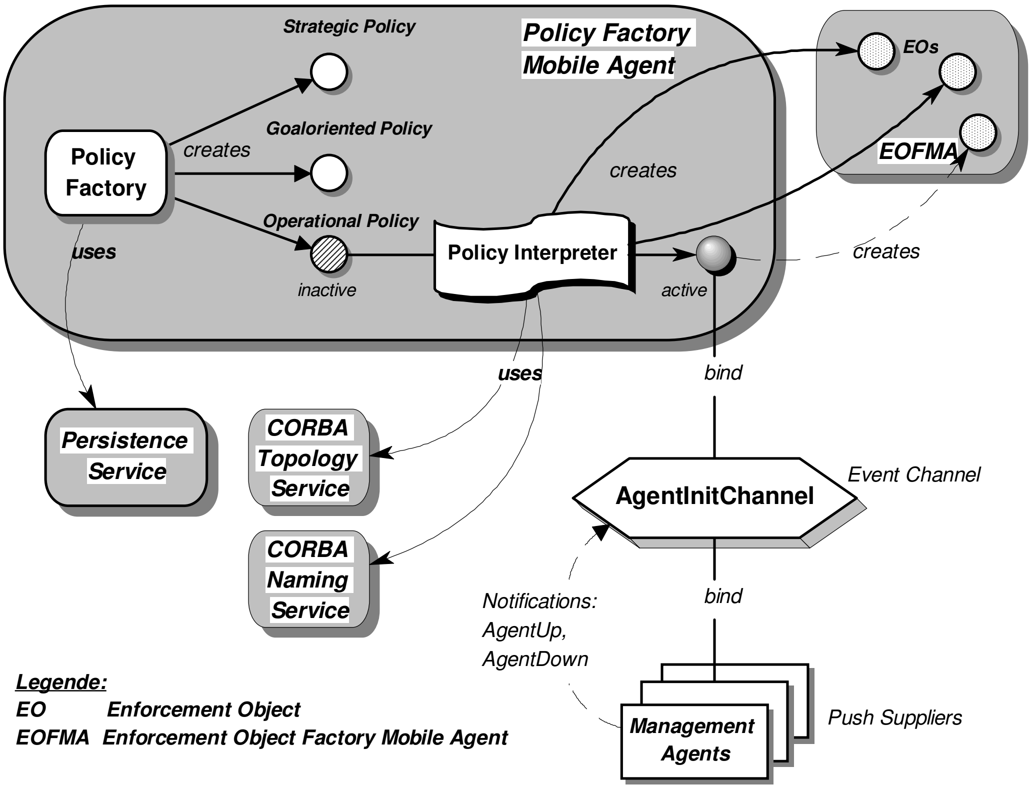

Abbildung 4-17: Die Komponenten des Policy Service

Im folgenden werden die in den Abbildungen 4-18 bis 4-20 dargestellten IDL Spezifikationen des Policy Service beschrieben.

Das CORBA Interface *Policy Object*, dessen IDL Spezifikation in Abbildung 4-18 zu sehen ist, repräsentiert den kleinsten gemeinsamen Nenner der Operationen, die alle drei Policy-Typen unterstützen müssen. Demzufolge erben die spezifizierten Interfaces der strategischen, zielorientierten und operationalen Policy von dieser Schnittstelle, deren Operationen nachstehend erklärt werden:

string getID()

Es wird der eindeutige Identifikator des Policy Objects zurückgeliefert, wobei es sich hierbei um den durch den Naming Service zugewiesenen *compound name* des CORBA Objekts auf dem ORB handelt. Der tatsächlich eindeutige Name der Policy, der dem letzten *name component* des *compound names* entspricht, besteht aus einer einmalig vergebenen Ziffernfolge, die während der Objektkreierung durch die *Policy Factory* errechnet wird. Die Darstellung des *compound names* erfolgt wie üblich als *erweiterter Domänenbezeichner*.

string getParentID()

Der eindeutige Identifikator der Eltern-Policy aus dem Wortbereich des *erweiterten Domänenbezeichners* wird durch Aufruf dieser Operation zurückgegeben, falls dieser durch die Operation *setParentID()* gesetzt worden ist. Ist dies nicht der Fall, so wird '*not set*' zurückgeliefert.

void setParentID(string PolID)
Diese Operation ermöglicht das Setzen des Identifiers der Eltern-Policy, der momentan der entsprechende *compound name* des CORBA Objekts auf dem ORB ist.

string[] getChildrenID()
Ein Array der IDs aller Kind-Policies ist das Ergebnis der Ausführung dieser Operation. Es wird ein 0-elementiges Array zurückgegeben, falls keine Kind-Policies spezifiziert worden sind.

boolean addChildID(string PolID)
Eine neue Kind-Policy-ID wird diesem Policy Object hinzugefügt. Alle IDs werden intern in einer Liste gehalten.

boolean removeChildID(string PolID)
Die ID *PolID* wird aus der Kinder-ID-Liste entfernt. Diese Operation wird in der Regel von der Kind-Policy selber, bei deren vollständigen Entfernung durch Ausführung der Operation *destroyPolicy()* der *PolicyFactory*, aufgerufen.

```
module policy {
  enum PolicyModality { Obligation, Authorization };
  exception BNFNotOK {string reason};
  exception BindToORBfailed { string reason;};
  exception PolicyNotFound {string reason;};
  exception ActivationFailed {string reason;};
  exception DeactivationFailed {string reason;};

  interface PolicyObject {
    string getID();
    string getParentID();
    void setParentID(in string PolID);
    IDlist getchildrenID();
    boolean addChildID(in string PolID);
    boolean removeChildID(in string PolID); };

  typedef sequence<PolicyObject> PolicyObjectlist;
};
```

Abbildung 4-18: IDL Spezifikation des Policy Objects

In Abbildung 4-19 sind die IDL Spezifikationen der Interfaces der strategischen, zielorientierten und operationalen Policy dargestellt, deren Operationen im nachfolgenden nacheinander erläutert werden. Da sowohl die strategische als auch die zielorientierte Policy nur den Entwicklungsprozeß zur operationalen Policy widerspiegeln, spielen diese bisher im Gesamtsystem eher eine untergeordnete Rolle, so daß auch die in deren IDL Interface spezifizierten Operationen sich auf das Zurückliefern ihrer festgelegten Attribute beschränken. Denkbar wäre hier als zukünftige Erweiterung des Policy Service, eine Strategie für die automatische

Überführung der zielgerichteten Policy in operationale Policies, zu entwickeln. Dies würde sich u.a. in einer Erweiterung der IDL Schnittstellen äußern.

Das *StrategicPolicy*-Interface weist die folgende zusätzliche Operation in ihrer Schnittstelle auf:

string getDescription()
Es wird die in englischer oder deutscher Prosa verfaßte strategische Policy zurückgeliefert.

Im *GoalorientedPolicy*-Interface wurden die folgenden Operationen spezifiziert, die sich nach der in Kapitel 3 festgelegten Schablone für zielorientierte Policies richtet:

string getName()
Das Ergebnis dieser Operation ist ein bedeutungsvollerer Name der zielgerichteten Policy. Dieser unterscheidet sich von dem Identifier der Policy insofern, daß dieser vom Policy-Ersteller festgesetzt wird und deswegen nicht eindeutig sein muß. Es wird jedoch erhofft, daß durch eine aussagekräftigere Bedeutung dieses Namens, im Gegensatz zu der puren Ziffernfolge des Identifiers, eine intuitivere Suche nach einer Policy möglich wird.

PolicyModality getModality()
Durch Ausführung dieser Operation wird festgestellt, ob es sich bei der vorliegenden zielorientierten Policy um eine Verpflichtungs- oder Ermächtigungspolicy handelt.

string getSubject()
Es wird der spezifizierte Subjekttyp zurückgegeben.

string getSDomain()
Der Domänenausdruck für das Subjekt wird zurückgeliefert.

string getTarget()
Wie bei der entsprechenden Operation für das Subjekt, kann in diesem Fall der Targettyp festgestellt werden.

String getTDomain()
Die bei der Instanziierung dieser zielgerichteten Policy festgelegte Domäne des Targets wird zurückgegeben.

string getAction()
Diese Operation liefert den Aktionsbereich der zielorientierten Policy zurück.

string getConstraint()
Mit dieser Operation kann die Bedingung für die Anwendbarkeit der Policy festgestellt werden.

Das *OperationalPolicy*-Interface enthält die folgenden Operationen:

string getName()
Es wird der spezifizierte Name der operationalen Policy zurückgegeben. Dieser Name unterscheidet sich von dem Identifier, der mit *getID()* eingeholt werden kann, insofern, daß dieser im Gesamtsystem nicht eindeutig sein muß und eher mnemonisch ist. Das Namensattribut der operationalen Policy wird erst nach der ersten Aktivierung gesetzt, da der Name in der PDL-Beschreibung der Policy enthalten ist und diese erst beim Parserdurchlauf dekodiert wird. Ansonsten wird '*not set*' zurückgegeben.

string getDescription()
Die PDL-Beschreibung der operationalen Policy wird zurückgeliefert.

void checkPDL()
Mit dem Aufruf dieser Operation kann die Syntax der in PDL verfaßten operationalen Policy überprüft werden. Wird ein Syntax-Fehler in der Beschreibung festgestellt, so wir dieser innerhalb der geworfenen Exception zurückgeliefert.

void activate(string EOFactoryID, string agentInitCh)
Durch Aufruf dieser Operation wird die operationale Policy aktiviert, indem die in PDL verfaßte Beschreibung dem *Policy Interpreter* übergeben wird. Der darin enthaltene Parser interpretiert den Inhalt und kreiert die erforderlichen *Enforcement Objects* (*EO*), die die operationale Policy tatsächlich durchsetzen. Die EOs werden in der durch *EOFactoryID* angegebenen *Enforcement Factory* produziert. Bei EO-FactoryID handelt es sich, wie bei allen anderen CORBA Objekten auch, um den eindeutigen *compound name* der Enforcement Factory in der Form des erweiterten Domänenbezeichners. Da nur der augenblickliche Netzzustand mit Hilfe des *Topology Service* bzw. *Naming Service* von dem Policy Parser in Betracht gezogen werden kann, bindet sich zusätzlich die aktive operationale Policy an den Event Channel *agentInitCh*, bei dem u.a. sich alle neu hinzukommenden Managementagenten anmelden. Ist der Neuankömmling für die aktive operationale Policy von Bedeutung, so wird ein entsprechendes EO kreiert. Intern werden von der operationalen Policy die IDs der kreierten EOs in einer Liste gehalten. Um diese Liste immer auf dem aktuellen Stand zu halten, wird beim Abmelden eines Managementagenten überprüft, ob für diesen ein EO kreiert wurde. Fällt diese Überprüfung positiv aus, so wird die ID des EOs aus der Liste gelöscht.

void deactivate()
Eine bereits aktivierte operationale Policy wird deaktiviert. Es werden alle Schritte die während der Aktivierung durchgeführt wurden, rückgängig gemacht. Dies bedeutet, daß alle kreierten EOs entfernt werden sowie die operationale Policy vom *AgentInitChannel* getrennt wird.

boolean isActive()
Es wird überprüft, ob sich die operationale Policy im aktiven Zustand befindet.

string getEOFactoryID()
Der *compound name* der *Enforcement Object Factory*, in der die EOs bei der Aktivierung dieser Policy kreiert wurden, wird in der Form des erweiterten Domänenbezeichners zurückgegeben. Falls die operationale Policy noch nicht aktiviert worden ist, so erhält der Aufrufer '*not set*' zurück.

string getAgentInitCh()
Der *compound name* des *AgentInitChannels*, an den sich die Policy bei deren Aktivierung gebunden hat, wird in der Form des erweiterten Domänenbezeichners zurückgegeben. Falls die operationale Policy noch nicht aktiviert worden ist, so erhält der Aufrufer '*not set*'.

TargetListElement[] getTargetObjects()
Die während der Aktivierung der Policy kreierten EOs und die dazugehörigen festgestellten Zielagenten, an die sich jeweils ein EO gebunden hat, werden geordnet innerhalb eines Arrays zurückgeliefert. Intern wird eine Liste der entsprechenden Paare gehalten, um bei sich abmeldenden Agenten überprüfen zu können, ob ein EO für

diesen kreiert worden ist und dieser entsprechend aus der Liste gestrichen werden muß. Ist die Policy noch nicht aktiviert worden oder wurden aufgrund fehlender aktiver Agenten keine EOs kreiert, so ist ein 0-elementiges Array Ergebnis dieser Operation.

```
module policy {
 interface StrategicPolicy : PolicyObject {
   string getDescription(); };

 interface GoalorientedPolicy : PolicyObject {
   string getName();
   PolicyModality getModality();
   string getSubject();
   string getSDomain();
   string getTarget();
   string getTDomain();
   string getAction();
   string getConstraint(); };

 struct TargetListElement { string EnfObjID;
                            string AgentID; };

 typedef sequence<TargetListElement> TargetList;

 interface OperationalPolicy : PolicyObject {
   string getName();
   string getDescription();
   void checkPDL()
         raises (BNFNotOK);
   void activate(in string EOFactoryID, in string agentInitCh)
         raises (ActivationFailed);
   void deactivate()
         raises (DeactivationFailed);
   boolean isActive();
   string getEOFactoryID();
   string getAgentInitChID();
   TargetList getTargetObjects(); };
};
```

Abbildung 4-19: IDL Spezifikation der verschiedenen Policytypen

Die IDL Spezifikation der *Policy Factory* weist v.a. Operationen zum Kreieren und Löschen von Policies auf. Intern werden die Objektreferenzen aller kreierten Policies innerhalb einer *java-Hashtable* geführt. Während des Hochfahrens des *PFMA* werden die Zustände aller Policies aus einer angegebenen Datenbank des *Persistence Service*, wiederhergestellt. Wird der PFMA wieder heruntergefahren, so werden die Zustände aller in der internen Hashtable gehaltenen Policies in derselben Datenbank des Persistence Service aktualisiert und gespeichert.

Die von CORBA Clients verfügbaren Operationen des IDL Interfaces der Policy Factory, sind in Abbildung 4-19 dargestellt und werden im folgenden detailliert beschrieben:

StrategicPolicy createStratPol(string PolDescr)

Eine strategische Policy, mit der in Prosa verfaßten Beschreibung, wird neu angelegt. Es wird ein eindeutiger Identifier für diese Policy errechnet und zugewiesen, der anschließend mit *getID()* abgefragt werden kann. Mit diesem ID wird die strategische Policy als CORBA Objekt dem ORB bekannt gemacht und ist von anderen CORBA Clients durch dessen IDL Interface ansprechbar. Zusätzlich wird die CORBA Objektreferenz der internen Hashtable hinzugefügt. Um bei einem Absturz des *PFMA* alle Policies wiederherstellen zu können, werden sofort nach deren Kreierung diese in der allokierten DB des Persistence Service gespeichert. Als PID des dafür kreierten *Persistent Objects* wird der compound name der Policy genommen.

GoalorientedPolicy createGoalPol(string pName, PolicyModality pModality, pSubject, pSDomain, pTarget, pTDomain, pAction, pConstraint)

Bei Aufruf dieser Operation wird mit den übergebenen Attributen eine zielgerichtete Policy geschaffen. Es werden dieselben Schritte wie bei der strategischen Policy unternommen: ID berechnen; Objektreferenz in Hashtable ablegen; persistent in der allokierten DB des Persistence Service anlegen.

OperationalPolicy createOperPol(string PolDescr)

Diese Operation liefert mit der Beschreibung der Policy in PDL, eine operationale Policy zurück. Es werden dieselben Schritte wie bei der strategischen und operationalen Policy durchgeführt. Nach der Kreierung der Policy bietet sich dem Aufrufer dieser Operation die Möglichkeit, mit *checkPDL()* die Syntax der operationalen Policy zu überprüfen.

boolean destroyPolicy(string aPolicyID)

Die Policy mit dem gegebenen Identifier *aPolicyID* wird vollständig beseitigt. Dies bedeutet, daß die Policy vom ORB, aus der DB des Persistence Service und schließlich aus der internen Hashtable entfernt wird. Falls es sich dabei um eine aktivierte operationale Policy handelt, so werden ebenfalls alle kreierten EOs zerstört. Besitzt die Policy *Kinder*, so werden auch diese vollständig entfernt. Ist auch ein entsprechender Eintrag für die *Eltern* vorhanden, so wird die ID dieser Policy aus der Kinder-ID-Liste der Eltern entfernt.

PolicyObject getPolicy(string policyID)

Die Policy mit dem gegebenen Identifier wird zurückgegeben. Es ist selbstverständlich auch möglich, die entsprechende Policy über den *Naming Service* anzusprechen. Oft erweist sich aber dieser Weg als schneller, da die interne Hashtable der Policy Factory nur die Objektreferenzen der Policy Objekte verwaltet und nicht aller sich auf dem ORB befindenden Objekte, wie es beim Naming Service der Fall ist.

PolicyObject[] listAllPolicies()

Die Objektreferenzen aller sich derzeit in der internen Hashtable befindenden Policy Objects, werden innerhalb eines Arrays zurückgeliefert. Dies beinhaltet sowohl die Objekte, die während des Hochfahrens des Agenten wiederhergestellt worden sind als auch die Objekte, die seitdem kreiert worden sind.

```
module policy {
  interface PolicyFactory {
    StrategicPolicy createStratPol (in string PolDescr)
                    raises (BindToORBfailed);
    GoalorientedPolicy createGoalPol (in string pName,
                              in PolicyModality pModality,
                              in string pSubject,
                              in string pSDomain,
                              in string pTarget,
                              in string pTDomain,
                              in string pAction,
                              in string pConstraint)
                          raises (BindToORBfailed);
    OperationalPolicy createOperPol (in string PolDescr)
                       raises (BindToORBfailed);
    boolean destroyPolicy(in string aPolicyID)
             raises (PolicyNotFound, DeactivationFailed);
    PolicyObject getPolicy(in string policyID)
                 raises (PolicyNotFound);
    PolicyObjectlist listAllPolicies ()
                     raises (PolicyNotFound);};
};
```

Abbildung 4-20: IDL Spezifikation der Policy Factory

4.3.4 Der Policy Enforcement Service

Die angebotene Funktionalität des *Policy Enforcement Service* wird i.d.R. vom *Policy Service* durch Aktivierung einer operationalen Policy in Anspruch genommen. In Abbildung 4-21 sind die Bestandteile des Policy Enforcement Service zu sehen, der sich, wie die übrigen Komponenten der MA auch, in eine *Enforcement Object Factory* (*EOF*) und den von dieser Factory produzierten *Enforcement Objects* (*EO*) teilt. Beide Bestandteile werden jeweils auf dem *Enforcement Object Factory Mobile Agent* (*EOFMA*) instanziiert. Jeder EOFMA ist dabei einem bestimmten Managementagenttypen (z.B. DHCP-Agent) zugeordnet, für den der EOFMA ausschließlich EOs produziert. Beim Hochfahren des Agenten werden zunächst aus einer gegebenen DB des *Persistence Service* (*PS*) alle darin enthaltenen aktiven operationalen Policies dahingehend überprüft, ob deren Spezifikation auch *Zielobjekte* des der EOFMA zugewiesenen Managementagenttypen enthält. Fällt die Überprüfung positiv aus, so werden die benötigten EOs kreiert. Dadurch wird einerseits die grundsätzliche Unabhängigkeit des Policy Enforcement Service vom Policy Service etabliert und andererseits wird mit dieser Strategie die Konsistenz des MS gewahrt. Als Alternative zu diesem Vorgehen wäre das separate persistente Anlegen der EOs denkbar, um die Überprüfung jeder einzelnen Policy zu vermeiden und ein schnelleres Hochfahren des Agenten durch einfaches Wiederherstellen der EOs zu ermöglichen.

Dabei ergeben sich jedoch erhebliche Konsistenzprobleme, wenn von dem zusätzlichen Speicheraufwand zunächst abgesehen wird:

> Ist zum Zeitpunkt der Deaktivierung einer operationalen Policy der entsprechende EOFMA nicht erreichbar, weil dieser in der Zwischenzeit heruntergefahren wurde, so werden die bei der Aktivierung kreierten und nun persistenten EOs nicht beseitigt. Um also die Konsistenz zu wahren, müßte der EOFMA auf jeden Fall immer während des Startvorgangs überprüfen, welche Policies noch aktiv sind. Da also diese Überprüfung, um die Konsistenz zu wahren, nicht umgangen werden kann, wird auf die separate Speicherung der EOs verzichtet.

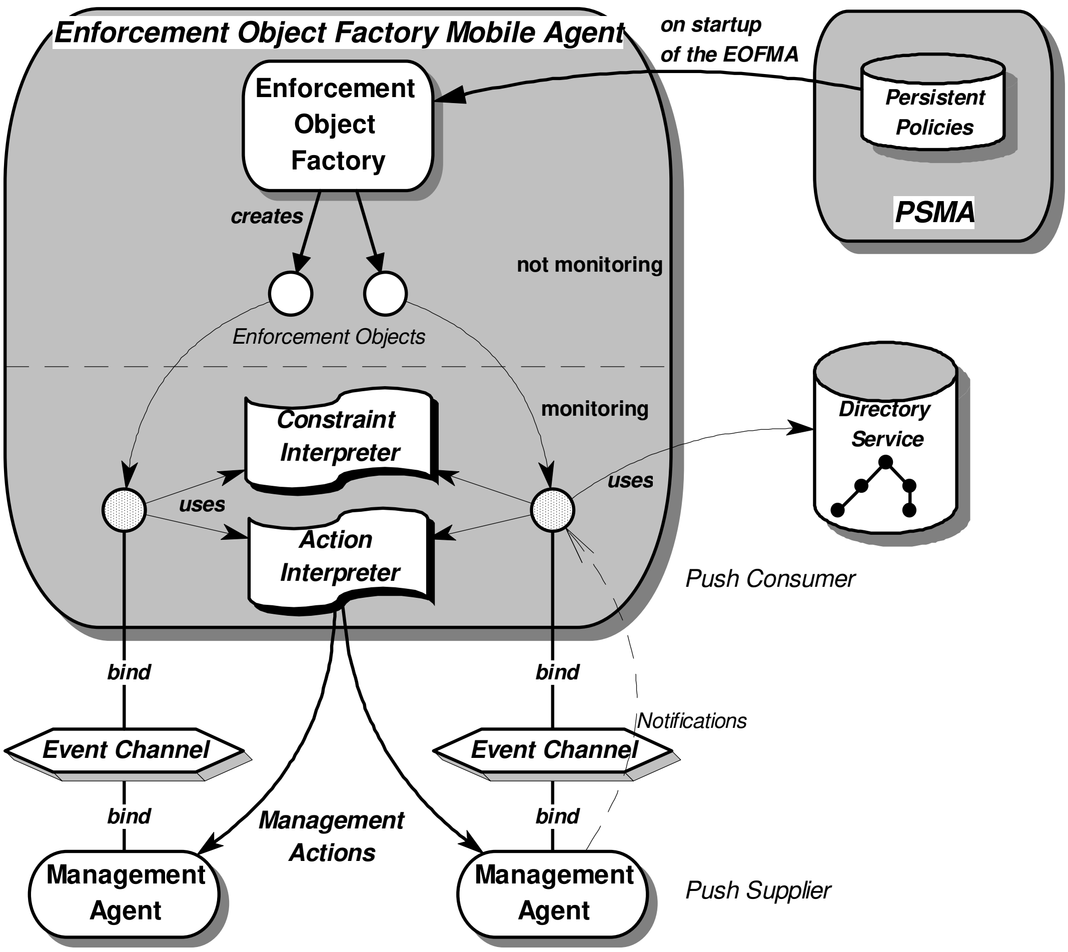

Abbildung 4-21: Die Komponenten des Policy Enforcement Service

Sowohl die Enforcement Object Factory als auch die Enforcement Objects sind als CORBA Server realisiert worden, wobei die IDL Spezifikationen deren nach außen hin sichtbaren Schnittstellen im folgenden eingehend erklärt werden.

In Abbildung 4-22 ist das spezifizierte IDL Interface des Enforcement Objects dargestellt, das aus den folgenden Operationen besteht:

boolean startMonitoring()

Durch Aufruf dieser Operation bindet sich das Enforcement Object an den Event Channel des Targetobjekts. Ist ein Event aufgetreten, von dem das EO durch eine Notification des Targetobjekts in Kenntnis gesetzt wird, so wird zunächst der Verursacher, im vorliegenden Fall also das nomadische System, des Events festgestellt, dessen ID ein Bestandteil der erhaltenen Notification ist. Durch Nutzung eines *Directory Services* wird die Heimatdomäne des NoS auf Übereinstimmung mit der Subjektspezifizierung der Ursprungspolicy geprüft. Fällt die Überprüfung positiv aus, so wird der weitere Inhalt der Notification zusammen mit den in der Ursprungspolicy spezifizierten Bedingungen dem *Constraint Interpreter* zur Auswertung übergeben. Liefert der Constraint Interpreter ein positives Ergebnis zurück, so werden die Managementaktionen durch Aufruf des *Action Interpreters* gefeuert. Mit Hilfe des Action Interpreters ist es v.a. möglich, Bedingungen zwischen den einzelnen Managementaktionen durch *if-then-else*-Konstruktionen[5] festzusetzen. Scheitert die Ausführung einer Managementaktion daran, daß das entsprechende Zielobjekt nicht erreichbar ist, so wird an dieser Stelle abgebrochen und versucht, alle vorhergehenden Aktionen rückgängig zu machen[6].

boolean stopMonitoring()

Das EO trennt sich von dem zuvor angebundenen Event Channel. Damit wird das Monitoring des dem EO zugewiesenen Zielobjekt unterbrochen.

string getSourcePolID()

Es wird der eindeutige Identifier der operationalen Quellpolicy zurückgegeben.

domainData getSubjectDomain()

Die Subjektbeschreibung wird in Form der Darstellung der Subjektdomäne zurückgegeben, wobei auch die Festlegung auf genau ein Subjekt möglich ist. Momentan besteht die Möglichkeit, eine Domäne in *TQL* oder in Form eines erweiterten Domänenbezeichners bzgl. des *Naming Service* zu formulieren.

string getEventChannelID

Der Name bzw. eindeutige Identifier des Event Channels, an den das EO sich momentan gebunden hat, wird zurückgeliefert. Befindet sich das EO gerade im Nicht-Monitoring Zustand, so wird '*not set*' zurückgegeben.

string getTargetAgentID()

Der Name des Targets in Form des erweiterten Domänenbezeichners, der von diesem EO beobachtet werden soll, ist das Ergebnis dieser Operation.

---

[5] Diese Art der Festlegung von Bedingungen zwischen den Aktionen ist momentan im vorliegenden Prototypen noch nicht vorgesehen, jedoch ist die Implementierung des *Action Interpreters* für derartige Erweiterung bewußt offen gehalten worden.

[6] Auf Grund der nicht trivialen Aufgabe der allgemeinen Zustandswiederherstellung im Netz, ist dies ebenfalls in der vorliegenden Prototyp-Implementierung noch nicht realisiert worden; statt dessen wird nur die Ausführung weiterer Mgmt.-aktionen unterbunden.

string getConstraint()
Das Constraint, das der PDL-Syntax entspricht und bei Auftreten eines Events evaluiert wird, indem es dem *Constraint Interpreter* übergeben wird, wird zurückgeliefert.

string listAllOperations()
Das Ergebnis dieser Operation ist die Spezifikation, der bei Auftreten eines Events auszuführenden Managementaktionen gemäß der PDL-Syntax.

string getID()
Der eindeutige Name bzw. Identifier des EOs wird in Form des erweiterten Domänenbezeichners zurückgeliefert.

```
module enforcement {
  exception ORBBindFailed { string reason;};
  exception EONotFound {string reason;};

  enum domainType { NS, TQL };
  struct domainData {
          string domainExpr;
          domainType kind; };

  interface EnforcementObject {
    boolean startMonitoring();
    boolean stopMonitoring();
    string getSourcePolID();
    domainData getSubjectDomain();
    string getEventChannelID();
    string getTargetAgentID();
    string getConstraint();
    string listAllOperations();
    string getID(); };

  typedef sequence<EnforcementObject> EnfObjList;
};
```

Abbildung 4-22: IDL Spezifikation des Enforcement Objects

In Abbildung 4-23 sind die Operationen der Enforcement Object Factory (EOF) notiert und nachstehend erklärt:

EnforcementObject createEnforceObject(string srcPolID, string AgentID, domainData subjectDomain, string constraint, string actions)
Diese Operation wird i.d.R. im *Policy Interpreter* des *Policy Service*, während des Aktivierungsprozesses einer operationalen Policy, aufgerufen. Während des Parsevorgangs der in *PDL* formulierten operationalen Policy, werden die für das Enforcement Object relevanten Informationen festgestellt. Sowohl der *Action*- als auch der *Constraint*-Teil der Policy-Beschreibung werden direkt übernommen und dem EO übergeben, der diese, zum gegebenen Zeitpunkt bei Eintreffen eines Events, mit Hil-

fe des *Constraint Interpreters* und des *Action Interpreters* auswertet. Während der Kreierung des EOs wird ebenfalls überprüft, ob nicht bereits ein, mit identischen Attributen belegtes EO existiert. Fällt diese Überprüfung positiv[7] aus, so wird eine Referenz auf dieses EO zurückgeliefert. Ansonsten wird sofort nach der Kreierung des EOs mit der Beobachtung des Targets begonnen, indem die *startMonitoring()*-Operation des EOs aufgerufen wird. Intern verwaltet die EOF alle kreierten EOs in einer Liste, so daß die jeweils neu hinzukommenden EOs dieser Liste hinzugefügt werden.

void destroyEnforceObject(string aEnfObjID)

Das EO mit dem ID *aEnfObjID* wird vollständig vom ORB und aus der EOF-internen Liste entfernt. Diese Operation wird i.d.R. bei der Deaktivierung einer operationalen Policy aufgerufen.

EnforcementObject[] listAllEnfObj()

Alle sich gegenwärtig in der internen Liste der EOF befindenden EOs werden innerhalb eines Arrays zurückgegeben.

EnforcementObject getEnfObj(string EnfObjID)

Eine Referenz auf das EO mit dem ID *EnfObjID* wird zurückgegeben. Es ist selbstverständlich ebenso möglich das entsprechende EO direkt über den *Naming Service* anzusprechen. Oft erweist sich aber dieser Weg als schneller, da die interne Liste der EOF nur die Objektreferenzen der EOs verwaltet und nicht aller sich auf dem ORB befindenden Objekte, wie es beim Naming Service der Fall ist.

```
module enforcement {
 interface EnforcementObjectFactory {
   EnforcementObject createEnforceObject( in string
                          srcPolID, in string AgentID,
                          in domainData subjectDomain, in
                          string constraint, in string actions)
        raises (ORBBindFailed);
   void destroyEnforceObject(in string aEnfObjID)
        raises (EONotFound, DestroyingFailed);
   EnfObjList listAllEnfObj();
   EnforcementObject getEnfObj(in string EnfObjID)
        raises (EONotFound); };
 };
```

Abbildung 4-23: IDL Spezifikation der Enforcement Object Factory

[7] Dies ist insbesondere immer dann der Fall, wenn ein PFMA in der Zwischenzeit heruntergefahren wird und bei dessen Hochfahren, insbesondere während der Wiederherstellung der aktiven operationalen Policies, versucht wird, die dynamischen Attributfelder, wie die EO-ID-Liste, durch vollständige Neukreierung der EOs zu aktualisieren.

4.3.5 Der Directory Service

Wie schon eingangs im Überblick festgestellt worden ist, wird eine statische Repräsentierung der Domänenaufteilung einer Netzstruktur benötigt, um jeweils die Heimatdomäne der NoS feststellen und somit den Anfragen der EOs bezüglich der Subjektspezifizierung Rechnung tragen zu können. Sowohl der *CORBA Naming Service* als auch der *Topology Service* kommen v.a. für die Darstellung der *momentanen* Netzsituation in dem zu managenden Bereich in Frage und sind dementsprechend für diesbezügliche Anfragen ausgelegt. Für die Modellierung der Heimatdomänen der nomadischen Systeme eignet sich statt dessen eine einfache (relationale) Datenbank jedoch besser, in der neben dem eindeutigen Identifikator des NoS, wie z.B. die MAC-Adresse, die dazugehörige Heimatdomäne eingetragen wird. Probleme entstehen bei dieser Lösung jedoch durch den daraus resultierenden Anspruch, daß diese Datenbank das vollständige Wissen über alle Vorkommnisse von nomadischen Systemen in einem Unternehmensnetz haben muß. Hierbei ergibt sich neben der Notwendigkeit der zentralen Bereitstellung dieser Datenbank, um die Konsistenz zu wahren, auch die Schwierigkeit den Datenbestand immer auf dem aktuellen Stand zu halten. Während dies bei überschaubaren Netzen mit kleinerem Umfang keine Konflikte bereitet, führt dies bei Netzen die sich über mehr als einen Standort spannen, unweigerlich zu oben benannten Problemen.

Als Ausweg ist der Einsatz eines *Directory Services* zu empfehlen. Ein Directory Service ist immer dann einer Datenbank vorzuziehen, wenn damit gerechnet werden kann, daß um ein Vielfaches mehr Anfragen als Updates stattfinden werden. Die in einem Directory enthaltene Information wird in der *Directory Information Base (DIB)* gehalten und ist baumartig im sog. *Directory Information Tree (DIT)* strukturiert und v.a. in Hinblick auf Domänenstrukturierungen innerhalb von Netzen ausgerichtet. Die Information wird jeweils von einem Directory Server aufbereitet, an den Clients Anfragen, durch Nutzung eines *Directory Access Protocols (DAP)* wie *LDAP* [RFC2251], schicken können. Der entscheidende Vorteil ergibt sich aber durch die Möglichkeit, ein Directory durch Verbinden mehrerer Directory Server darzustellen. Kann ein Server die Anfrage eines Clients nicht beantworten, so schickt er diese an den nächsten Server. Dies wird solange wiederholt, bis die Anfrage beantwortet oder endgültig nicht aufgelöst werden kann[8]. Folglich läßt sich damit das oben beschriebene Problem lösen. Jede selbstverwaltete Domäne erhält einen Directory Server und sorgt sich um die Aktualität der darin enthaltenen Information. Desweiteren werden alle Directory Server geeignet miteinander verbunden, um die oben beschriebene Anfrage zu ermöglichen. Auch muß die Information den Gegebenheiten entsprechend im DIT modelliert werden. Da der naming graph des Naming Service dem DIT ähnlich ist, empfiehlt es sich, die Modellierung beider Systeme anzupassen. Auf tieferes Eingehen in die Materie wird an dieser Stelle verzichtet und statt dessen auf die einschlägige Literatur verwiesen [X.500]. Parallel zu dieser Arbeit wurde am Lehrstuhl ebenfalls im Rahmen einer Diplomarbeit [Gars98] das Management von Directory Services untersucht.

Trotz bestehender Standardisierungen von Java APIs zur Nutzung von Directory Services ([JNDI97], [JSPI97]) wurde, aufgrund fehlender Implementierungen dieser APIs, innerhalb des MS auf eine einfache relationale DB zurückgegriffen. Dabei kam JMAPI zum Einsatz, das eine API zur Verfügung stellt, die v.a. auf das Netz- und Systemmanagement ausgerichtet ist. Der Vorteil hierbei ergab sich durch die einfache Modellierung der Objekte und Generierung einer Standardoberfläche zur Eingabe der Daten. Die dadurch erstellte Datenbank ist

[8] Neben dieser Strategie der Anfragebeantwortung, werden noch zahlreiche andere angeboten, wie z.B. *Chaining* oder *Multicasting*, die im weiteren unbetrachtet bleiben.

bewußt durch ihren provisorischen Charakter einfach gehalten und enthält zwei String-Felder: das erste Feld enthält die MAC-Adresse des NoS, mit dem eine eindeutige Identifizierung des Systems ermöglicht wird und das zweite Feld beinhaltet einen Domänenausdruck aus dem Wortbereich <XFN_IDENTIFIER>. Die EOs suchen bei Auftreten eines Events mit der erhaltenen MAC-Adresse, die Bestandteil jeder von den Agenten versendeten Notifications an das MS ist, den korrespondierenden Domäneneintrag in der DB. Dieser Wert wird daraufhin mit der Subjektbeschreibung der spezifizierten Quellpolicy verglichen. Wird eine Übereinstimmung festgestellt, d.h. daß das NoS von der Policy betroffen ist, so wird im EO die Ausführung durch die Evaluierung des Constraints fortgesetzt.

4.3.6 Konventionen innerhalb des MS

Innerhalb des MS mußten, v.a. in Hinblick auf die Namensgebung der Objekte, gewisse Festlegungen getroffen werden, damit die korrekte Funktionsweise der Komponenten der MA gewährleistet werden kann. Diese Konventionen werden nun im folgenden dargestellt:

Objektnamen

Alle kreierten Objekte, dazu zählen selbstverständlich auch die kreierten Event Channels, werden durch Nutzung des *Naming Service* an einen Namen gebunden. In Kapitel 4.2.3.1 wurde bereits erklärt, daß ein *name component* aus den Attributen *identifier* und *kind* besteht. Der identifier wird immer zur Identifizierung des *name components* benützt, während das **kind-Attribut** folgende Belegungen haben kann:

domain	Diese Wertebelegung sagt aus, daß es sich bei diesem *name component* um einen Domänenbezeichnung handelt und nicht ein CORBA Objekt identifiziert. Dies schließt ein, daß ein *name component* nie ein *binding* zu einem echten CORBA Objekt haben kann, sondern immer auf einen *naming context* verweist. Wenn man sich das Beispiel des *naming graph* vor Augen führt, so stellt ein *naming component* mit *domain*-Kennzeichnung immer die inneren Kanten des Graphen dar und endet nie in einem Blatt des Graphen.

Die folgenden Belegungen stellen immer den Typen eines Objekts dar, wobei der dazugehörige *identifier* immer den tatsächlichen Namen eines Objekts repräsentiert. Ein *name component* mit einem der folgenden Belegungen für das kind-Attribut ist immer an ein echtes Objekt gebunden. Im dazugehörigen *naming graph* führt dieser *name component* immer zu einem Blatt. So ist die Identifizierung des Objekttyps schon anhand des Namens möglich:

eo	Enforcement Object
eof	Enforcement Object Factory
pf	Policy Factory
sp	Strategic Policy
gp	Goaloriented Policy
op	Operational Policy
ps	Persistence Service
dhcp_agent	DHCP-Managementagent
pcswitch_agent	PCSwitch-Managementagent

Es ist natürlich ohne weiteres möglich, diese Liste, v.a. bezüglich der Agenten, analog zu erweitern und neuen Bedürfnissen anzupassen.

PID der Persistent Objects
Als PID des Persistent Objects wird immer der *compound name* des CORBA Objekts verwendet. Dies hat den Vorteil, daß sowohl eine intuitivere Suche in der *Persistent Object Database* möglich ist als auch daß schon anhand des kind-Attributs festgestellt werden kann, welche Art von Objekt in serialisierter Darstellung vorliegt. Dies ist v.a. bei dem Wiederherstellungsprozeß der EOs durch den EOFMA nützlich, um die *Operational Policies* von den *Goaloriented* und *Strategic Policies* zu unterscheiden.

Managementagenttypen
Wie in Kapitel 4.3.4 schon erwähnt worden ist, wird eine EOF genau einem Agententypen zugeordnet, für dessen Instanzen ausschließlich EOs kreiert werden. Um dies zu gewährleisten, wird bei der Instanziierung des EOF dieser Agententyp mit einem, im vorhergehenden Abschnitt spezifizierten Kürzel, wie z.B. *dhcp_agent*, angegeben.

Domänenaufbau mit dem Naming Service
Bevor das Managementsystem das erste Mal in Betrieb genommen werden kann, muß zunächst eine Strukturierung des von dem MS zu managenden Netzbereichs durch Spezifizierung von Domänenbereichen durchgeführt werden. Dies wird mit Hilfe des *Naming Service* realisiert, mit dem einmalig diese statische Struktur aufgebaut wird. Allen daraufhin neu kreierten Objekten werden Namen relativ zu diesem dadurch erstellten *naming graph* zugewiesen. Insbesondere die *Factories* der einzelnen Services müssen demzufolge Kenntnis von dieser Struktur haben, um den neu produzierten Objekten die korrekten Namen zuteilen zu können. Diese Information wird den Factories bei deren Instanziierung übergeben.

4.4 Spezifikation der Managementschnittstellen der Agenten

In Kapitel 4.3 wurden die Komponenten der Managementanwendung sowie deren Schnittstellen beschrieben. Damit das Managementsystem wie in der Anforderungsanalyse in Kapitel 4.1 realisierbar ist, muß die Schnittstelle zwischen Anwendung und den Managementagenten festgelegt werden, um eine Kommunikation zwischen diesen Parteien überhaupt erst zu ermöglichen. Die Spezifikation dieser Schnittstelle wird nun im folgenden vorgestellt.

In Abbildung 4-24 ist die von einem Agenten zu unterstützende Schnittstelle dargestellt, die grundsätzlich aus den nachstehenden drei Teilen besteht:

CORBA IDL Interface des Agenten
Der Agent implementiert ein CORBA IDL Interface, in dem Operationen angeboten werden, mit denen von außen auf den Zustand und das Verhalten des Agenten gegenüber den tatsächlichen MOs Einfluß genommen werden kann. Diese Schnittstelle ist, abhängig vom Agententypen, immer unterschiedlich. Trotzdem müssen eine Rei-

he von Operationen definiert werden, die eine Abfrage von grundlegenden Daten ermöglichen und von jedem Agenten angeboten werden müssen.

Event Channel des Agenten

Der Agent kreiert einen eigenen Event Channel, an den sich die Managementanwendung via eines EOs bindet, um von Zustandsänderungen des Agenten durch Empfangen der entsprechenden Notifications zu erfahren. Die gesendeten Notifications sind jeweils in Form eines *StructuredEvents* verpackt.

Well-known Event Channel des MS

Damit das Managementsystem von neu hinzukommenden sowie von sich entfernenden Agenten erfährt, sendet jeder Agent jeweils beim Hoch- und Herunterfahren ein festgelegtes Datenpaket (wieder in Form eines *StructuredEvents*) an einen well-known Event Channel des MS, der im weiteren mit *AgentInitChannel* bezeichnet wird.

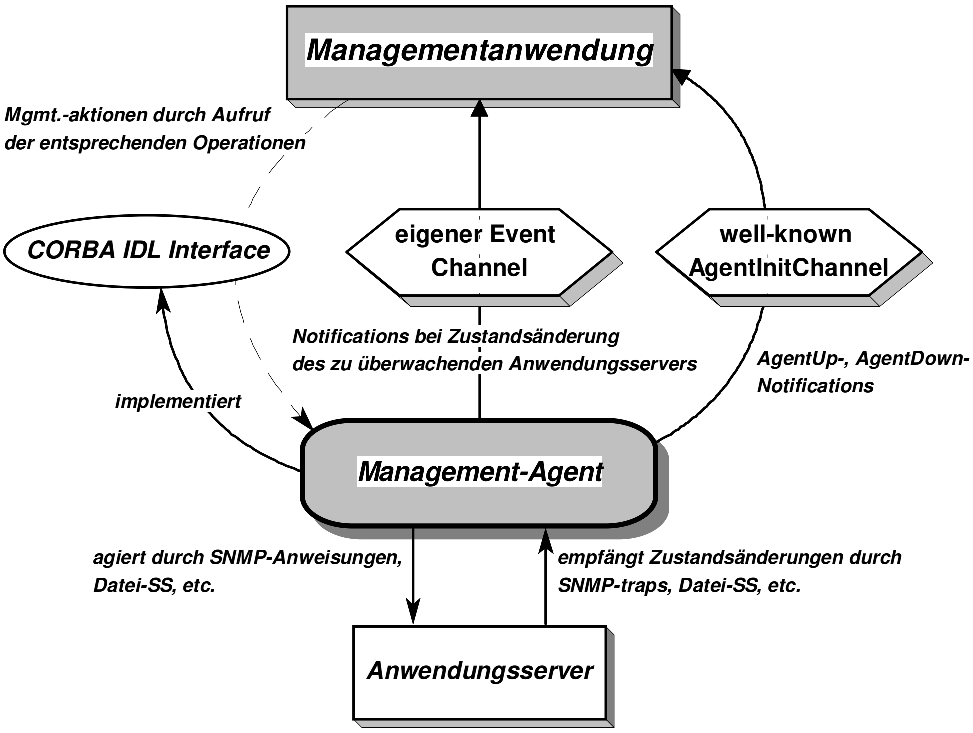

Abbildung 4-24: Überblick der Agentenschnittstelle

In Abbildung 4-25 ist die Spezifikation des Interfaces in IDL dargestellt, die alle Agenten unterstützen müssen. Erreicht wird diese Vorgabe, indem die jeweils individuell festgelegten Schnittstellen der Agenten zwingend von diesem IDL Interface erben müssen. Im folgenden wird das erwartete Verhalten der Operationen erörtert:

string getID()

Es wird der *compound name* des Agenten in Form des erweiterten Domänenbezeichners zurückgeliefert.

string getEventChName()

Der *compound name* des von dem Agenten kreierten Event Channels ist das Ergebnis dieser Operation. Bei Auftreten eines internen Ereignisses, wie z.B. einer Zustandsänderung des MOs, wird eine Notification an diesen Channel gesendet.

string getType()

Der Agententyp wird ausdrücklich zurückgeliefert, auch wenn dieser im Namen des Agenten durch das *kind*-Attribut bereits kodiert ist.

Es ist durchaus denkbar, daß bei einer vorliegenden Implementierung des *Notification Services*, die Agenten nicht mehr einen eigenen Event Channel kreieren, sondern sich an bestehende, im Vorfeld festgelegte Channels binden. Die Consumer, in diesem Fall also u.a. auch die Managementanwendung, könnten sich durch entsprechende Filtereinstellungen für die Events bestimmter Agenten abonnieren.

```
module mo {
  interface NoCSMgmtAgent {
    string getID();
    string getEventChName();
    string getType(); };
};
```

Abbildung 4-25: IDL-Basisinterface eines Managementagenten

Im weiterem gilt es noch die Struktur der von den Agenten gesendeten *StructuredEvents* festzulegen. In Abbildung 4-26 ist der Aufbau eines Events, der jeweils beim **Hoch- bzw. Herunterfahren** des Agenten in den *AgentInitChannel* gesendet wird, veranschaulicht. Folgende zwingende Festlegungen bezüglich der gesendeten Notification wurden getroffen:

- der *Event Header* besteht lediglich aus dem *Fixed Header* und hat für dessen Felder die folgenden Belegungen:

  domain_type: *Systemmanagement*

  event_type: *AgentUp*, falls der Agent hochgefahren wurde und
  AgentDown, falls der Agent heruntergefahren wird

  event_name: Dieses Feld ist nicht von Bedeutung und wird deswegen frei gehalten

- der *Filterable Body* besteht aus den folgenden Attributnamen und den dazugehörigen Belegungstypen:

  agent_type : In diesem *String*-Feld wird der Typ des Agenten angegeben, so daß z.B. der Policy Service bei neu hinzukommenden Agenten entscheiden kann, ob der Agent von einer aktiven operationalen Policy betroffen ist.

  agent_name : Dieses *String*-Feld enthält den *compound name* des Agenten in Form des erweiterten Domänenbezeichners.

  event_channel_name : Der *compound name* des Event Channels, an den sich der Agent gebunden hat bzw. den dieser kreiert hat, um seine Notifications bei Zustandsänderungen abzuschicken, wird in diesem *String*-Feld ebenfalls in Form des erweiterten Domänenbezeichners angeführt.

Zu diesen drei obligatorischen Feldern, können von den Agenten noch zusätzliche definiert werden. Hierbei muß lediglich die Einschränkung beachtet werden, daß nur Belegungen des Typs *string*, *long* und *boolean* erlaubt sind.

- der *Remaining Body* kann von dem Agenten ebenfalls belegt werden, sollte aber i.d.R. freigehalten werden

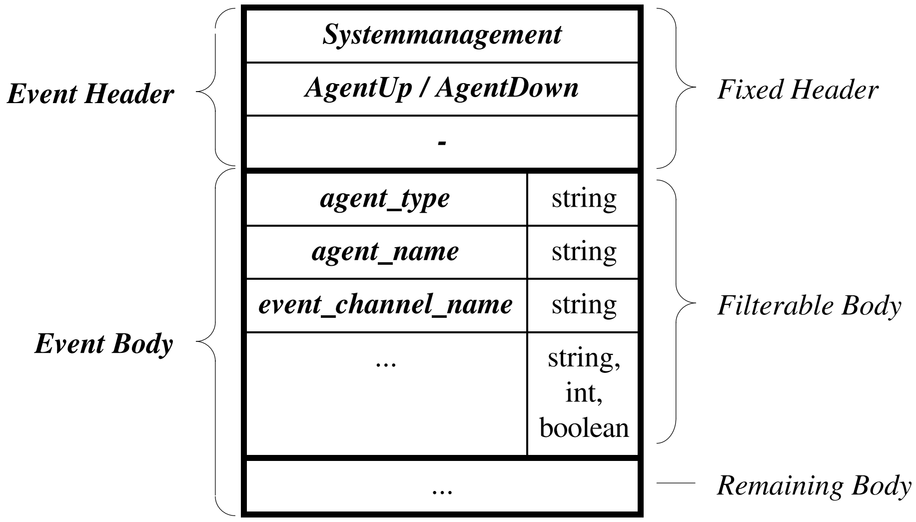

Abbildung 4-26: Die AgentUp/AgentDown-Eventstruktur

In Abbildung 4-27 ist die Struktur eines Events dargestellt, der bei Auftreten individueller Ereignisse in den Agenten-eigenen Channel geschickt wird. Folgende Auflagen muß jeder Event erfüllen:

- Der *Event Header* besteht wieder nur aus dem *Fixed Header*, der die folgenden Eigenschaften aufweist:

  domain_type: *Systemmanagement*

  event_type: An dieser Stelle wird der Eventtyp angegeben, wobei Belegungen wie z.B. *NewRequest*, *NewNodeAttached*, etc. denkbar wären, um anzuzeigen, daß der Wunsch einer Ressourcennutzung seitens eines NoS aufgetreten ist.

  event_name: Dieses Feld ist momentan irrelevant und sollte nicht belegt werden.

- Der *Filterable Body* muß die folgenden zwei Felder enthalten:

  agent_name: An dieser Stelle wird der *compound name* des Agenten in Form des erweiterten Domänenbezeichners als *String* angegeben; dies ist v.a. dann relevant, wenn sich mehrere Agenten an genau einen Event Channel binden, um die Quelle des erhaltenen Events festzustellen.

  hardware_address: Dieses Feld enthält die MAC-Adresse[9] des nomadischen Systems, das den Ressourcenzugriff getätigt hat.

[9] momentan werden die nomadischen Systeme anhand der MAC-Adresse eindeutig identifiziert

Abhängig von der Art des Events können und sollten weitere Felder definiert werden, wobei auch hier die Enschränkung gilt, daß nur Belegungen des Typs *string*, *long* und *boolean* erlaubt sind. Ebenfalls ist die Belegung des *Remaining Body* freigestellt.

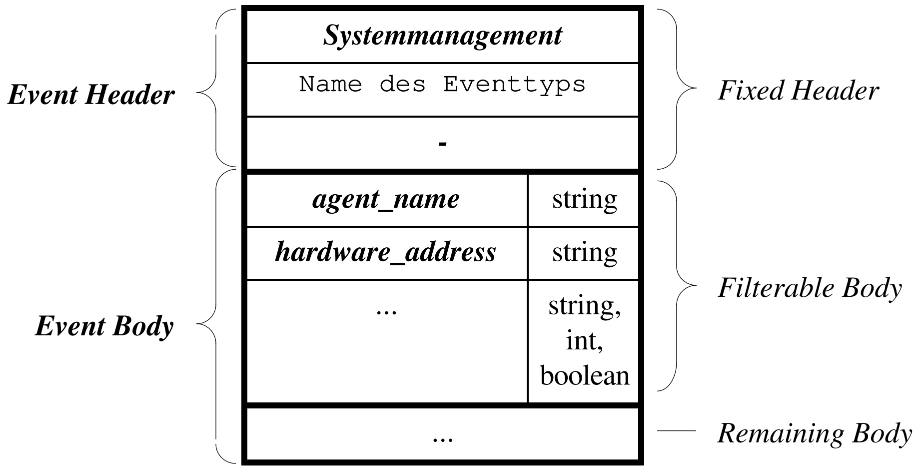

Abbildung 4-27: Die Eventstruktur der Managementagenten

Während der Entwicklung der hier vorgestellten Managementanwendung, wurde parallel dazu ein Managementagent für PC-Switches [Allg98] und für DHCP Server [Demm98] implementiert, die sich nach diesen hier vorgestellten Vorgaben richten und somit eine Zusammenarbeit mit dem MS ermöglichen. Im folgenden Kapitel wird das Zusammenspiel zwischen diesen Agenten und dem MS anhand von Diagrammen veranschaulicht.

4.5 Typischer Managementablauf

In diesem Kapitel wird anhand eines einfachen Beispiels ein typischer Managementablauf mit Hilfe von Diagrammen veranschaulicht. Als Beispiel dient das Management eines Layer-2-Switchs.

Es wird eine vereinfachte Version des ersten Beispiels aus Abschnitt 3.7 als operationale Beispielpolicy herangezogen, der umgangssprachlich so lautet:

Öffne den Port des Switchs, falls das neu hinzukommende Endgerät bekannt ist.

Es wird angenommen, daß die Ports eines Switches default-mäßig abgeschaltet sind und nur durch explizite Managementanweisungen geöffnet werden. Weiterhin gilt für den vorliegenden Switch-Managementagenten, daß dieser in die Kernelsoftware des tatsächlichen Switchs integriert ist. Die operationale Policy ist aktiviert, so daß das entsprechende *Enforcement Object* bereits kreiert wurde und sich an den Event Channel des Managementagenten gebunden hat.

In Abbildung 4-28 sind die einzelnen Stationen des Ablaufs anhand eines Flußdiagramms in UML-Notation dargestellt und lauten wie folgt:

(1) Ein nomadisches System schließt sich an einen freien Port des Switchs an.

(2) Die Software des Switchs, und damit auch dessen Managementagent, bemerkt den Neuanschluß.

(3) Der Managementagent schickt das Event *NewNodeAttached* in den eigenen CORBA Event Channel per *push*-Methode.

(4) Das *Enforcement Object*, daß die obige Policy durchsetzt, erhält das Event und beginnt daraufhin mit der Auswertung des Subjektbereichs und des Constraints.

(5) Fällt die Auswertung positiv aus, so werden die spezifizierten Operationen ausgeführt; in diesem Fall handelt es sich um die Operation *openPort()*, die an der CORBA Schnittstelle des Switch Managementagenten aufgerufen wird.

(6) Der Managementagent führt daraufhin die Schritte innerhalb der Kernelsoftware durch, um den Port zu öffnen.

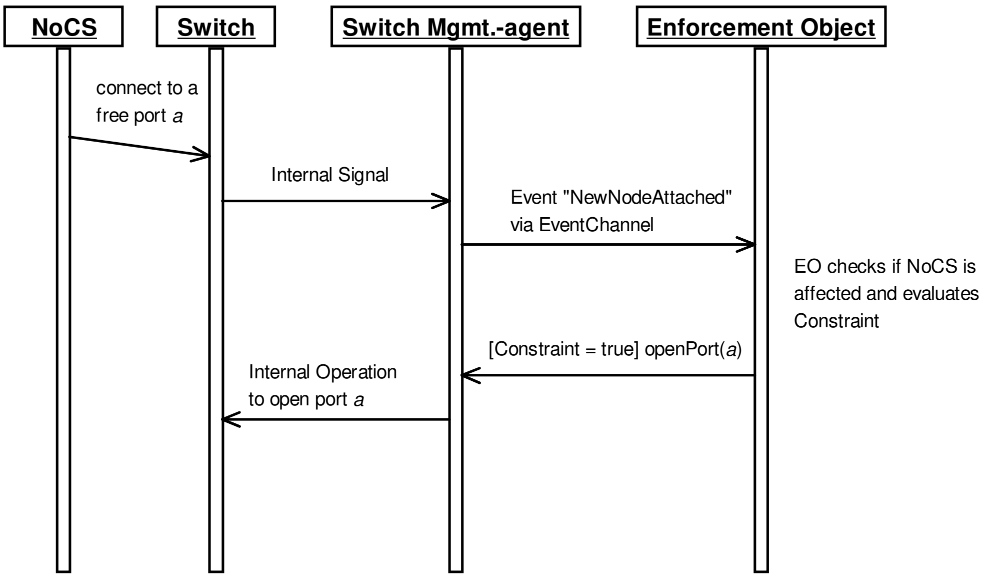

Abbildung 4-28: Flußdiagramm des Managementablaufs

Nach diesem Gesamtüberblick des Managementablaufs, wird in Abbildung 29 zusätzlich, anhand des Zustandsdiagramms (ebenfalls in UML Notation) eines *Enforcement Objects*, der Kommunikationsablauf innerhalb der verteilten Managementanwendung veranschaulicht:

(1) Das EO erhält ein Event durch den Aufruf der eigenen *push()*-Methode.

(2) Die Information innerhalb des Events, bei dem es sich immer ausschließlich um ein *StructuredEvent* handelt, wird extrahiert.

(3) Die mitgelieferte MAC-Adresse des Endgeräts führt zu mehreren Anfragen beim Verzeichnisdienst, der die Heimatdomäne des Endgeräts zurückliefert, sowie beim *Naming* oder *Topology Service*.

(4) Entspricht die Heimatdomäne nicht der spezifizierten Subjektdomäne, so wird an dieser Stelle abgebrochen.

(5) Es wird geprüft, ob bereits ein CORBA Objekt kreiert wurde, der den Zustand des Endgeräts repräsentiert; falls nicht, wird die Kreierung angestoßen.

(6) Das Constraint wird mit Hilfe des *Constraint Interpreters* ausgewertet; dies kann u.U. zu Operationsaufrufen innerhalb der CORBA Umgebung kommen, die durch Nutzung des *Dynamic Invocation Interfaces(DII)* realisiert werden.

(7) Bei positiver Auswertung werden die spezifizierten Operationen gefeuert, indem der *Action Interpreter* zum Einsatz kommt; die Operationen werden mit Hilfe des *DII* ausgeführt, das auch eine Anfrage an die *Interface Repository* beinhaltet.

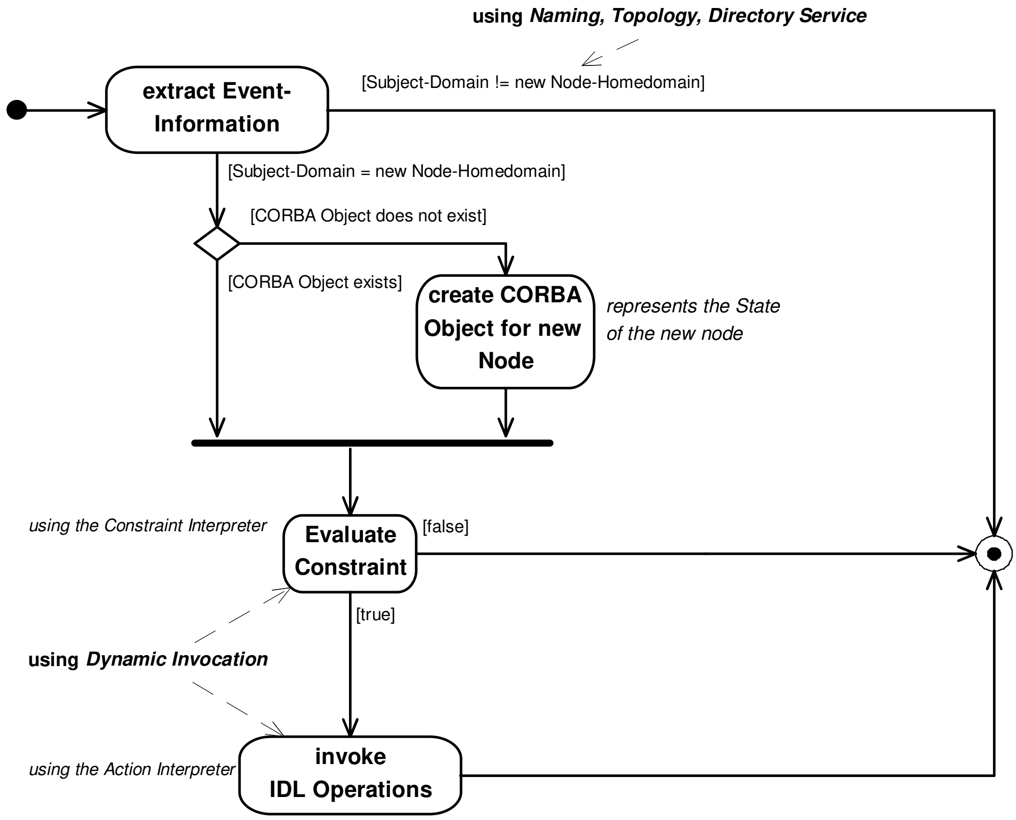

Abbildung 4-29: Zustandsdiagramm eines EOs

4.6 Zusammenfassung

In diesem Kapitel wurde die entwickelte Managementarchitektur sowie deren Umsetzung in einer CORBA Umgebung vorgestellt. CORBA stellte sich durch seinen standardisierten Status sowie den vielen standardisierten *CORBA Services*, als besonders geeignete Middleware für Managementzwecke heraus. Nach einer ersten Anforderungsanalyse zeigte sich das Bedürfnis nach einer hohen Skalierbarkeit und Ausfallsicherheit der einzelnen Komponenten der MA, das zu einer Integration von zusammengehörenden Funktionen in *Services* der Managementanwendung führte. Dies ergab als Folge den *Policy Service*, den *Policy Enforcement Service* und den *Persistence Service*, die als CORBA Server Objekte ihre Funktionalität anderen Komponenten der MA anbieten. Um jedoch die geforderte Gesamtfunktionalität der Anwendung zu erfüllen, wurden ebenfalls die bereits standardisierten CORBA Services *Naming Service, Topology Service* und der *Event Service* herangezogen. Die Managementanwendung ist durch diesen Aufbau ohne weiteres durch Hinzufügen neuer Komponenten erweiterbar. Um die Ausfallsicherheit der implementierten Services der Anwendung zu erhöhen, wurden diese als *Mobile Agents* realisiert, deren Konzept und Umsetzung der MASIF-Spezifikation ([OMG97c], [Kemp98]) entspricht. Dadurch ergibt sich die Möglichkeit die *Service*-Instanzen während des laufenden Betriebs an eine andere Stelle des Netzes zu migrieren, um z.B. die Lastverteilung im Netz auszugleichen.

Ebenfalls wurde in diesem Kapitel die Schnittstelle der Managementagenten festgelegt, die erst eine erfolgreiche Zusammenarbeit mit der Managementanwendung ermöglicht und somit eine Voraussetzung für alle zukünftig zu entwickelten Agenten darstellt.

Zum Schluß dieses Kapitels erfolgte die Beschreibung eines typischen Managementablaufs mit den teilnehmenden Komponenten der MA, um u.a. die Funktionsweise der Anwendung anhand von Diagrammen zu erläutern.

Kapitel 5
Das Objektmodell

Im vorhergehenden Kapitel wurde die Managementarchitektur basierend auf CORBA vorgestellt. Darin wurden die Komponenten der MA, deren nach außen hin sichtbaren IDL Interfaces sowie deren Zusammenspiel untereinander u.a. in Form von Diagrammen und Abbildungen veranschaulicht. Dieses Kapitel dient der Illustration der tatsächlich *implementierten* Objekte innerhalb der Managementanwendung. Anhand von Klassendiagrammen werden die Beziehungen zwischen den Objektklassen, wie z.B. eine Vererbungshierarchie, erläutert. I.d.R. wird bei der Realisierung einer verteilten Anwendung bzw. einer Software allgemein, die folgende Vorgehensweise gewählt: zuerst werden die Anforderungen bestimmt, um daraufhin ein Objektmodell zu generieren, das im nächsten Schritt durch die Implementierung in einer Programmiersprache umgesetzt wird. Hier fließen selbstverständlich auch Methoden des *Software-Engineerings* (siehe hierzu auch [Hagg96]) ein, die jeweils nach jedem vollendeten Schritt eine Validierung der vorhergehenden Schritte vorsehen, so daß z.B. während der Implementierung oftmals das Objektmodell angepaßt und verbessert wird, da neue Einflußfaktoren wie Programmiersprache und gewählte Middleware, zunächst nicht berücksichtigt werden können. Dies heißt insbesondere auch, daß ein Objektmodell grundsätzlich unabhängig von der später eingesetzten Programmiersprache sein sollte. Davon wird aber in den nun folgenden Abschnitten abgesehen und somit ein niedrigerer Abstraktionsgrad gewählt. Dieses Kapitel soll v.a. der Dokumentation der erstellten Software dienen, um das Verstehen der Struktur der verteilten Anwendung zu erleichtern. Dies ist v.a. in Hinblick auf spätere Verbesserungen und Erweiterungen durch Dritte vorteilhaft.

5.1 Einführung in OMT

Ein Objektmodell ist die *graphische Darstellung* von Objekten und deren Beziehungen untereinander. Um die Möglichkeit zu haben, einerseits nicht nur den Modell-Erstellern sondern auch unbeteiligten Dritten das Verstehen des Objektmodells zu ermöglichen und andererseits auch das Modell in einen konkreten Programmrahmen einer gewählten Implementierungssprache *automatisch* umsetzen zu können, ist eine eindeutige *Notation* für die Modelldarstellung erforderlich. Diese Forderung ist durch die *Object Modeling Technique (OMT)* erfüllt, die nicht nur *Diagrammtypen*, *Objekttypen* und *Beziehungen* in Form eines Regelrahmens festlegt, sondern auch konkrete Techniken zur Modell- und Diagrammerstellung beinhaltet. Nachstehend werden die Basissymbole von OMT kurz erklärt, um auch dem OMT-unerfahrenen Leser das Verständnis der in den nächsten Abschnitten folgenden Diagrammen zu ermöglichen. Auf einen vertiefenden Einblick in OMT wird jedoch an dieser Stelle verzichtet und auf die einschlägige Literatur ([Aoni96], [Aoni96a]) verwiesen. Es werden jedoch im folgenden Grundkenntnisse der Objektorientierung (OO) vorausgesetzt.

Ein *Modell* ist die meist graphische Darstellung eines (Software-)Systems in Form von *Diagrammen*. Je komplexer ein zu modellierendes System ist desto höher ist die Anzahl der benötigten Diagramme, um das System vollständig von dem Modell zu erfassen. Ebenso reicht meist ein Sichtwinkel auf das betrachtete System nicht aus, um es komplett beschreiben zu können. In OMT werden aus diesem Grund verschiedene *Diagrammtypen* zur Modellierung angeboten, die jeweils verschiedene Sichtweisen auf ein System ermöglichen:

- das *Objektmodelldiagramm* (auch *Klassendiagramm* genannt), für die statische Darstellung der verschiedenen Objektklassen und deren Beziehungen untereinander (wie z.B. Vererbungs- und Enthaltenseinshierarchien)
- das *Dynamische Modelldiagramm* (auch *Zustandsdiagramm* genannt), in dem die zeitliche Dimension der wesentliche Faktor ist und durch die verschiedenen Zustände eines Objekts in Form eines endlichen Automaten visualisiert wird
- das *Funktionale Modelldiagramm*, in dem das Hauptaugenmerk auf die Informations- und Datenflüsse zwischen den Objekten gerichtet ist
- das *Use Case Diagramm*, in dem verschiedene Szenarios in Interaktion mit dem zu modellierenden System veranschaulicht werden
- das *Interaktionsdiagramm*, indem die Abfolge einzelner Interaktionsschritte zwischen den Objekten modelliert wird

In den nun folgenden Abschnitten wird nur das Klassendiagramm zur Modellierung der implementierten Komponenten der MA eingesetzt. Es werden deswegen nachstehend nur die verwendeten Elemente und Symbole des Klassendiagramms erläutert (siehe hierzu auch Abbildung 5-1):

- Eine *Klasse* wird als Rechteck dargestellt, das in drei Felder geteilt ist. Das erste Feld enthält den *Namen* der Klasse, das zweite die *Attribute* und das dritte die *Operationen* der Klasse. Nur das Anzeigen des Namens einer Klasse ist zwingend.
- Eine *Beziehung (Assoziation, Relation)* zwischen Klassen wird durch eine Linie, die die Klassen miteinander verbindet, visualisiert. Um die Art der Beziehung anzuzeigen, werden entweder Symbole, wie bei der Vererbungs- und Enthaltenseinshierarchie, oder Linienbeschriftungen verwendet.
- Eine *Vererbungshierarchie* (*generische Beziehung*) zwischen Klassen wird durch ein *gleichschenkliges Dreieck* angezeigt. Die Dreiecksspitze zeigt auf die *Oberklasse (Superklasse)*, die an die *Unterklasse (Subklasse)* ihre Attribute und Operationen vererbt.
- Eine *Enthaltenseinshierarchie* wird durch eine *Raute* visualisiert. Die Raute ist mit der Klasse verbunden, die eine andere Klasse enthält bzw. die aus anderen Klassen zusammengesetzt wird. Diese Art der Beziehung läßt sich auch dadurch beschreiben, indem die „zusammengesetzte“ Klasse die jeweils andere Klasse als Attribut im Klassenrechteck enthält.

- Die Assoziationen zwischen den Objekten werden immer von *links nach rechts* gelesen, es sei denn, daß durch Pfeile eine andere Richtung angezeigt wird.

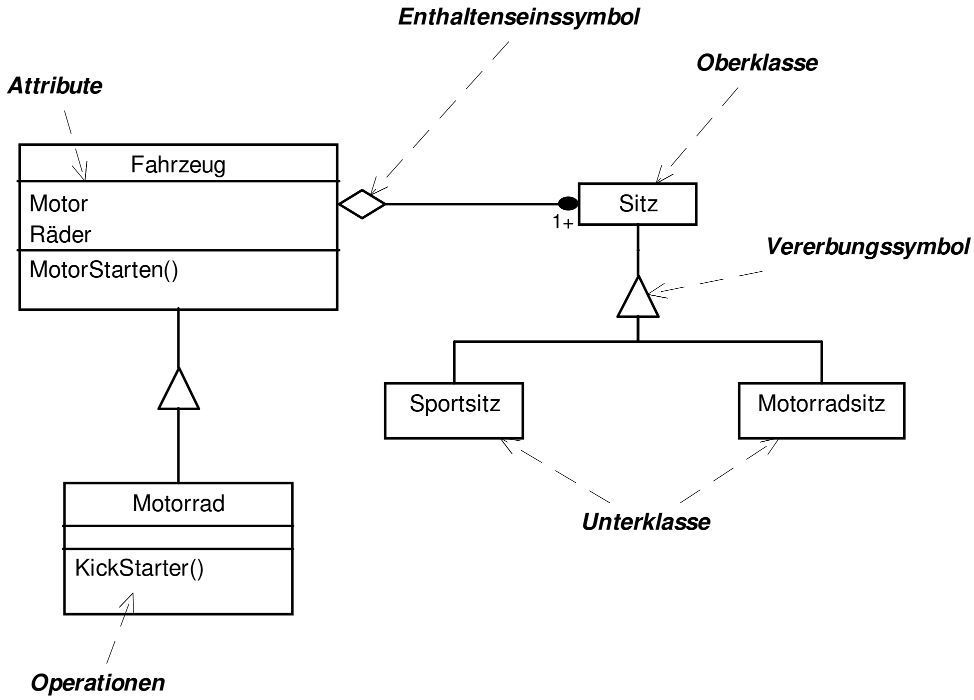

Abbildung 5-1: Die Elemente und Symbole des Klassendiagramms

Dieser kurze Einblick in die Notation und Semantik von OMT-Klassendiagrammen ist als ausreichend zu bewerten, um die in den nächsten Abschnitten dargestellten Diagramme zu verstehen.

5.2 Das Gesamtobjektmodell

Ein Überblick der an der verteilten Managementanwendung beteiligten Objekte, ist in Abbildung 5-2 zu sehen. Bei den hier und im folgenden dargestellten Klassen sowie deren Beziehungen, fließt bereits sowohl der Einfluß der eingesetzten Programmiersprache *Java* als auch die eingesetzte *CORBA*-Implementierung *Visibroker 3.0* von *Visigenic* ein. Implementierungsgesichtspunkte bezüglich Java und dem Visibroker werden gesondert in Kapitel 6 betrachtet. Zum besseren Verständnis werden aber bereits in Abschnitt 5.3 einige Erläuterungen diesbezüglich vorgezogen.

Die in Kapitel 4 erarbeiteten *Services* der Anwendung sind folgendermaßen im Objektmodell wiederzufinden:

- *Policy Service*: Realisierung durch die *PolicyFactoryMobileAgent*-Klasse und der *PolicyObjectImpl*-Klasse (und deren Unterklassen)

- *Policy Enforcement Service*: Realisierung durch die *EnforcementObjectFactoryMobileAgent*-Klasse und der *EnforcementObjectImpl*-Klasse
- *Persistence Service:* Realisierung durch die *PersistenceServiceMobileAgent*-Klasse, der *PersistentObjectFactory*-Klasse und der *PersistentObject*-Klasse

Ebenfalls sind in diesem Klassendiagramm die Rollen notiert, die einzelne Objekte gegenüber anderen Objekten einnehmen, um in Interaktion mit diesen zu treten. Das *PolicyFactoryMobileAgent*-Objekt und das *EnforcementObjectFactoryMobileAgent*-Objekt agieren gegenüber dem *PersistenceServiceMobileAgent*-Objekt jeweils als CORBA Client. Das *OperationalPolicyImpl*-Objekt agiert hingegen gegenüber dem *EnforcementObjectFactoryMobileAgent*-Objekt als CORBA Client, um Objekte der Klasse *EnforcementObjectImpl* zu kreieren.

In den nun folgenden Abschnitten werden die Klassen der hier dargestellten Objekte gesondert betrachtet und erläutert.

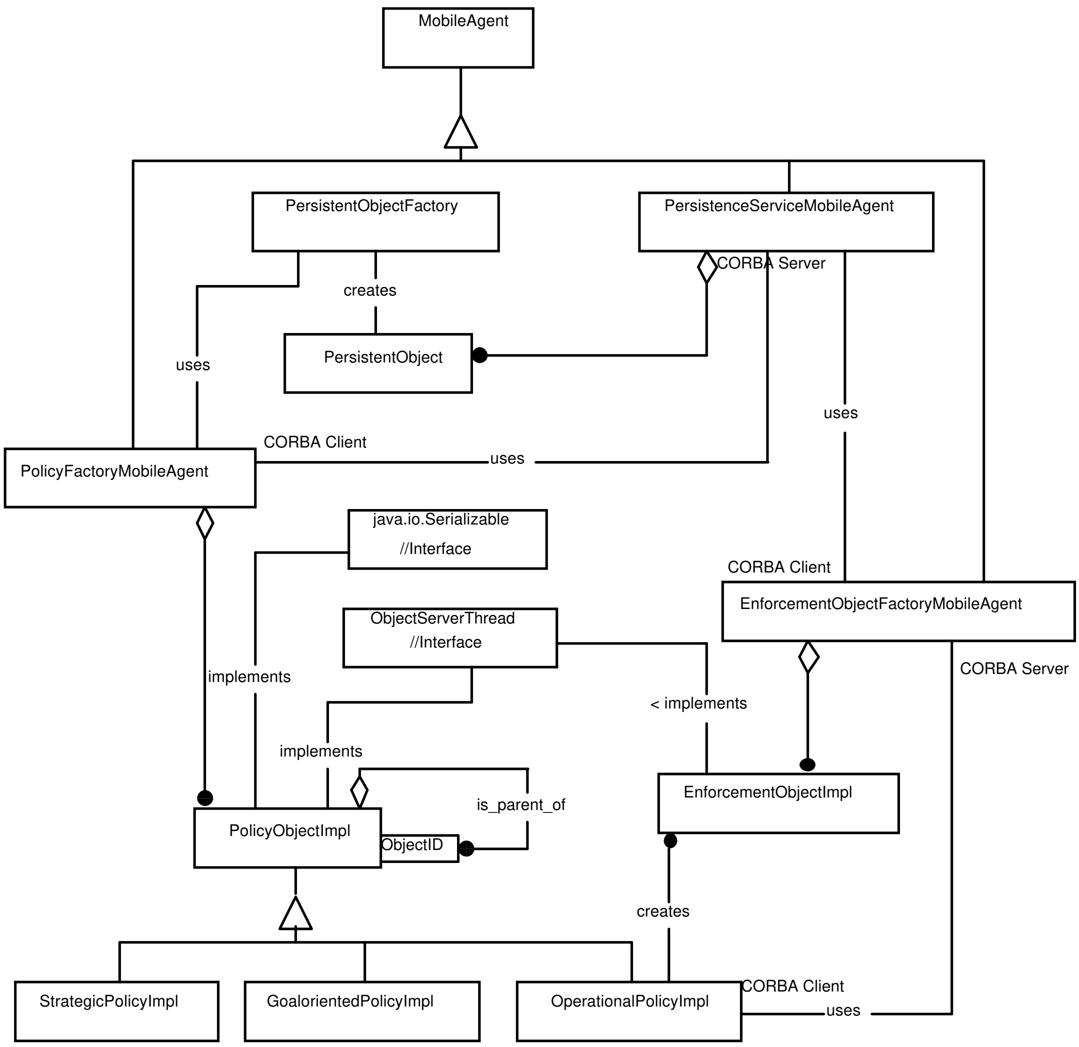

Abbildung 5-2: Das Gesamtobjektmodell

5.3 Allgemeine Klassen und Interfaces

Dieser Abschnitt enthält diejenigen Klassen und Interfaces, die wiederholt in Beziehung zu den Klassen der einzelnen *Services* stehen. Diese werden im folgenden genau erklärt und treten in den darauffolgenden Klassendiagrammen nur mit dem Klassennamen auf. Desweiteren erfolgen bereits an dieser Stelle aus Gründen der Übersichtlichkeit, einige Hinweise zur Implementierung der Objekte. Die Merkmale und Elemente der eingesetzten Programmiersprache *Java* werden aber erst in Kapitel 6 ausführlich betrachtet.

5.3.1 Die MobileAgent-Klasse

Wie schon in Abschnitt 4.3 erläutert wurde, sind die einzelnen entwickelten *Services* der Managementanwendung als *Mobile Agents* realisiert worden, um die Fähigkeit der Migration innerhalb eines Netzwerks zu haben. Die in Abbildung 5-3 veranschaulichte Struktur ist ein Teil der Umsetzung der MASIF-Spezifikation, die in einer Diplomarbeit [Kemp98] am Lehrstuhl erarbeitet wurde. Damit eine beliebige Implementierung, die als eigenständiger Agent laufen soll, die Vorzüge eines *Mobile Agents* annimmt, müssen folgende Punkte erfüllt werden:

- Die Objektklasse muß von der Klasse *MobileAgent* erben.
- Das Objekt-eigene IDL-Interface muß von dem IDL-Interface *Migration* erben.
- Der Klassenname des Agenten setzt sich aus dem IDL-Interfacenamen und dem Postfix „MobileAgent" zusammen.
- Die Operation *cleanUp()* muß von dem Agenten implementiert werden, das zur Ressourcenfreigabe dient, wenn der Agent heruntergefahren bzw. auf ein anderes Agentensystem migriert wird.

Die Tatsache, daß das Agenten-eigene IDL-Interface vom Interface *Migration* erben muß, impliziert, daß die Agenten als CORBA Server realisiert werden müssen. Desweiteren muß dieses Interface nicht tatsächlich von der Agentenklasse implementiert werden, da bereits deren Oberklasse *MobileAgent* eine Implementierung aufweist.

Die Oberklasse *Agent* der Klasse *MobileAgent* implementiert sowohl das Interface *java.lang.Runnable*, das die Realisierung der Agenten als *Threads* anzeigt, als auch das Interface *java.io.Serializable*, das die Serialisierbarkeit eines Agenten signalisiert. Erst diese Tatsache ermöglicht die Migration der mobilen Agenten zwischen verschiedenen Agentensystemen. I.d.R. muß während der Implementierung des tatsächlichen Agenten (im Klassendiagramm mit *myInterfaceMobileAgent* bezeichnet) nichts weiter beachtet werden, da u.a. die Serialisierung von Objekten in *Java* automatisch durchgeführt wird. Enthält die Klasse des Agenten jedoch Attribute, die selber nicht serialisierbar sind, so muß diesen Attributen entweder das *Java*-Schlüsselwort *transient* vorangestellt werden oder die Klasse stellt zwei eigene Methoden *readObject()* und *writeObject()* zur Verfügung, die die tatsächliche Serialisierung und Deserialisierung des Objekts übernehmen. Tritt dieser Fall ein, so wird dies in den Subklassen von *MobileAgent* im weiteren durch Notierung dieser beiden Methoden angezeigt. Eine eigene *run()*-Methode muß auch nur dann implementiert werden, falls während des Startvorgangs agentenspezifische Schritte durchgeführt werden müssen. Auch dies wird im folgenden durch eine Notierung dieser Methode in der jeweiligen Klasse angezeigt.

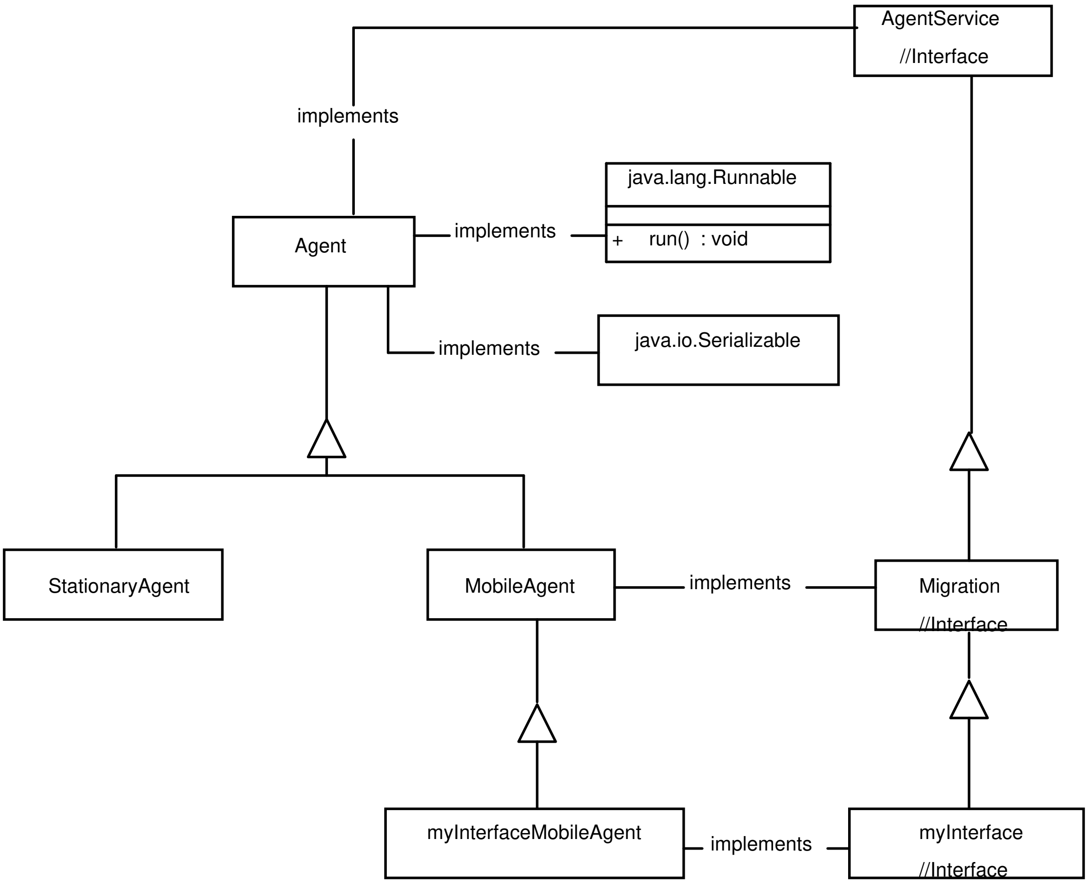

Abbildung 5-3: Das MobileAgent-Klassendiagramm

5.3.2 Das ObjectServerThread-Interface

Das Interface *ObjectServerThread* wird von allen Klassen derjenigen CORBA Server Objekte implementiert, die nicht als Agenten realisiert werden. Dies betrifft im Rahmen dieser Arbeit die *ManagedObjectImpl*-Klasse, EnforcementObjectImpl-Klasse, die *StrategicPolicyImpl*-Klasse, die *GoalorientedPolicyImpl*-Klasse und die *OperationalPolicyImpl*-Klasse. Dieses Interface dient v.a. zum einheitlichen Zugriff auf CORBA-relevante Daten. Nachstehend werden die Operationen des Interfaces erläutert:

boolean startThread(): Diese Methode bindet das Objekt an den ORB, indem es einen neuen *Thread* erzeugt.

boolean stopThread(): Diese Methode entfernt das Objekt wieder vom ORB, indem der entsprechende *Thread* angehalten wird.

boolean is_up(): Es wird überprüft, ob das Objekt an den ORB gebunden ist und somit für andere CORBA Objekte zugänglich ist.

Thread getThreadRef(): Es wird eine Referenz auf den Thread zurückgeliefert, der während der *startThread()*-Methode kreiert wurde.

String getORBObjRef(): Die CORBA-Referenz des Objekts auf dem ORB wird in Form eines Strings zurückgeliefert. Mit der ORB-Methode *string_to_Object()* kann der String wieder in eine echte CORBA-Objektreferenz umgewandelt werden.

void run(): Diese Methode wird vom Standard-Interface *java.lang.Runnable* geerbt und muß ebenfalls implementiert werden. Die *run()*-Methode wird immer dann aufgerufen, wenn ein Thread kreiert wird, also insbesondere immer dann, wenn die *startThread()*-Methode aufgerufen wird. Konkret werden in der *run()*-Methode diejenigen Schritte durchgeführt, die das Objekt an den ORB binden. Zusätzlich können weitere Maßnahmen, die das Objekt betreffen, durchgeführt werden, wie z.B. daß das Objekt zusätzlich an einen Namen gebunden wird, indem der *Naming Service* verwendet wird.

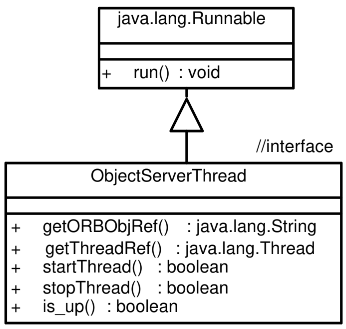

Abbildung 5-4: Das ObjectServerThread-Klassendiagramm

5.3.3 Die ManagedObjectImpl-Klasse

Die Klasse *ManagedObjectImpl* implementiert das IDL-Interface *ManagedObject* indem es von der Klasse *_ManagedObjectImplBase* erbt. Die Klasse dient der internen Repräsentierung von nomadischen Systemen und wird immer dann von einem EO instanziiert, wenn dieser von dem Anschluß eines neuen NoS innerhalb des Managementbereichs erfährt (siehe hierzu das Zustandsdiagramm eines EOs in Abbildung 4-29). Folgende Attribute und Methoden sind in der Klasse deklariert:

Attribute:

String ObjID: Dies ist der *compound name* des Objekts in <EXTENDED_XFN>-Syntax.

String Name: Dieses Attribut enthält einen bedeutungsvolleren Namen des Objekts, wie z.B. den *FQDN* des Endgeräts.

String IPAddr: Dies ist die IP-Adresse des NoS.

tState State: Der derzeitige Status des Objekts wird an dieser Stelle gespeichert. Die folgenden drei Zustände sind möglich: *unknown*, *identified*, *authentified*.

String Description: Dieses Attribut enthält eine Kurzbeschreibung des Objekts.

Die folgenden fünf Attribute werden für die Implementierung des *ObjectServerThread*-Interfaces benötigt und sind ebenfalls in den Klassen *StrategicPolicyImpl*, *GoalorientedPolicyImpl*, *OperationalPolicyImpl* und *EnforcementObjectImpl* deklariert. Sie werden allerdings nur nachfolgend erklärt. In den jeweiligen Beschreibungen der übrigen Klassen, wird ein Verweis auf diese Stelle erfolgen.

String ORB_ObjectReference: Die Objektreferenz des Objekts auf dem ORB wird an dieser Stelle als String gespeichert.

boolean up: Dieser boolesche Wert wird auf *true* gesetzt, wenn das Objekt sich auf dem ORB befindet.

boolean abort: Dieser Wert wird auf *true* gesetzt, falls während des Startvorgangs, das mit *startThread()* initiiert wird, ein Fehler auftritt und dieser deswegen abgebrochen werden mußte.

Thread ThreadReference: Dieses Attribut hält eine Referenz auf den Thread, der während des Startvorgangs kreiert wurde.

Methoden:

ManagedObjectImpl(String oID, String oName, tState st): Der Konstruktor belegt die Attribute *ObjID*, *Name* und *State* mit den jeweiligen Parameterwerten.

String getID(): Der Wert des Klassenattributs *ObjID* wird zurückgeliefert.

String getName(): Mit dieser Operation wird das Attribut *Name* abgefragt.

String getIPAddr(): Die IP-Adresse des Endgeräts ist das Ergebnis dieser Operation.

void setIPAddr(String newIP): Diese Operation ermöglicht das Setzen der IP-Adresse.

tState getState(): Der gegenwärtige Status des Objekts wird abgefragt.

tState setState(tState newState): Das Attribut *State* wird mit dem Wert *newState* belegt.

String getDescr(): Die Beschreibung des Objekts wird mit dieser Operation zurückgegeben.

void setDescr(String newDescr): Dem Objekt wird eine neue Beschreibung zugewiesen.

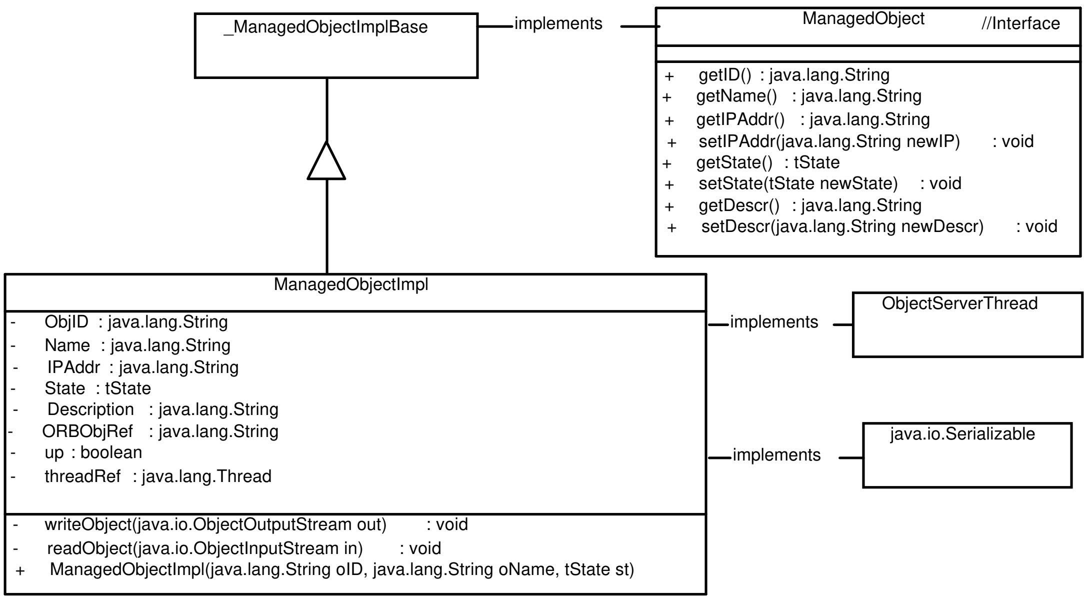

Abbildung 5-5: Das ManagedObjectImpl-Klassendiagramm

5.3.4 Die TargetListElement-Klasse

Die Klasse *TargetListElement* ist innerhalb der IDL-Interface-Deklaration der *Operational-Policy* als *struct* spezifiziert worden und wird deswegen automatisch durch den *idl2java*-Compiler generiert. Diese Klasse wird bei der Aktivierung einer operationalen Policy verwendet, um die kreierten *EnforcementObjects* zusammen mit den Namen der Managementagenten zu speichern, die diese beobachten. Die Attribute der Klasse haben die folgende Bedeutung:

String EnfObjID: Der *compound name* des EOs, der als String aus der Wortmenge des <EXTENDED_XFN> dargestellt wird.

String AgentID: Der *compound name* des zu beobachteten Managementagenten, der ebenfalls als String aus der Wortmenge des <EXTENDED_XFN> dargestellt wird.

Bei den Methoden der Klasse handelt es sich ausschließlich um die Konstruktoren der Klasse sowie einer Standardoperation zur Umwandlung des Klassenobjekts in einen String.

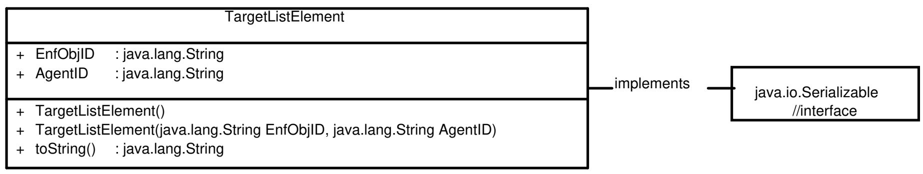

Abbildung 5-6: Das TargetListElement-Klassendiagramm

5.3.5 Die domainData-Klasse

Die Klasse *domainData* wird unter anderem als Rückgabewert des *Policy Interpreters* benützt. Es kann sowohl die Domänenbezeichnung für das Subjekt als auch für das Zielobjekt der operationalen Policy enthalten. Diese Klasse ist ebenfalls in der IDL-Interface-Deklaration der operationalen Policy als *struct* definiert worden, so daß diese automatisch von dem *idl2java*-Compiler während des Übersetzvorgangs generiert wird. Die Attribute der Klasse haben die folgende Bedeutung:

String domainExpr: Dieses Attribut enthält den Domänenausdruck, der in der Policy spezifiziert worden ist. Dies kann u.a. ein *TQL*-Ausdruck oder ein *compound name* des *Naming Service* sein.

domainType kind: Dieses Feld spezifiziert, welche Syntax zur Domänenangabe im vorhergehenden Feld verwendet wurde. Momentan sind die folgenden Kürzel erlaubt: TQL (für einen *TQL*-Ausdruck) und NS (für einen *compound name*).

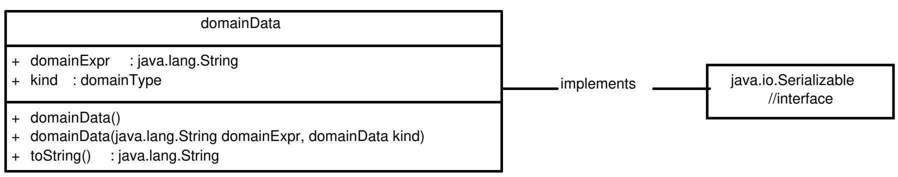

Abbildung 5-7: Das domainData-Klassendiagramm

5.4 Die Klassen des Persistence Service

5.4.1 Die PersistentObject-Klasse

Die Klasse *PersistentObject* ist in der IDL-Interface-Deklaration des *PersistenceService* als *struct* spezifiziert worden und wird während des Übersetzvorgangs des *idl2java*-Compilers automatisch generiert. Die Klasse wird als Datenstruktur zur Speicherung von *CORBA Objekten* mit Hilfe des *Persistence Service* verwendet. Die folgende zwei Attributfelder sind spezifiziert worden:

byte[] Obj: Dieses Klassenattribut enthält das entsprechende serialisierte *CORBA Objekt* in Form eines *Byte*-Arrays.

String ObjID: Dieses Feld enthält den eindeutigen Identifikator des *PersistentObjects*. Um eine leichtere Handhabung und intuitivere Suche nach persistenten Objekten zu ermöglichen, sollte dieses Feld den eindeutigen ID des *CORBA Objekts* enthalten. Innerhalb der hier implementierten Managementanwendung handelt es sich immer um den *compound name* des Objekts in <EXTENDED_XFN>-Syntax.

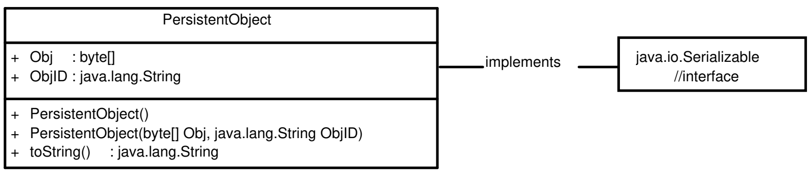

Abbildung 5-8: Das PersistentObject-Klassendiagramm

5.4.2 Die PersistenceServiceMobileAgent-Klasse

Die Klasse *PersistenceServiceMobileAgent* implementiert das in Abbildung 4-16 dargestellte IDL-Interface *PersistenceService*. Tatsächlich wird von der Klasse das *Java*-Interface *PersistenceServiceOperations* implementiert, das während der Übersetzung mit *idl2java* generiert wird. Wie erwartet erbt die Klasse von der *MobileAgent*-Klasse und erfüllt die in Abschnitt 5.3.1 aufgestellten Forderungen. Da die Operationen des *PersistenceService*-Interfaces bereits in Abschnitt 4.3.2 erklärt wurden, werden nachstehend nur die Attribute und Methoden betrachtet, die nicht diesem Interface angehören:

Attribute:

String ObjectID: Dies ist der eindeutige *compound name* des Agenten in der üblichen <EXTENDED_XFN>-Syntax.

Hashtable PersistentObjectDB: In dieser *java.util.Hashtable* werden die dem Agenten übergebenen *PersistentObjects* gespeichert. Wird die Operation *store()* des IDL-Interfaces aufgerufen, so wird die ganze Hashtable serialisiert und in ein File geschrieben. Es ist denkbar, diese *Hashtable* durch eine „echte“ Datenbank zu ersetzen und über die *JDBC*-Schnittstelle von *Java* anzusprechen. Davon wurde während der Implementierung des Agenten aufgrund einer fehlenden *JDBC*-kompatiblen Datenbank am Lehrstuhl jedoch abgesehen.

String databaseName: Dieses Feld enthält den Namen der Datenbank.

boolean database_allocated: Dieser Wert wird auf *true* gesetzt, falls eine Datenbank neu allokiert oder mit der Operation *restore()* wiederhergestellt wurde.

Vector usersList: Dieser *java.util.Vector* beinhaltet die momentanen User des *Persistence Service* und damit der allokierten Datenbank.

Methoden:

PersistenceServiceMobileAgent(String ObjectID): Dies ist der Konstruktor des Agenten, der lediglich den *compound name* als Parameter führt.

cleanUp(): Diese Operation wird während des Herunterfahrens des Agenten aufgerufen. Es wird der *compound name* des Agenten beim *Naming Service* gelöscht, so daß der Agent nicht mehr über diesen Namen von anderen CORBA Objekten erreichbar ist.

run(): Es wird der Agent mit dem, bei dessen Instanziierung durch den Konstruktoraufruf übergebenen *compound name* beim *Naming Service* eingetragen.

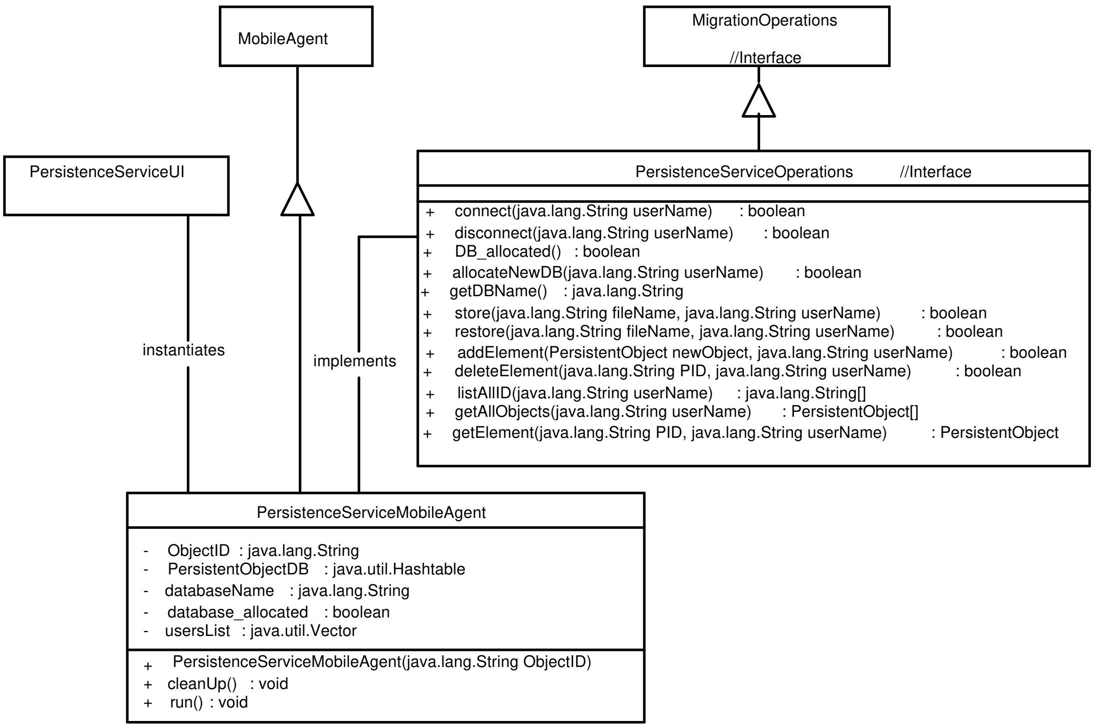

Abbildung 5-9: Das PersistenceServiceMobileAgent-Klassendiagramm

5.5 Die Klassen des Policy Service

5.5.1 Die PolicyObjectImpl-Klasse

Die Klasse *PolicyObjectImpl* ist die Oberklasse der *StrategicPolicyImpl-*, *GoalorientedPolicyImpl-* und *OperationalPolicyImpl-*Klasse. Sie implementiert das IDL Interface das in Abbildung 4-18 spezifiziert worden ist, indem es als Unterklasse der *_PolicyObjectImplBase*-Klasse deklariert wird, die während des Übersetzungsvorgangs der IDL-Datei vom *idl2java*-Compiler generiert wurde. Es können jedoch keine Instanzen der Klasse kreiert werden, da es sich um eine *abstrakte Klasse* handelt. Dies heißt, daß nicht alle Operationen tatsächlich implementiert werden, die als solche in der Klassensignatur angegeben werden. Im vorliegenden Fall sind die Operationen des *ObjectServerThread*-Interfaces nicht implementiert, so daß dies von den Unterklassen von *PolicyObjectImpl* übernommen werden muß. *PolicyObjectImpl* ist serialisierbar, um die Policies persistent anlegen zu können, indem der *Persistence Service* verwendet wird. Die Operationen des *PolicyObject*-Interfaces wurden bereits in Abschnitt 4.3.3 erläutert, so daß nur noch die in diesem Interface nicht enthaltenen Attribute und Methoden erklärt werden:

Attribute:

String ObjectID: Dies ist der *compound name* des Objekts in <EXTENDED_XFN>-Syntax.

String parentPolicyID: Dieses Attribut enthält den *compound name* der Eltern-Policy in <EXTENDED_XFN>-Syntax.

Vector childPolicyIDList: Der *java.util.Vector* enthält die *compound names* der Kinder-Policies.

Methoden:

PolicyObjectImpl(String ObjID): Dies ist der Konstruktor des Objekts, dem lediglich der Name der Policy als Parameter übergeben werden muß.

writeObject(ObjectOutputStream out): Diese Methode enthält die Standardoperation zur Serialisierung des Objekts und sollte von den Unterklassen überschrieben werden, falls besondere Schritte zur Serialisierung bestimmter Attribute durchgeführt werden müssen.

readObject(ObjectInputStream in): Diese Methode enthält ebenfalls nur den Aufruf der Standardoperation zur Deserialisierung des Objekts und sollte von einer Unterklasse überschrieben werden, falls bei bestimmten Attributen besondere Schritte durchgeführt werden müssen, um deren Wertebelegung zu erfahren.

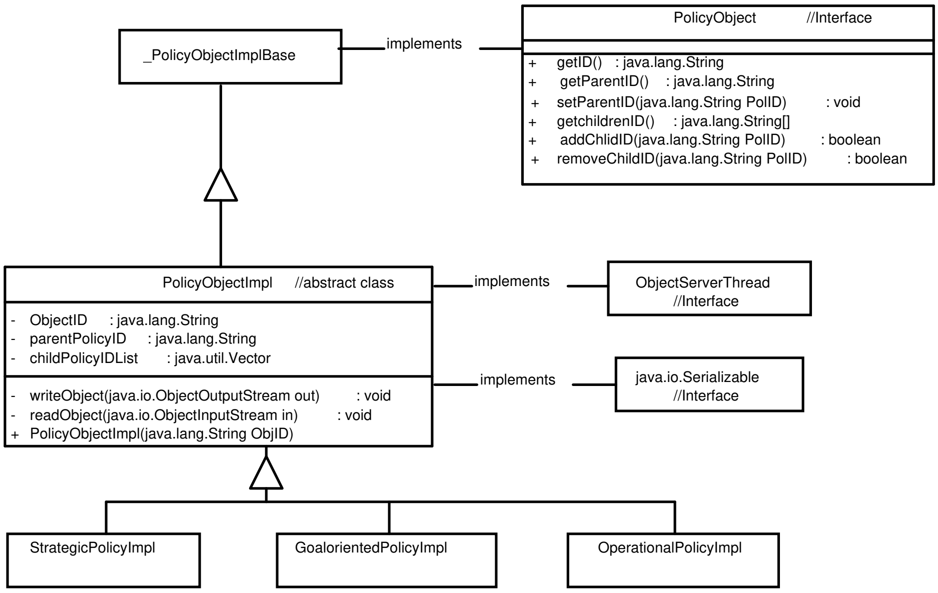

Abbildung 5-10: Das PolicyObjectImpl-Klassendiagramm

5.5.2 Die StrategicPolicyImpl-Klasse

Die Klasse *StrategicPolicyImpl* implementiert das *Java*-Interface *StrategicPolicyOperations*, das während der Übersetzung des IDL-Interfaces *StrategicPolicy*, deren Operationen in Abschnitt 4.3.3 beschrieben sind, von dem *idl2java*-Compiler generiert wird. Folgende Attribute und Methoden wurden in dieser Klasse zusätzlich deklariert:

Attribute:

String Description: Dieses Attribut enthält die in Prosa verfaßte Beschreibung der strategischen Policy.

Die übrigen Attribute werden für das *ObjectServerThread*-Interface benötigt und wurden bereits in Abschnitt 5.3.3 erklärt.

Methoden:

void writeObject(ObjectOutputStream out): In dieser Methode wird die Standard-Serialisierungsmethode aufgerufen.

void readObject(ObjectInputStream in): Bei der Deserialisierung wird vor der tatsächlichen Instanziierung überprüft, ob das Objekt bereits auf dem ORB ist. Dies ist z.B. immer dann der Fall, wenn zwei *PolicyFactories* nacheinander hochfahren und sich beide an die gleiche Datenbank binden, um die Policies wiederherzustellen. Ist dies der Fall, so wird der Deserialisierungsvorgang abgebrochen.

StrategicPolicyImpl(String ObjectID, String PolDescr): Dem Konstruktor der Klasse muß der *compound name* in der üblichen <EXTENDED_XFN>-Syntax und die Beschreibung der strategischen Policy übergeben werden.

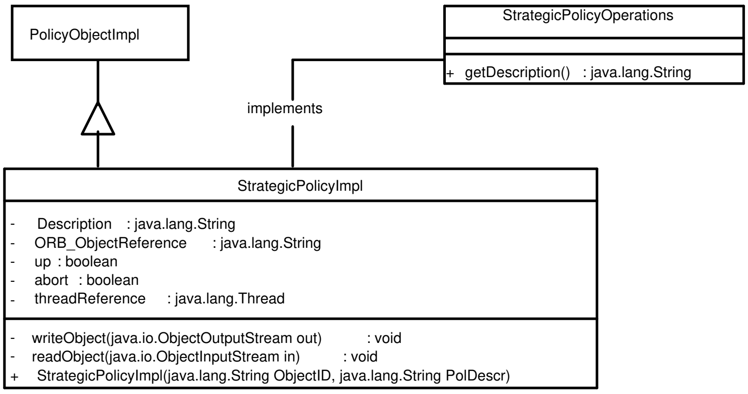

Abbildung 5-11: Das StrategicPolicyImpl-Klassendiagramm

5.5.3 Die GoalorientedPolicyImpl-Klasse

Die Klasse *GoalorientedPolicyImpl* implementiert das *Java*-Interface *GoalorientedPolicyOperations*, das u.a. bei der Übersetzung des IDL-Interfaces *GoalorientedPolicy* durch den Compiler generiert wird. Die Operationen des IDL-Interfaces wurden bereits in Abschnitt 4.3.3 erläutert, so daß nachstehend nur die zusätzlichen Attribute und Methoden der Klasse betrachtet werden:

Attribute:

Die ersten acht Klassenattribute entsprechen den Template-Feldern einer zielorientierten Policy, die in Abschnitt 3.4 dieser Ausarbeitung definiert wurden.

String policyName: Dies ist der Name der zielorientierten Policy, der nicht mit dem Attribut *ObjectID* der Oberklasse verwechselt werden darf. *ObjectID* ist im Gegensatz zu *policyName* eindeutig innerhalb der CORBA-Umgebung und entspricht der <EXTENDED_XFN>-Syntax, so daß ein beliebiger CORBA Client eine Referenz auf die Policy über den *Naming Service* erhalten kann.

PolicyModality modality: Dies entspricht der Modalität der zielorientierten Policy und kann mit den Werten *Permission, Prohibition* und *Obligation* belegt werden.

String subject: Der Wert dieses Attributs entspricht dem Subjekt der Policy.

String subjectDomain: An dieser Stelle wird die Domäne des Subjekts gehalten.

String target: Das Zielobjekt der Policy wird an dieser Stelle gespeichert.

String targetDomain: Dieses Attribut wird mit der Domäne des Zielobjekts belegt.

String actions: Die spezifizierten Managementaktionen der zielorientierten Policy werden durch dieses Attribut repräsentiert.

String constraint: Dieses Attribut entspricht dem Constraint der zielorientierten Policy.

Die übrigen Attribute werden für das *ObjectServerThread*-Interface benötigt und wurden bereits in Abschnitt 5.3.3 erklärt.

Methoden:

void writeObject(ObjectOutputStream out): Diese Methode enthält einen Aufruf der Standard-Serialisierungsmethode.

void readObject(ObjectInputStream in): Wie bei der *StrategicPolicyImpl*-Klasse wird bei der Deserialisierung vor der tatsächlichen Instanziierung des Objekts überprüft, ob es bereits auf dem ORB ist.

GoalorientedPolicyImpl(String ObjectID, String pName, PolicyModality pModality, String pSubject, String pSDomain, String pTarget, String pTDomain, String pAction, String pConstraint): Dem Konstruktor der Klasse muß neben dem üblichen *compound name* des Objekts, die Belegung der Template-Felder als Parameter übergeben werden. Diese werden dann den ersten acht Klassenattributen zugewiesen.

In Abbildung 5-12 ist das dazugehörende Klassendiagramm dargestellt.

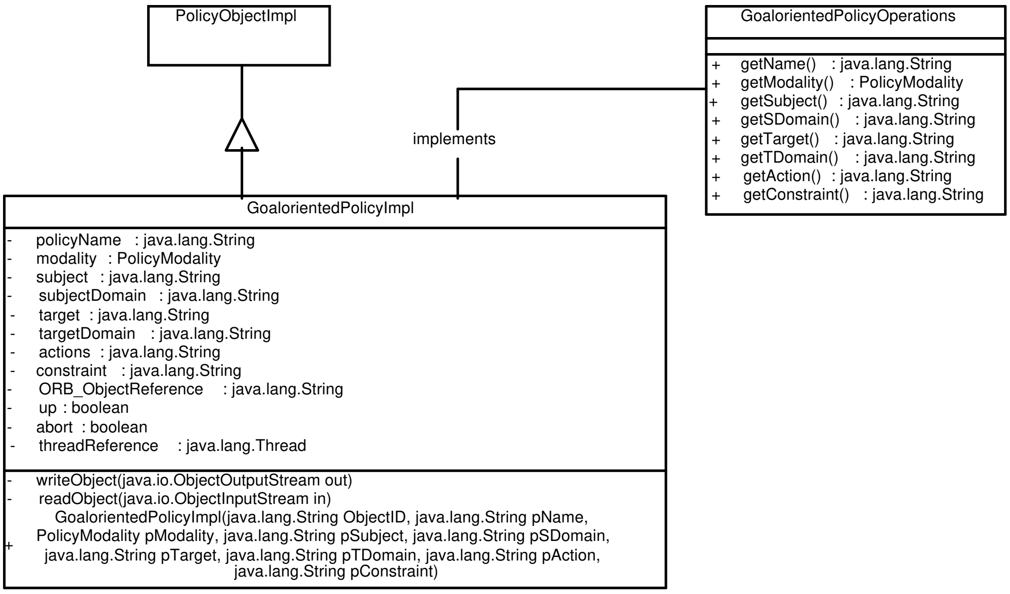

Abbildung 5-12: Das GoalorientedPolicyImpl-Klassendiagramm

5.3.4 Die OperationalPolicyImpl-Klasse

Die Klasse *OperationalPolicyImpl* implementiert das *Java*-Interface *OperationalPolicyOperations*, das während der Übersetzung des IDL-Interfaces *OperationalPolicy*, das in Abschnitt 4.3 erklärt wurde, durch den *idl2java*-Compiler generiert wird. Ebenfalls wurde in Abschnitt 4.3 erläutert, daß sich dieses Objekt an den *AgentInitChannel* bindet, um das Herauf- und Herunterfahren von Managementagenten zu bemerken. Aus diesem Grund muß die Klasse zusätzlich die Operationen des *PushConsumerOperations*-Interfaces implementieren, das aus den beiden Methoden *push()* und *disconnect_push_consumer()* besteht. Die Klasse enthält die folgenden Attribute und Methoden, die ebenfalls in Abbildung 5-12 veranschaulicht sind:

Attribute:

String PDL_description: Dieser String enthält die Beschreibung der operationalen Policy in PDL.

String policyName: Der Name der operationalen Policy, der in der PDL-Beschreibung kodiert ist. Dieses Attribut wird während der ersten Ausführung der Operation *checkPDL()* oder *activate()* belegt.

Vector targetList: Dieser *Vector* enthält die IDs der während der Ausführung von *activate()* kreierten EOs zusammen mit den IDs der von diesen EOs beobachteten Managementagenten. Für diese Liste wird die Datenstruktur *TargetListElement* verwendet, die in Abschnitt 5.3.4 erklärt wurde.

String enforcementObjectFactoryName: Der *compound name* der EOF wird an dieser Stelle gespeichert. Erst bei der Aktivierung der Policy mit *activate()* wird dieses Attribut mit einem Wert belegt.

String agentInitChannelName: Dieser String beinhaltet den *compound name* des *AgentInitChannels* in <EXTENDED_XFN>-Syntax. Dieses Attribut wird erst beim Aufruf der Operation *activate()* gesetzt.

domainData targetDomain: Der Domänenausdruck des Zielobjektbereichs wird in diesem Attribut gespeichert. Erst nach der Aktivierung der Policy wird *targetDomain* tatsächlich ein Wert zugewiesen.

domainData subjectDomain: Der in der operationalen Policy spezifizierte Domänenausdruck für den Subjektbereich wird in diesem Attribut abgelegt. Auch *subjectDomain* wird erst bei der Aktivierung ein Wert zugewiesen.

String constraint: Nach dem Aufruf der Operation *activate()* wird ebenfalls der Constraint der operationalen Policy als String in PDL-Syntax abgespeichert.

String actions: Der in PDL spezifizierte Aktionsbereich wird an dieser Stelle als String gespeichert.

boolean active: Dieses Attribut wird nach der Aktivierung der Policy auf *true* gesetzt.

boolean was_active: Dieses Attribut wird nur von einem *EOFMA* während des Hochfahrens abgefragt, um festzustellen ob für diese operationale Policy EOs kreiert werden sollen.

ProxyPushSupplier _ProxyPushSupplier: Eine Referenz auf den, während der Aktivierung der Policy kreierten *ProxyPushSupplier*, wird an dieser Stelle gehalten. Diese Referenz wird u.a. zum Trennen der Verbindung zwischen dem *OperationalPolicyImpl*-Objekt und dem Event Channel benötigt.

PushConsumer _PushConsumer: An dieser Stelle wird eine Referenz auf den kreierten *PushConsumer* gehalten.

Die übrigen Attribute werden für das *ObjectServerThread*-Interface benötigt und wurden bereits in Abschnitt 5.3.3 erklärt.

Methoden:

void writeObject(ObjectOutputStream out): In dieser Methode wird die Standard-Serialisierungsmethode aufgerufen.

void readObject(ObjectInputStream in): Bei der Deserialisierung wird vor der tatsächlichen Instanziierung überprüft, ob das Objekt bereits auf dem ORB ist.

OperationalPolicyImpl(String ObjectID, String PolDescr): Der Konstruktor der Klasse weist neben dem *compound name* des Objekts ebenfalls die Beschreibung der Policy in PDL auf.

boolean was_active(): Diese Methode fragt das Attribut *was_active* ab.

void push(Any any): Die *push()*-Methode wird ausschließlich von dem *ProxyPushSupplier* des *AgentInitChannels* aufgerufen, wenn ein Managementagent hoch- bzw. herunterfährt. Bei dem übergebenen *Any* handelt es sich immer um ein

StructuredEvent, das u.a. den Namen und den Typen des Agenten enthält. Ist der Agent von der operationalen Policy betroffen, so werden die nötigen Schritte unternommen.

void disconnect_push_consumer(): Diese Methode wird entweder aufgerufen, wenn der *AgentInitChannel* zerstört oder die operationale Policy deaktiviert wird.

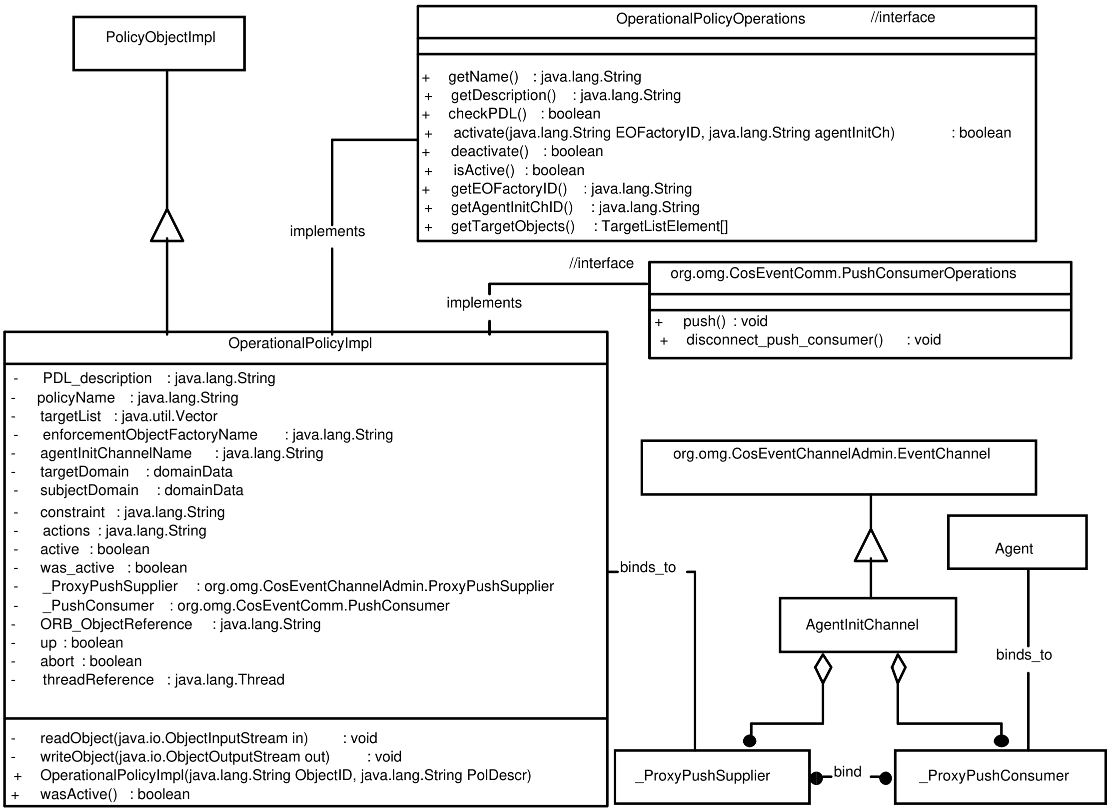

Abbildung 5-13: Das OperationalPolicyImpl-Klassendiagramm

5.5.5 Die PolicyFactoryMobileAgent-Klasse

Die Klasse *PolicyFactoryMobileAgent* implementiert das IDL-Interface *PolicyFactory*, das in Abschnitt 4.3.3 erläutert wurde, indem es tatsächlich das *Java*-Interface *PolicyFactoryOperations* implementiert, das während des Übersetzvorgangs des entsprechenden IDL-Files generiert wurde. Die folgenden zusätzlichen Attribute und Methoden wurden in der Klasse definiert und sind in Abbildung 5-14 dargestellt:

Attribute:

String ObjectID: Der eindeutige *compound name* der Policy Factory in <EXTENDED_XFN>-Syntax.

PersistenceServiceOperations _PersistenceService: Dies ist eine Referenz auf den *Persistence Service*, der zum Speichern und Wiederherstellen der Policies benötigt wird.

String persistenceServiceName: Der *compound name* des Persistence Service in <EXTENDED_XFN>-Syntax.

String peristentDatabaseName: Der (File-)Name der Datenbank, in die alle neuen Policies eingefügt werden.

String policyDomainPrefix: Dies ist der Ast (in <EXTENDED_XFN>-Syntax) des *naming graphs*, ab dem alle neu kreierten Policies eingefügt werden. D.h., daß allen neu generierten Policies dieser String als Prefix ihres Namens vorangestellt wird.

Hashtable local_policyObjectDatabase: Diese *Hashtable* enthält alle Policies, die gegenwärtig von der *Policy Factory* verwaltet werden.

boolean prev_started: Dieses Attribut wird nach dem erfolgreichen Hochfahren des Agenten auf *true* gesetzt. Es wird während der Ausführung der *run()*-Methode abgefragt, um festzustellen, ob der Agent migriert wurde.

Methoden:

PolicyFactoryMobileAgent(String ObjectID, String PSAgentName, String PersistentDBName, String domainBranch): Dem Konstruktor des Agenten werden die entsprechenden Wertebelegungen für die oben beschriebenen Klassenattribute als Parameter übergeben.

boolean connect_to_PSMA(): Diese Methode versucht über den *Naming Service* mit dem *persistenceServiceName* eine Verbindung zu einem *PSMA* zu etablieren. Im nächsten Schritt werden alle Policies die in der Datenbank *persistentDatabaseName* enthalten sind, wiederhergestellt.

boolean reconnect_to_PSMA(): Durch Aufruf dieser internen Methode, wird versucht, die Verbindung zum *PSMA* wiederherzustellen. Sie wird immer dann aufgerufen, wenn die Referenz *_PersistenceService* nicht mehr gültig ist und deswegen angenommen werden muß, daß der *PSMA* in der Zwischenzeit migriert ist.

void cleanUp(): Diese Methode wird vom Agentensystem immer dann aufgerufen, wenn der Agent entweder heruntergefahren wird oder migriert werden soll. In diesem Fall enthält die Operation die Anweisungen, den *compound name* des Agenten beim *Naming Service* zu löschen, sowie alle Policies zusätzlich in der Datenbank des *PSMA* zu aktualisieren und zu speichern.

void run(): Die *run()*-Methode fügt den *compound name* des Agenten beim *Naming Service* ein und ruft die Operation *connect_to_PSMA()* auf.

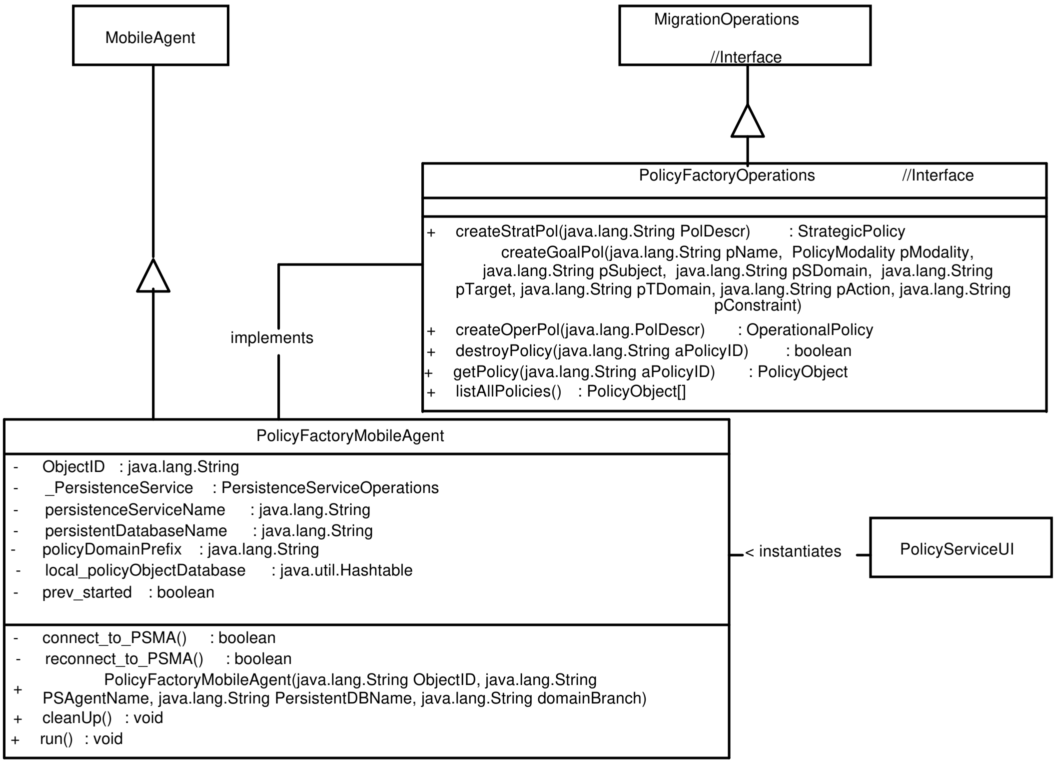

Abbildung 5-14: Das PolicyFactoryMobileAgent-Klassendiagramm

5.6 Die Klassen des Policy Enforcement Service

5.6.1 Die EnforcementObjectImpl-Klasse

Die Klasse *EnforcementObjectImpl* implementiert das IDL-Interface *EnforcementObject*, das u.a. in Abschnitt 4.3.4 ausführlich erläutert wurde, indem es von der Klasse *_EnforcementObjectImplBase* erbt, die vom *idl2java*-Compiler während des Übersetzungsvorgangs des IDL-Files automatisch generiert wurde. Aus der Beschreibung des *Policy Enforcement Services* aus Abschnitt 4.3.4 ist bekannt, daß sich ein *Enforcement Object* an den Event Channel eines Managementagenten bindet, um diesen zu beobachten. Dies erfordert, daß die Objektklasse die Operationen *push()* und *disconnect_push_consumer()* des *PushConsumer*-Interfaces des *Event Services* implementiert. In Abbildung 5-15 ist das dazugehörige Klassendiagramm dargestellt, das die folgenden zusätzlichen Attribute und Methoden der Klasse aufweist:

Attribute:

String ObjectID: Dies ist der *compound name* des EOs in <EXTENDED_XFN>-Syntax.

String targetAgentID: Dies ist der *compound name* des Managementagenten, der von dem EO beobachtet wird.

String eventChannelName: Der *compound name* des Event Channels, an den sich das EO gebunden hat, um den spezifizierten Managementagenten zu beobachten.

String constraint: Der Constraint der ursprünglichen operationalen Policy in PDL-Syntax, der, bei Auftreten eines Events, dem *Constraint Interpreter* zur Evaluierung übergeben wird.

domainData subjectDomain: Dies ist der in der operationalen Policy spezifizierte Domänenbereich des Subjekts.

String sourcePolicyID: Der *compound name* der Ursprungspolicy wird an dieser Stelle gespeichert.

String managementActions: Der spezifizierte *Action*-Ausdruck in PDL-Syntax wird bei Auftreten eines Events dem *Action Interpreter* übergeben.

String nomadicSystemDomain: Dieser String ist der Prefix des *compound names* desjenigen CORBA Objekts, das beim erstmaligen Auftreten eines NoS kreiert wird und dessen Zustand widerspiegelt.

ProxyPushSupplier _ProxyPushSupplier: Eine Referenz auf den, während der Aktivierung des EOs kreierten *ProxyPushSupplier*, wird an dieser Stelle gehalten. Diese Referenz wird u.a. zum Trennen der Verbindung zwischen dem *EnforcementObjectImpl*-Objekt und dem Event Channel benötigt.

PushConsumer _PushConsumer: An dieser Stelle wird eine Referenz auf den kreierten *PushConsumer* gehalten.

boolean monitoring_started: Diese boolesche Variable wird während des Aufrufs der Methode *startMonitoring()* auf den Wert *true* gesetzt.

Die übrigen Attribute werden für das *ObjectServerThread*-Interface benötigt und wurden bereits in Abschnitt 5.3.3 erklärt.

Methoden:

EnforcementObjectImpl (String ObjID, String srcPolID, String AgentID, domainData subjectDomain, String constraint, String actions, String ncsDomain): Dem Konstruktor der Klasse wird neben dem *compound name* des Objekts ebenfalls die Belegungen für die weiteren sieben Klassenattribute übergeben, die oben beschrieben worden sind.

boolean checkOriginDomain (String ncs_id, String subjectDomain): Diese interne Methode prüft, ob ein NoS mit dem ID *ncs_id*, das momentan der MAC-Adresse des NoS entspricht, aus dem spezifizierten Subjektbereich kommt. Für diese Zwecke wird bei einer *JMAPI*-Datenbank nachgefragt.

Die durchgeführten Aktionen innerhalb *push()*-Methode wurden bereits im Zustandsdiagramm in Abbildung 4-29 veranschaulicht.

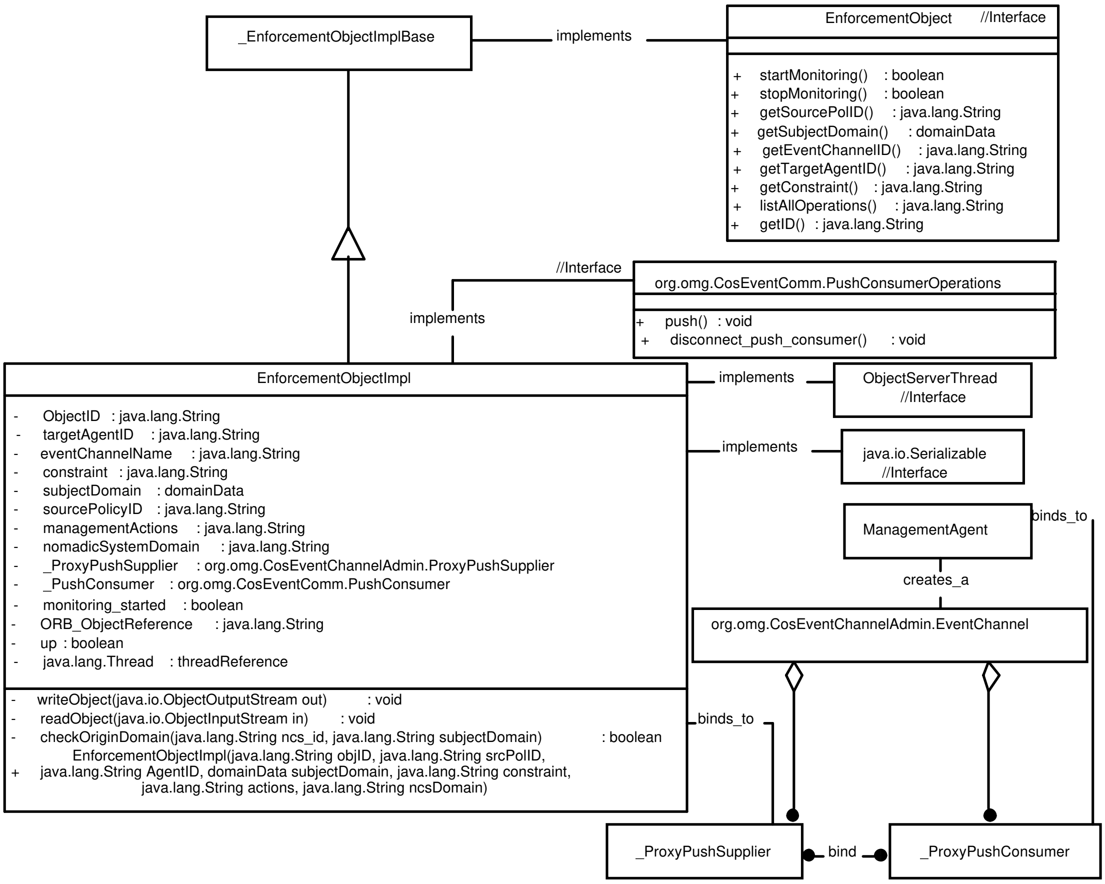

Abbildung 5-15: Das EnforcementObjectImpl-Klassendiagramm

5.6.2 Die EnforcementObjectFactoryMobileAgent-Klasse

Die Klasse *EnforcementObjectFactoryMobileAgent* implementiert das IDL-Interface *EnforcementObjectFactory*, das in Abschnitt 4.3.4 erläutert wurde, indem es tatsächlich das *Java*-Interface *EnforcementObjectFactoryOperations* implementiert, das während des Übersetzungsvorgangs des entsprechenden IDL-Files generiert wurde. Die folgenden zusätzlichen Attribute und Methoden wurden in der Klasse definiert und sind in Abbildung 5-16 dargestellt:

Attribute:

String ObjectID: Der eindeutige *compound name* der Enforcement Object Factory in <EXTENDED_XFN>-Syntax.

PersistenceServiceOperations _PersistenceService: Dies ist eine Referenz auf den *Persistence Service*, der zum Wiederherstellen der EOs während des Hochfahrens des Agenten benötigt wird.

String persistenceServiceName: Der *compound name* des Persistence Service in <EXTENDED_XFN>-Syntax.

String peristentDatabaseName: Der (File-)Name der Datenbank, aus der die EOs wiederhergestellt werden.

String managementAgentType: Der Typ des Managementagenten für den ausschließlich EOs produziert werden.

String enforcementObjectDomainPrefix: Derjenige Ast des *naming graphs*, unter den alle EOs eingefügt werden. Dies heißt, daß alle EOs diesen String als Prefix in ihrem *compound name* tragen werden.

String nomadicSystemDomain: Derjenige Ast des *naming graphs*, unter den alle CORBA Objekte eingefügt werden, die den Zustand eines im Managementbereich angeschlossenen NoS repräsentieren.

Hashtable local_enforcementObjectDatabase: Diese *Hashtable* enthält alle von der EOF verwalteten EOs.

boolean prev_started: Dieses Attribut wird nach dem erfolgreichen Hochfahren des Agenten auf *true* gesetzt.

Methoden:

EnforcementObjectFactoryMobileAgent(String ObjectID, String PSAgentName, String ManagementAgentType, String PersistentDBName, String domainBranch, String ncsDomain): Der Konstruktor der *EOFMA* enthält als Parameter die Wertebelegungen der oben beschriebenen Klassenattribute.

void cleanUp(): Die Operation löscht den *compound name* des Agenten beim *Naming Service*.

void run(): Die *run()*-Methode fügt den *compound name* des Agenten beim *Naming Service* ein und versucht eine Verbindung zum Persistence Service zu etablieren. Gelingt dies, so werden aus der gegebenen Datenbank des PS alle operationalen Policies dahingehend untersucht, ob diese von dem *EOFMA* betroffen sind und die benötigten EOs kreiert werden sollen.

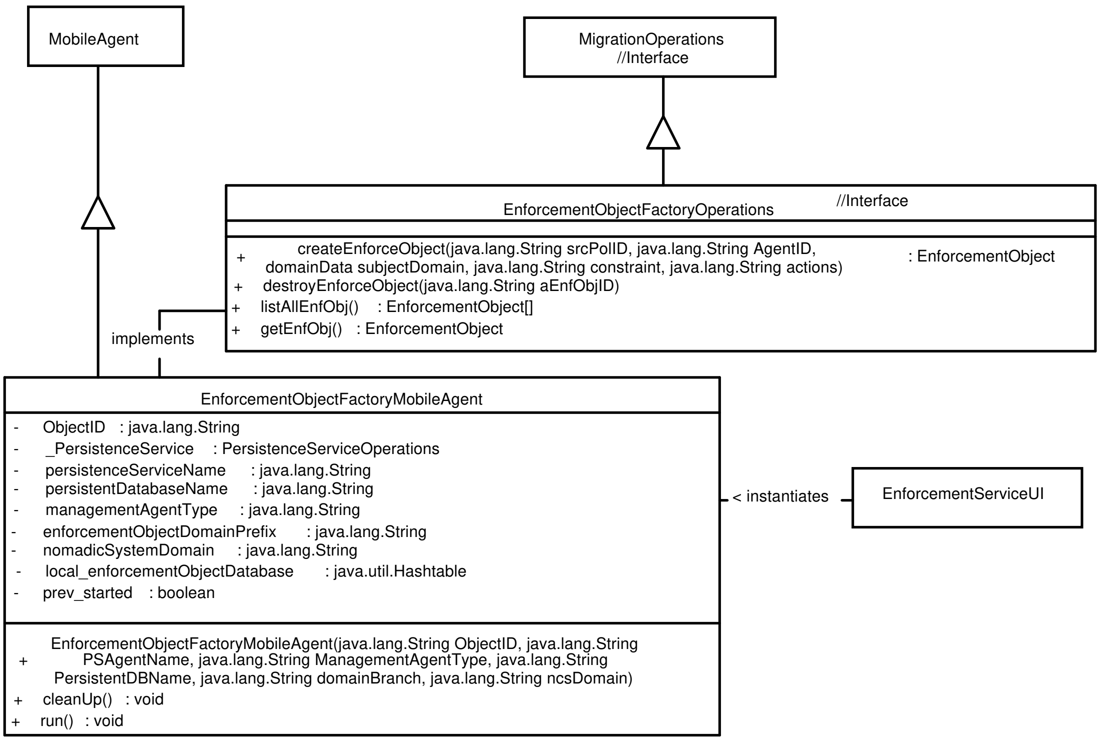

Abbildung 5-16: Das EnforcementObjectFactoryMobileAgent-Klassendiagramm

5.6 Zusammenfassung

In diesem Kapitel wurde das Objektmodell der implementierten Managementanwendung vorgestellt. Für die graphische Darstellung wurde die Notation der *Object Modeling Technique (OMT)* verwendet. Da dieses Kapitel v.a. der Dokumentation der erstellten Software dienen soll, wurde ein niedrigerer Abstraktionsgrad gewählt, so daß bereits die Einflüsse sowohl der eingesetzten Implementierungssprache *Java* als auch der CORBA-Implementierung *Visibroker 3.0* von *Visigenic* in den jeweiligen Diagrammen zu sehen sind. In dem nun folgenden Kapitel 6 werden die eingesetzten Programmier- und Modellierungswerkzeuge detaillierter betrachtet.

Kapitel 6

Implementierung

Dieses Kapitel beschäftigt sich mit der konkreten Umsetzung der in den vorhergehenden Kapiteln erarbeiteten Managementanwendung in eine prototypische Implementierung. Neben einer Darstellung der besonderen Merkmale der eingesetzten, objektorientierten Programmiersprache *Java*, erfolgt ebenfalls eine Beschreibung der verwendeten Programmier- und Modellierungswerkzeuge. Das Kapitel wird zum Schluß mit einem Erfahrungsbericht abgerundet.

6.1 Java

6.1.1 Einführung

Die Programmiersprache *Java* ist derzeit im Rahmen des *Java Development Kit (JDK) Version 1.1.6* von Sun Microsystems über deren Homepage frei beziehbar. Da Java bis vor kurzem noch v.a. im Bereich des Web-Site-Designs eingesetzt wurde, erfuhr sie seit ihrer ersten Veröffentlichung im Jahre 1995 eine hohe Verbreitung in der Internet-Gemeinde. Wegen ihrer besonderen Merkmale jedoch, wird sie immer häufiger auch als Implementierungssprache für Stand-Alone-Anwendungen eingesetzt. Folgende Merkmale (siehe hierzu auch [Flan97]) zeichnen Java aus:

- Java ist eine *objektorientierte* Programmiersprache, d.h. alle Elemente innerhalb von Java, außer den fundamentalen Sprachbausteinen (wie z.B. *int*, *boolean, etc.*), sind in Klassen gekapselt, die nur über die öffentlichen Methoden und Attribute manipuliert werden können.
- Der in Java geschriebene Quellcode wird *nicht* wie sonst üblich in Maschinencode übersetzt, sondern in einen sog. *Bytecode*. Dieser Bytecode wird von einer *Java Virtual Machine (VM) interpretiert.*
- Java ist *Hardware-unabhängig*, da der Java-Bytecode auf jedem System interpretiert werden kann für den eine VM existiert, *ohne* daß der Quellcode neu übersetzt werden muß. Dies entspricht auch dem nicht nur mit Java verfolgtem Ziel *„Write once, run anywhere.“*.

- Java ist eine *robuste* und *sichere* Sprache, da keine Adressen- und Pointermanipulationen, wie z.B. in *C++*, möglich sind. Dadurch ist eine der am häufigsten auftretenden Fehlerquellen ausgeschlossen.
- Java bietet mit den Klassen seiner Standard-Packages eine umfassende API für Programmierer (siehe auch [GoYe96]) an. Einige der interessantesten Klassen sind die folgenden:
  *java.net.**: Klassen, die sich um die Kommunikation in einem Netzwerk kümmern
  *java.io.**: Klassen, die umfangreiche Methoden für die Ein- und Ausgabe anbieten; besonders interessant ist das *java.io.Serializable*-Interface, das die Serialisierung von Objekten übernimmt
  java.lang.Thread: Mit dieser Klasse ist es möglich, eigene Programme als *Threads* zu realisieren; innerhalb einer Hauptroutine können auf dieser Weiser mehrere Programmteile nebenläufig gestartet und gesteuert werden
  *java.awt.**: Dies ist eine umfangreiche API zur Erstellung von GUIs
- Java ist *dynamisch*, da die Möglichkeit besteht, während der Laufzeit neue Klassen zu laden und zu instanziieren.

Diese Vorteile von Java waren ausschlaggebend für deren Wahl als Implementierungssprache.

6.1.2 Die Package-Hierarchie

Bereits im vorhergehenden Abschnitt wurde der Begriff des *Packages* eingeführt. Ein *Package* dient zur Gruppierung mehrerer Klassen. Die Klassen innerhalb eines Packages haben besondere Zugriffsrechte auf die Attribute und Methoden der sich ebenfalls in diesem Package befindenden Klassen. Eine Klasse wird eindeutig durch den Package- und den Klassen-Namen identifiziert. Um die Möglichkeit zu haben, Klassen und Packages *global* eindeutig identifizieren zu können, wird der Package-Name nach dem Schema der *Fully Qualified Domain Names (FQDN)* gebildet (siehe hierzu auch [Flan97]). Für die im Rahmen dieser Arbeit entwickelten Klassen wurde der folgende Prefix für die Package-Namen gewählt:

`de.unimuenchen.informatik.mnm.ncscontrol.`

Ausgehend von diesem Prefix wurden die folgenden Packages definiert:

- `persistence`: In diesem Package befinden sich alle Klassen, die im Zusammenhang mit dem *Persistence Service* stehen.
- `policy`: Dieses Package enthält alle diejenigen Klassen, die vom *Policy Service* benötigt werden.
- `enforcement`: Alle Klassen, die den *Policy Enforcement Service* umsetzen, befinden sich in diesem Package.
- `parser`: Dieses Package enthält die Klassen für den *Policy Interpreter*, *Constraint Interpreter* und den *Action Interpreter*.
- `NcSControlCommonData`: Diejenigen Klassen, die von allen obigen Packages benötigt werden, sind in diesem Package zusammengefaßt.

- `domain`: Dieses Package enthält nur die Klasse *DomainBuilder*, die eine Domänenstruktur aufbaut, indem ein *naming graph* mit Hilfe des *Naming Service* konstruiert wird.
- `mo`: In diesem Package ist u.a. die Klasse *ManagedObjectImpl* zu finden, deren Instanz jeweils ein neu hinzugekommenes nomadisches System innerhalb des Managementbereichs repräsentiert. Zudem ist in diesem Package bereits das Java-Interface *NoCSMgmtAgent* enthalten, das dem spezifiziertem IDL-Interface in Abbildung 4-25 entspricht.

6.1.3 Das Interface-Konzept

In Java wird im Gegensatz zu anderen OO Programmiersprachen wie *C++ keine* Mehrfachvererbung unterstützt. Dies heißt, daß eine Klasse immer nur höchstens eine Oberklasse haben darf. Um trotzdem die Möglichkeit zu haben einer Klasse bestimmte, festgelegte Eigenschaften und ein vorher definiertes Verhalten zuzuweisen, werden *Interfaces* definiert, die Klassen *implementieren* können. Interfaces enthalten Konstanten und die *Signaturen* von Methoden, d.h. es werden jeweils nur der Methodenname, die Parameter und der Rückgabetyp festgelegt. In den Klassen, die anzeigen, daß diese ein Interface implementieren, muß der entsprechende Code hinzugefügt werden. Die Anzahl der Interfaces, die eine Klasse implementiert, ist nicht begrenzt.

Obwohl dies zunächst wie eine unnötige Beschränkung auf die OO wirkt, ergeben sich spätestens durch die eindeutigen Vererbungshierarchien die Vorteile dieses Konzepts, da u.a. der Quellcode für Außenstehende besser lesbarer wird. Eine langwierige Suche nach der tatsächlichen Implementierung einer Methode in einer der Oberklassen entfällt meistens. In [CoMa97] wird u.a. eine Vorgehensweise beschrieben, wie festgestellt werden kann, bei welchen Gegebenheiten die Spezifizierung eines Interfaces einer Vererbung vorzuziehen ist, so daß der interessierte Leser darauf verwiesen wird.

Innerhalb des implementierten Prototypen wurde dieses Konzept z.B. für den einheitlichen Zugriff auf CORBA-relevante Daten angewendet, indem das Interface *ObjectServerThread* definiert wurde (siehe hierzu auch das Klassendiagramm in Abbildung 5-4). Zusätzlich werden alle IDL-Interfaces auf Java-Interfaces abgebildet.

6.1.4 Die Serialisierung von Objekten

Die Serialisierung eines Objekts, also die geeignete Umwandlung dessen Zustands in einen Bytestrom, ist in Java durch die API des *java.io*-Packages sehr einfach realisierbar. Grundsätzlich ist jedes Objekt serialisierbar, das das Interface *java.io.Serializable* implementiert. Dieses Interface weist keine Signaturen, auf sondern dient lediglich als Auszeichnung, daß dieses Objekt serialisiert werden kann (und darf). Wird die Serialisierung eines Objekts tatsächlich angestoßen, so wird rekursiv die Serialisierung der Elemente des Objekts eingeleitet. Auf diese Weise wird ein *Serialisierungsbaum* aufgebaut und in einen Bytestrom geschrieben, der bei der Deserialisierung wieder abgearbeitet wird. I.d.R. reicht demzufolge die bloße Angabe des *java.io.Serializable*-Interfaces in der Klassensignatur eines Objekts aus, um dessen Serialisierung zu ermöglichen. Bei komplexeren Objekten kommt es jedoch nicht selten vor, daß diese nicht serialisierbare Attribute aufweisen. In diesem Fall müssen die Serialisierungsund Deserialisierungsmethoden *writeObject()* und *readObject()* mit einer eigenen Implementierung überschrieben werden.

Da die *Services* der entwickelten, verteilten Anwendung als *MobileAgents* realisiert sind, müssen alle Objekte innerhalb des Prototypen serialisierbar sein, um (zumindest prinzipiell) eine Migration des Agenten zu ermöglichen. Desweiteren kommt die Serialisierung von Objekten ebenfalls bei der Nutzung des *Persistence Service* zum Einsatz.

6.2 Visibroker 3.0 for Java

Der *Visibroker 3.0 for Java* [Visi97] der Firma *Visigenic* ist in seinem Hauptbestandteil die Implementierung eines CORBA 2.0 konformen ORBs. Neben dem ORB werden sowohl eine komplette Entwicklungsumgebung für die Generierung von CORBA Objekten als auch Administrationstools zur Überwachung der Kommunikation mitgeliefert. Die folgenden Komponenten des Visibrokers kamen für die Implementierung des Prototypen zum Einsatz:

- *idl2java*: Dies ist ein Compiler, der aus gegebenen IDL-Schnittstellen die für die Objektimplementierung benötigten Client-Stubs, Server-Skeletons und Interfaces in der Programmiersprache Java generiert.
- *idl2ir*: Dieses Tool fügt neue IDL Interfaces in die *Interface Repository* ein.
- *osagent*: Der *osagent* ist ein sog. *Smart Agent*, der, nach dem Hochfahren auf einem Host im Netz, die Kommunikationsgrundlage für CORBA Objekte etabliert. Die Nutznießer des ORBs, also alle Client und Server Objekte, versuchen bei der ersten Kontaktaufnahme den *Smart Agent* durch einen UDP-Broadcast ausfindig zu machen. Die dafür nötige Port-Adresse kann u.a. durch Umgebungsvariablen gesetzt werden.
- *irep*: Hiermit wird eine *Interface Repository* hochgefahren, die u.a. auch eine GUI zur Darstellung aller enthaltenen Interfaces bietet.

Von Visigenic wurde ebenfalls eine gemeinsame Implementierung des *Naming* und *Event Service* [Visi97b] verwendet. Im Rahmen des Prototypen werden vom *Event Service* nur die APIs verwendet, während für die Nutzung des *Naming Service* zusätzlich ein Agent gestartet werden muß, der den Verzeichnisdienst für Clients zur Verfügung stellt.

6.2.1 Der Entwicklungsprozeß

In Abbildung 6-1 ist der Vorgang der Entwicklung von CORBA Objekten veranschaulicht und lautet wie folgt:

(1) Der Entwickler legt die Eigenschaften und das Verhalten eines CORBA Server Objekts fest, indem die nach außen hin sichtbare Schnittstelle in IDL beschrieben wird.

(2) Mit der Verwendung des *idl2java*-Compilers werden die für die Implementierung notwendigen Client-Stubs, Server-Skeletons und Interfaces in Java generiert. Zusätzlich kann der *Visigenic*-Compiler bereits einen Programmrahmen für die Ser-

ver-Implementierung anlegen, der leere Methodenrümpfe enthält. Der Entwickler muß daraufhin nur noch die entsprechenden Methodenrümpfe mit Code füllen.

(3) Ein Client-Programm hingegen enthält, nachdem der Kontakt zum entsprechenden Server Objekt zustande gekommen ist, lediglich den Aufruf der Methoden. Über die Client-Stubs wird der Aufruf zum Server-Objekt weitergereicht, so daß sich der Entwickler mit der Kommunikation zwischen Client und Server nicht weiter beschäftigen muß.

(4) Der Quellcode des Servers und des Clients werden mit dem Java-Compiler *javac* übersetzt und damit die von einer VM interpretierbare *class*-Files generiert.

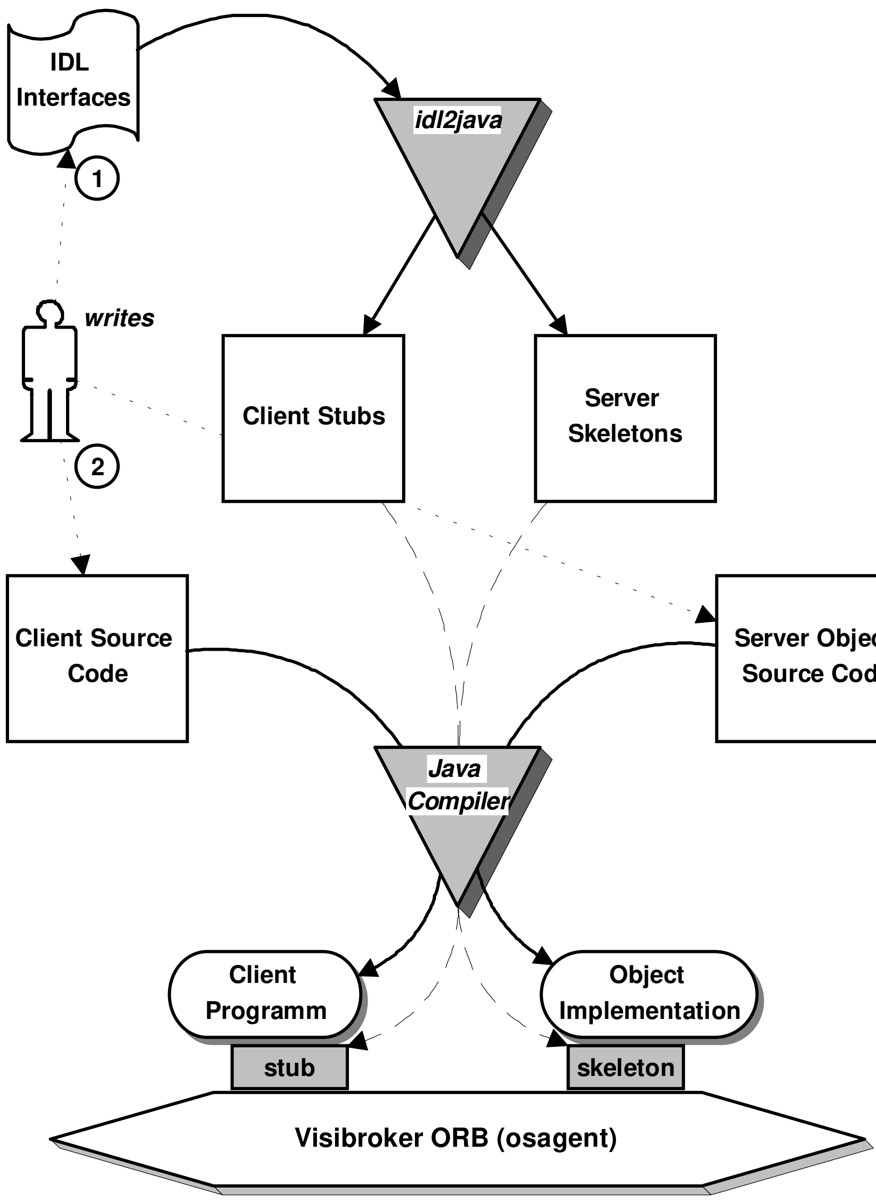

Abbildung 6-1: Die Entwicklung von CORBA Objekten

Ein Entwickler greift demzufolge nur ein zwei Stellen im Entwicklungsprozeß ein, um die nötigen CORBA Objekte zu generieren.

Die folgenden Dateien werden von dem *idl2java*-Compiler bei der Übersetzung eines IDL-Files generiert (<*InterfaceName*> steht nachfolgend für den *Namen* eines in IDL spezifizierten Interfaces):

- <*InterfaceName*>`.java`: Diese Datei enthält ein Java-Interface, das dem IDL-Interface entspricht.
- <*InterfaceName*>`Helper.java`: Diese Klasse enthält nützliche Methoden, wie z.B. die *narrow()*-Methode, um ein beliebiges CORBA-Objekt auf das <*InterfaceName*>-Interface zu casten. Erst dann ist es möglich die Methoden der <*InterfaceName*>-Schnittstelle aufzurufen.
- <*InterfaceName*>`Holder.java`: Diese Klasse wird für die Parameterübergabe verwendet.
- `_`<*InterfaceName*>`ImplBase.java`: Dies ist die Server-Skeleton-Klasse, die bereits von den für CORBA relevanten Java-Klassen erbt. Tatsächlich ist die Klasse abstrakt, da sie bereits die Implementierung des Interfaces <*InterfaceName*> zwar anzeigt, aber nicht durchführt. Dies muß von den Unterklassen übernommen werden. I.d.R. sind alle Server-Objektklassen, die vom Entwickler implementiert werden, *Unterklassen* dieser Klasse.
- `_tie_`<*InterfaceName*>`.java`: Diese Klasse wird benötigt, falls die Server-Objektklasse *nicht* von der `_`<*InterfaceName*>`ImplBase`-Klasse erben kann (siehe Abschnitt 6.2.2).
- <*InterfaceName*>`Operations.java`: Dieses Interface wird im Zusammenhang mit der `_tie_`<*InterfaceName*>-Klasse benötigt.
- `_portable_stub_`<*InterfaceName*>`.java`: Diese Klasse entspricht dem Client-Stub.

6.2.2 Der Tie-Mechanismus

Im vorhergehenden Abschnitt wurde bereits erwähnt, daß die Objektklasse des Servers i.d.R. eine Unterklasse der generierten `_`<*InterfaceName*>`ImplBase`-Klasse ist. Dadurch ist es möglich eine Instanz der Klasse *direkt* beim ORB anzumelden. Nicht selten tritt aber die Notwendigkeit ein, daß die Objektklasse des Servers eine generische Beziehung zu einer weiteren Oberklasse aufweisen muß. Da aber in *Java* keine Mehrfachvererbung unterstützt wird, bietet die Entwicklungsumgebung des Visibrokers [Visi97a] einen Workaround mit dem sog. *Tie-Mechanismus* an:

- Die Objektklasse des Servers erbt von einer beliebigen Klasse und *nicht* von der `_`<*InterfaceName*>`ImplBase`-Klasse.
- Die Objektklasse des Servers implementiert *nicht* mehr das Java-Interface <*InterfaceName*> sondern statt dessen das <*InterfaceName*>`Operations`-Interface.
- Soll eine Instanz der Klasse dem ORB zur Anmeldung übergeben werden, so kann dies nicht mehr *direkt* geschehen. Statt dessen wird zunächst eine Instanz der

`_tie_<InterfaceName>`-Klasse kreiert, der als Parameter eine Instanz der ursprünglichen Objektklasse des Servers übergeben wird. Die so instanziierte _tie_-Klasse kann daraufhin beim ORB angemeldet werden.

In Abbildung 6-2 ist die Beziehung zwischen den einzelnen Objektklassen am Beispiel der *OperationalPolicyImpl*-Klasse veranschaulicht.

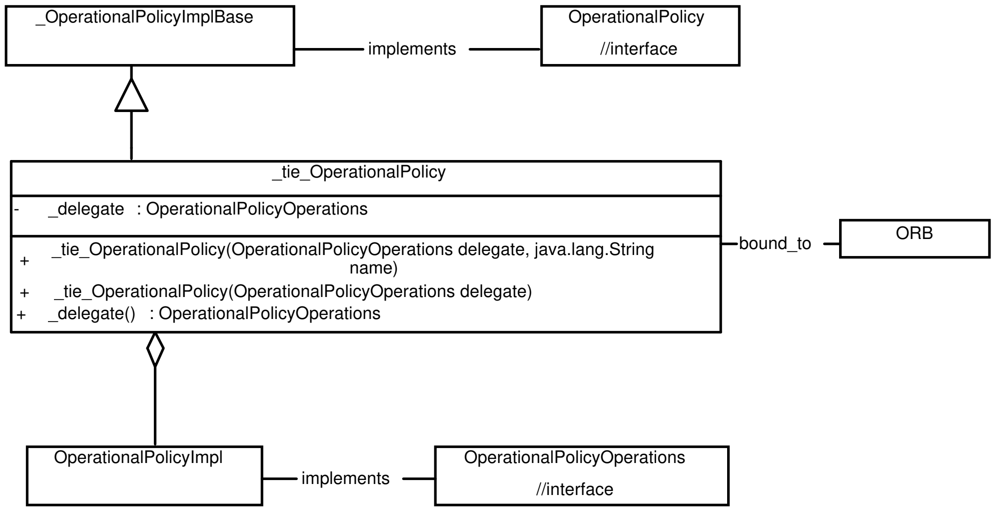

Abbildung 6-2: Der Tie-Mechanismus

Bei allen Klassen, die in Kapitel 5 das entsprechende `<IDL-Interface>Operations`-Interface implementieren, muß der Tie-Mechanismus angewendet werden, um das Objekt dem ORB bekannt zu machen.

6.3 Programmierwerkzeuge

Im Rahmen dieser Diplomarbeit wurden als Programmierwerkzeuge zur Softwareentwicklung das CASE-Tool *Software through Pictures (StP)* und der *Java Compiler Compiler (JavaCC)* verwendet, die in den nun folgenden Abschnitten kurz vorgestellt werden.

6.3.1 Software through Pictures (StP)

Das CASE-Tool *Software through Pictures (StP) Version 2.4.2* der Firma *Aonix* [Aoni97] unterstützt Software-Entwickler beim Entwurf objektorientierter Systeme durch den Einsatz unterschiedlicher Modellierungstechniken wie z.B. OMT. Für die tatsächliche Modellierung wird für jeden Diagrammtyp ein eigener Modellierungseditor angeboten. Für die Spezifizierung der Klassenattribute und -methoden innerhalb eines Klassendiagramms, wird zusätzlich ein *classtable*-Editor zur Verfügung gestellt, in dem u.a. auch bereits Programmiersprachen-spezifische Elemente festgelegt werden können. Die dabei erstellten Klassen, Objekte und Assoziationen werden in einer gemeinsamen *Repository* gespeichert. Auf diese Weise ist es möglich, sich während des Entwurfs eines Diagramms auf bereits entwickelte Objekte und deren Assoziationen zu beziehen. Dem Entwickler bietet sich zudem nach der Fertigstellung eines Systemmodells die Möglichkeit, dieses in konkrete Programmskelette einer objektorientierten Programmiersprache wie C++, Java oder IDL umzusetzen.

Im Rahmen dieser Diplomarbeit begleitete StP den ganzen Entwicklungsprozeß der hier vorgestellten Managementanwendung. Die in Kapitel 5 dargestellten Diagramme wurden ebenfalls mit StP erstellt.

6.3.2 Der Java Compiler Compiler (JavaCC)

In Kapitel 3 wurde die Grammatik der PDL für die Spezifizierung von operationalen Policies in der EBNF vorgestellt. Mit der Unterstützung von syntaktischen Programmierwerkzeugen [Schm95] ist es nun möglich, mit einer in EBNF spezifizierten Grammatik automatisch ein Programm für die lexikalische und syntaktische Analyse zu generieren. Die so generierten Analyseprogramme werden als *Parser* bezeichnet. Die Aufgabe eines Parsers besteht zunächst nur darin, einen Eingabetext dahingehend zu überprüfen, ob dieser der gegebenen Grammatik entspricht. Dies geschieht i.d.R. dadurch, daß ein *Syntaxbaum* aufgebaut und abgearbeitet wird, dessen Knoten den Nichtterminalsymbolen und dessen Blätter den Terminalsymbolen entsprechen. Mit den meisten syntaktischen Programmierwerkzeugen ist es allerdings zusätzlich möglich, dem Parser gewisse Anweisungen hinzuzufügen, die an bestimmten, festgelegten Stellen des Syntaxbaums ausgeführt werden. Die so spezifizierten Anweisungen werden als *Evaluierungsregeln* bezeichnet. Auf diese Weise kann folglich ein Programm generiert werden, das die operationalen Policies *interpretiert*.

Der *Java Compiler Compiler (JavaCC)* [JaCC98] ist ein derartiges Programmierwerkzeug, das Parser in der Programmiersprache Java[10] generiert. Für die nachstehende Kurzbeschreibung der Merkmale des JavaCC werden gewisse Grundkenntnisse des Compilerbaus vorausgesetzt:

- Es werden LL(k)-Parser generiert. Dies impliziert eine Top-Down-Analyse des Eingabetextes, das das Weiterreichen von Variablen entlang des Syntaxbaums in beiden Richtungen (auf- und abwärts) ermöglicht. Zudem erleichtert die Top-Down-Analyse das Debuggen von Grammatiken und bietet ebenso die Möglichkeit das Parsen mit einem beliebigen Nichtterminalsymbol zu beginnen.
- Sowohl die lexikalischen als auch die syntaktischen Regeln werden in *einer* Datei spezifiziert. Dies verbessert die Lesbarkeit der Grammatik und erleichtert die Erstellung des Parsers.
- Lexikalische Zustände können ebenfalls definiert werden, um den Verlauf der Syntaxanalyse zu steuern.
- Die Eigenschaften des zu generierenden Parsers können über das Setzen der vielfältigen Optionsparameter beeinflußt werden.
- Die während der Syntaxanalyse eines Eingabetextes auftretenden Abweichungen von der festgelegten Grammatik werden detailliert dokumentiert, so daß der Fehler im Eingabestrom schnell gefunden werden kann.

---

[10] Neben *JavaCC* existiert für die Parsergenerierung in Java zudem das Tool *SableCC*, das dem Aufbau eher dem Unix-Tool *yacc* ähnelt (modular, Bottom-Up-Analyse).

```
options {
...
}
-------------------------------
PARSER_BEGIN(parser_name)
 ...
 class parser_name ... {
 ...
}
...
PARSER_END(parser_name)
-------------------------------
TOKEN :
{
  < AND: "&&" >
  ...
}
...
-------------------------------
void input () :
 { ... }
  {
    expansion_choices
   }
...
```

Abbildung 6-3: Aufbau eines JavaCC Programms

Ein in JavaCC geschriebenes Programm hat das in Abbildung 6-3 dargestellte Aussehen und besteht aus den folgenden Komponenten:

1. An erster Stelle wird eine Liste von Optionen angegeben, die das Verhalten des zu generierenden Parsers beeinflussen. Hier können v.a. Optimierungen vorgenommen werden.

2. Als nächstes wird ein Java-Block festgelegt, der direkt in das zu generierende *java*-File übernommen wird. Der Block beginnt mit *PARSER_BEGIN* und enthält in Klammern den Namen des Parsers, der zugleich der Name der zu generierenden Klasse ist. Das Ende dieses Blocks wird mit *PARSER_END* markiert. Dazwischen können der Package-Name, *import*-Anweisungen und Klassenattribute spezifiziert werden.

3. Der dritte Teil eines JavaCC Programms besteht aus den lexikalischen Regeln der Grammatik. Es werden *Tokens* definiert, die, um sie besser zu unterscheiden, immer groß geschrieben werden.

4. Der letzte Teil beinhaltet die Produktionsregeln der Grammatik. Eine Produktionsregel ist durch einen Doppelpunkt zweigeteilt, bestehend aus genau einem *Nichtterminalsymbol* auf der linken Seite und mindestens einer *Expansionregel* auf der rechten

Seite. Die Nichtterminalsymbole werden in Form von Funktionssignaturen geschrieben, d.h. es werden Rückgabetyp und Parameter bestimmt. Damit können Variablenwerte durchgereicht werden. Eine Expansionsregel ist in geschweiften Klammern gefaßt und besteht wiederum aus einer Reihe von Nichtterminalsymbolen und Terminalsymbolen. Der Expansionsregel wird ein, ebenfalls in geschweiften Klammern umfaßter Java-Block vorangestellt, in dem z.B. Variablen definiert werden können. Die Expansionsregel kann selber durch Java-Blöcke „unterbrochen" werden, in denen Java-Anweisungen spezifiziert werden. Diese Java-Blöcke entsprechen damit den Evaluierungsregeln.

In Abbildung 6-4 ist der Parser-Entwicklungsprozeß mit JavaCC veranschaulicht und lautet wie folgt:

(1) Es wird ein JavaCC Programm nach dem in Abbildung 6-3 dargestellten Schema geschrieben.

(2) Der JavaCC-Compiler übersetzt das JavaCC Programm in Java Source-Code, indem dieser die dafür notwendigen Klassen generiert. Die Nichtterminalsymbole werden in Methoden der Parser-Klasse umgewandelt. So ist es möglich mit einem beliebigen Nichtterminalsymbol das Parsen zu beginnen.

(3) Durch ein Übersetzen der *java*-Files mit dem *javac*-Compiler werden die *class*-Files generiert.

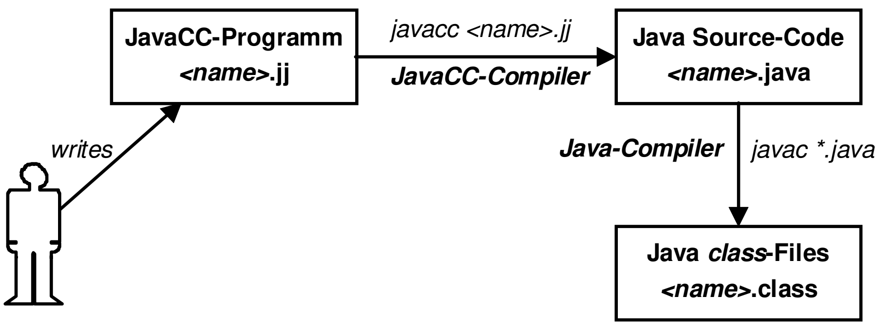

Abbildung 6-4: Der Parser Entwicklungsprozeß

JavaCC wurde zur Erstellung des *Policy Interpreters*, *Constraint Interpreters* und *Action Interpreters* eingesetzt. Der Policy Interpreter generiert, mit einer in PDL formulierten Policy als Eingabestrom, die erforderlichen Enforcement Objects und gibt u.a. deren IDs als Rückgabewert zurück. Der Constraint Interpreter evaluiert hingegen nur den spezifizierten Constraint einer operationalen Policy und liefert einen booleschen Wert zurück. Der Action Interpreter wertet den Aktionsbereich aus und feuert die spezifizierten IDL Operationen, indem diese per *Dynamic Invocation* aufgerufen werden. Diese Separierung erfolgte v.a. aus Gründen der Effizienz, besseren Übersichtlichkeit der Aufgabenaufteilung und der besseren Lesbarkeit der JavaCC Programme. Grundsätzlich wäre es möglich, die drei Parser in einem zu vereinen und die jeweilige Funktionalität des so erhaltenen Interpreters über unterschiedliche Startpunkte und lexikalische Zustände zu steuern. Dadurch wird jedoch die Komplexität des JavaCC Programms um ein Vielfaches erhöht, so daß einerseits das Debuggen und andererseits das Erweitern der PDL nicht unerheblich erschwert wird. Bei der hier durchgeführten Aufteilung ist jedoch zu beachten, daß Änderungen der PDL in allen drei Parsern durchzuführen sind.

6.4 Das Starten der Agenten

Die nachstehend angeführten Aufrufe sind, soweit nichts anderes angegeben, im folgendem Verzeichnis zu finden: `/usr/local/mnmcommon/vbroker-3.0/bin`

Folgende Schritte müssen für das erfolgreiche Starten der einzelnen Agenten der verteilten Managementanwendung durchgeführt:

(1) Der *Smart Agent* wird mit dem folgenden Skript hochgefahren (ausschließlich auf Sun Solaris):

```
/usr/local/mnmcommon/bin/vbjtcsh
```

Dies startet den *osagent* in einer separaten Shell im *verbose*-Mode, so daß u.a. die Kommunikation zwischen den einzelnen Objekten verfolgt werden kann. Für alle nachfolgend gestarteten Agenten muß beachtet werden, daß die `$OSAGENT_PORT`-Umgebungsvariable auf denselben Wert gesetzt ist.

(2) Der Agent des *Naming Service* wird folgendermaßen hochgefahren:

```
vbj -DORBservices=CosNaming \
com.visigenic.vbroker.services.CosNaming.ExtFactory \
myFactory /tmp/naming.log
```

myFactory ist der Name des *root Context*, der *jedem* Client und Server mit dem Aufruf der VM übergeben werden muß, damit diese den *Naming Service* nutzen können. Desweiteren bezeichnet */tmp/naming.log* die Logging-Datei des Agenten. Darin wird der während einer Session kreierte *naming graph* gespeichert. Ist bereits eine Logging-Datei vorhanden, so wird der darin enthaltene *naming graph* wiederhergestellt. Die Logging-Datei wird von dem Agenten bei Änderungen ständig aktualisiert. Bindet sich ein Server Objekt an einen Namen, aber löscht diese Bindung nicht wieder, wenn dieser nicht mehr auf dem ORB ist, so bleibt die Bindung trotzdem weiter erhalten und ein Objekt kann sich *nicht* mehr an denselben Namen binden. In diesem Fall muß die Logging-Datei gelöscht werden, damit der Name wieder freigegeben wird.

(3) Mit dem folgendem Aufruf wird ein vorher festgelegter *naming graph* aufgebaut, der die Domänen im zu managenden Bereich widerspiegelt:

```
vbj -DORBservices=CosNaming -DSVCnameroot=myFactory \
de.unimuenchen.informatik.mnm.ncscontrol.domain.\
DomainBuilder
```

Dieser Schritt muß immer nur dann durchgeführt werden, falls die Logging-Datei des *Naming Service* gelöscht wurde.

(4) Der folgende Aufruf startet die *Interface Repository*:

```
irep myRepository /tmp/ir.log
```

Mit *myRepository* wird der Name der kreierten Interface Repository festgelegt. */tmp/ir.log* ist der Name der Logging-Datei der Interface Repository, in der alle während einer Session hinzugefügten IDL-Interfaces gespeichert werden. Während des Starvorgangs wird diese Datei gelesen und alle darin enthaltenen Interfaces in die Datenbank eingefügt. Für die korrekte Funktionsweise der Managementanwendung ist es notwendig, daß sich *alle* IDL-Interfaces derjenigen *Managementagenten* in der Datenbank befinden, die als Zielobjekt einer operationalen Policy in Frage kommen.

Mit folgendem Befehl können neue Interfaces hinzugefügt werden:

```
idl2ir -ir myRepository <IDL-File>
```

<IDL-File> steht selbstverständlich für den Namen derjenigen Datei, die die Definition des Interfaces in IDL enthält.

(5) Als nächstes wird der *AgentInitChannel* hochgefahren, an dem sich alle Agenten an- und abmelden:

```
vbj -prof -DORBservices=CosNaming -DSVCnameroot=myFactory \
de.unimuenchen.informatik.mnm.masa.event.AgentChannel \
-Dmasa.propfile=/proj/fagent/masa_0.2/masa.properties
```

(6) Der folgende Aufruf startet ein *MASIF-Agentensystem* (siehe auch [Kemp98]) auf dem Host:

```
vbj -prof -DORBservices=CosNaming -DSVCnameroot=myFactory \
de.unimuenchen.informatik.mnm.masa.agentSystem.AgentSystem \
-Dmasa.propfile=/proj/fagent/masa_0.2/masa.properties
```

Die nächsten drei Schritte ermöglichen das Hochfahren der Agenten der Managementanwendung, die nur auf denjenigen Hosts gestartet werden können, auf denen bereits ein Agentensystem läuft. Die Reihenfolge der Schritte ist unbedingt einzuhalten, um die korrekte Funktionsweise der Agenten zu ermöglichen.

(7) Mit dem folgendem Aufruf wird ein Kommandozeilen-Interface für den *PersistenceServiceMobileAgent* gestartet:

```
vbj -DORBservices=CosNaming -DSVCnameroot=myFactory \
de.unimuenchen.informatik.mnm.ncscontrol.persistence.\
PersistenceServiceUI
```

Das Kommandozeilen-Interface bietet die Möglichkeit den Agenten zu starten, zu migrieren und wieder zu terminieren. Desweiteren können Methoden der *PersistenceService*-Schnittstelle aufgerufen werden, um sich einen Überblick über den gegenwärtigen Zustand des Agenten zu verschaffen. Das UI liest das *Properties*-File */proj/fagent/NoCScontrol/properties/psma.props* ein, so daß diese Datei vorhanden sein muß, damit der Agent erfolgreich gestartet werden kann.

(8) Hiermit wird das Kommandozeilen-Interface des *EnforcementObjectFactoryMobileAgent* aufgerufen:

```
vbj -DORBservices=CosNaming -DSVCnameroot=myFactory \
de.unimuenchen.informatik.mnm.ncscontrol.enforcement.\
EnforcementServiceUI
```

Auch für diesen Agenten wird ein *Properties*-File eingelesen:
/proj/fagent/NoCScontrol/properties/eofma.props

(9) Der folgende Aufruf startet das Kommandozeilen-Interface des *PolicyFactoryMobileAgent*:

```
vbj -DORBservices=CosNaming -DSVCnameroot=myFactory \
de.unimuenchen.informatik.mnm.ncscontrol.policy.\
PolicyServiceUI
```

Der Name des *Properties*-Files für dieses UI ist:
/proj/fagent/NoCScontrol/properties/pfma.props.

Damit die *class*-Files von der VM gefunden werden können, muß der Klassenpfad richtig gesetzt sein. Die folgenden Klassen müssen im Klassenpfad enthalten sein:

```
/proj/java/jdk1.1-solaris/jdk1.1.6/lib/classes.zip:\
/usr/local/mnmcommon/vbroker-3.0/lib/vbjcosnm.jar:\
/usr/local/mnmcommon/vbroker-3.0/lib/vbjcosev.jar:\
/usr/local/mnmcommon/vbroker-3.0/lib/vbj30.jar:\
/opt/JMAPI/classes:/opt/JMAPI/examples:\
/proj/java/JMAPI/jdbcDriver/FastForward/jdk113:\
/users/stud/schiller/schi98/Sourcen/NomadicSystems:\
/usr/local/mnmcommon/lib/advent:\
/proj/evcorr/public-htdocs/prototype-0.3/classes:\
/proj/fagent/CORBAServices/classes:\
/proj/fagent/masa_0.2_proto/classes:\
/proj/fagent/NoCScontrol/classes:\
```

Der Klassenpfad kann entweder über die `$CLASSPATH`-Umgebungsvariable gesetzt werden oder über die Option `-classpath` dem *vbj* übergeben werden.

6.5 Erfahrungsbericht

Java erwies sich tatsächlich als eine robuste und sehr vielseitige Programmiersprache. Die anfänglichen Bedenken dieser Sprache gegenüber, wie z.B. daß keine Zeiger verwendet werden, keine Mehrfachvererbung möglich ist und daß die Ausführung der Programme sehr langsam durch deren Interpretierung sein würde, erwiesen sich alle als unnötig und falsch. Das Debuggen von Java-Programmen gestaltet sich sehr einfach, da bei einem Ausführungsfehler die genaue Zeile im Source-Code angezeigt wird und diese Stelle auch immer den wahren Grund für den Fehler enthält. Ein langwieriges Suchen nach Fehlern, die durch Pointermanipulationen entstehen und erst 100 Zeilen später zum Absturz führen, erübrigt sich. Geschwindigkeitseinbußen während der Ausführung der Programme durch die VM-Interpretierung sind bei der hier entwickelten verteilten Anwendung kaum spürbar, da im vorliegenden Fall eher die Netzkommunikation zwischen den Objekten der wahre Flaschenhals ist. Einzig und allein die nicht unterstützte Mehrfachvererbung führte zu mancher „kreativen“ Denkpause, in der Workarounds überlegt werden mußten. Ist man sich allerdings dieser Tatsache erst wirklich bewußt, erscheint diese Eigenschaft nicht mehr wirklich als Manko. Java sammelt weiterhin Pluspunkte bei der Systemunabhängigkeit der Ausführung, der umfangreichen API und der ausführlichen Online-Dokumentation, die oft bei Unsicherheiten weiterhalf.

Die Parserentwicklung mit JavaCC ist ebenfalls sehr zügig vorangeschritten. Die Aufteilung des Parser-Codes ist intuitiver und wesentlich leichter zu Debuggen als bei den Unix-C-Tools *lex* und *yacc*. Die Möglichkeit, Variablenwerte sowohl auf- als auch abwärts entlang des Syntaxbaums weiterzureichen, ist im vorliegenden Fall der Grund für die schnelle Entwicklung der Parser gewesen. Die Nichtterminalsymbole als Funktionen darzustellen, ermöglichte zudem eine intuitivere Betrachtungsweise auf die Vorgehensweise des Parsers. Es stellte sich jedoch heraus, daß mit anwachsendem Umfang des hinzugefügten Codes die Übersicht leicht verloren geht und der anfängliche Vorteil der *All-in-One*-Philosophie sich zum Nachteil entwickelt. Trotzdem ist JavaCC empfehlenswert für den Einsatz als Tool zur Parserentwicklung.

Der *Visibroker 3.0 for Java* liefert eine stabile Entwicklungsumgebung, hat aber einen gravierenden Nachteil: die Dokumentation. Für die Entwicklung einfacher CORBA Objekte ist die Dokumentation gut und das *Programmer's Guide* [Visi97a] bietet für Einsteiger sogar eine sehr gute Einführung. Insgesamt gesehen, bleibt aber die Dokumentation auf diesem Niveau stehen. Sobald die CORBA Objekte komplexere Strukturen aufweisen und u.a. es deswegen notwendig ist, nach bestimmten Stichwörtern zu suchen, versagt v.a. das *Reference Manual* [Visi97]. Einige Stichwörter sind im Register nicht eingetragen, so daß oft das komplette Handbuch durchgeblättert werden muß, um die gesuchte Information zu finden. Methoden, die nur beiläufig in einem anderen Kontext erwähnt werden, sind erst gar nicht ausführlich dokumentiert, so daß in diesen Momenten oft nur die Möglichkeit bleibt, das Verhalten dieser Methoden auszutesten. Nicht selten kam es vor, daß das beschriebene Verhalten von Methoden nicht mit dem in der Dokumentation übereinstimmte. Dies führte oftmals zu Verwirrungen und verlängerte dadurch unnötig den Zeitraum für die Entwicklung der Objekte. Dies setzte sich bei der Dokumentation des *Naming* und *Event Service* [Visi97b] fort, die eindeutig zu knapp ausfiel. Die CORBA-Spezifikation der beiden Services [OMG97b] ist ausführlicher und leichter zu verstehen als die mitgelieferte Dokumentation. Die Hoffnung bleibt, daß die Dokumentation in den noch kommenden Versionen des Visibrokers verbessert wird.

Auch *StP* ist sehr zwiespältig zu bewerten. Einerseits ist der Aufbau dieses CASE-Tools u.a. durch die gemeinsame Repository, sehr gelungen. Auch die vielen Möglichkeiten, die sich mit StP bieten, um Quellcode-Skelette zu generieren, ist sehr lobenswert. Die Dokumentation [Aoni97] ist übersichtlich, leicht verständlich und bietet einen schnellen Einstieg in die Bedienung von StP. Aber genau hier setzt auch die Kritik an: die Bedienung dieses Tools ist nicht intuitiv. Es ist z.B. möglich, um nur ein Beispiel zu nennen, im *classtable*-Editor über das Menü der rechten Maustaste, Zeilen der Tabelle zu löschen, aber nicht neue einzufügen. Diese Funktion muß umständlich über die Menüzeile des Fensters aufgerufen werden. Dies ist nur ein Beispiel von vielen, die die gesamte Bedienung von StP zu einer mühseligen Prozedur machen. Zudem ist StP beim Laden und Speichern von Diagrammen und Tabellen sehr langsam. Dies stört anfänglich bei der ersten Modellierung des Systems nicht, wird aber lästig, wenn später nur Kleinigkeiten, wie Klassenattribute, geändert werden und mehr Zeit beim Warten als mit der tatsächlichen Arbeit verbracht wird. Trotzdem ist der Einsatz von StP zu empfehlen, da die Vorteile, die sich durch das spezielle Software-Engineering-Vorgehen ergeben, eindeutig überwiegen.

6.6 Zusammenfassung

In diesem Kapitel wurden Implementierungsgesichtspunkte betrachtet. Der Schwerpunkt lag auf der Darstellung der für die Entwicklung der Anwendung eingesetzten Programmierwerkzeuge. Die Programmiersprache *Java* und der Parsergenerator *JavaCC* wurden hierbei näher erläutert. Zum Ende erfolgte eine Beschreibung der Startprozedur der entwickelten Agenten. Abgerundet wurde dieses Kapitel schließlich mit einem Erfahrungsbericht des Autors.

Kapitel 7

Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde untersucht, wie dem Aufgabenbereich des Konfigurationsmanagements nomadischer Systeme mit Policy-basiertem Management begegnet werden kann. Der Schwerpunkt der Arbeit lag bei der Spezifizierung einer formalen Sprache für Policies, der Untersuchung der Managementarchitektur sowie der Entwicklung und prototypischen Implementierung einer Managementanwendung, die dem erstellten Konzept genügt.

Während der Suche nach bereits vorhandenen Ansätzen und Produkten mußte festgestellt werden, daß die bisherige Integration nomadischer Systeme im Netz- und Systemmanagement noch unbefriedigend ist. Die bereits vorhandenen Software-Produkte beschränken sich auf die reine Etablierung der Kommunikationsgrundlage zwischen den nomadischen Systemen und den im Netz vorhandenen Endgeräten. Ein echtes Management dieser Systeme, d.h. der Steuerung deren Verhaltens, ist mit keinem dieser Produkte möglich. Zudem fehlt allen untersuchten Software-Produkten eine standardisierte Managementschnittstelle, so daß auch deren Integration in ein bereits vorhandenes Managementsystem nicht ohne weiteres möglich ist. Die Forschung auf dem Gebiet des Policy-basierten Managements beschäftigt sich allgemein mit dem Management verteilter Systeme. Die in den verschieden Forschungsarbeiten entwickelten Formalisierungen von Policies sind deswegen nicht speziell auf bewegliche Managementobjekte ausgelegt, so daß an diesem Punkt angesetzt werden konnte, um eine neue Sprache speziell für das Management nomadischer Systeme zu spezifizieren. Ausgehend von einer Policy-Hierarchie, die u.a. den Rahmen des Entwicklungsprozesses von einer High-Level-Policy zu einer Low-Level-Policy festlegt, wurden Templates für zielorientierte Policies definiert und eine neue, dem Management nomadischer Systeme angepaßte Sprache für operationale Policies spezifiziert. Mit Hilfe des syntaktischen Programmierwerkzeugs *JavaCC* wurde ein Interpreter implementiert, der die in PDL verfaßten Policies in konkrete Managementanweisungen umsetzt.

Basierend auf einem Beispielszenario wurde ein erster Lösungsansatz der Managementaufgabe erarbeitet. Daraus ergaben sich bereits die ersten Anforderungen an die Managementarchitektur, die mit dem Bedarf nach Skalierbarkeit und Ausfallsicherheit der Managementanwendung schließlich die verwirklichte Architektur ergab. Die spezifizierte MA wurde in einer CORBA-Umgebung umgesetzt, die sich als besonders geeignete Middleware für Managementzwecke herausstellte. Als Implementierungssprache diente *Java*, die sich ebenfalls durch ihre Systemunabhängigkeit als ideale Programmiersprache für Anwendungen in einem heterogenen Umfeld erwies. Die Anwendung ist in den *Policy Service*, *Policy Enforcement Service* und *Persistence Service* aufgeteilt. Diese drei Komponenten wurden als CORBA Server in Form von *Mobile Agents*, die der *MASIF*-Spezifikation [OMG97c] entsprechen, verwirklicht, um so die Möglichkeit zu haben, diese während der Laufzeit auf ein anderes System im Netzwerk zu migrieren. Dies erhöht die Ausfallsicherheit und Skalierbarkeit der verteilten Anwendung zusätzlich.

Der modulare Aufbau sowohl der MA als auch der Anwendung, ermöglicht ein unkompliziertes Erweitern des Managementsystems. Momentan werden am Lehrstuhl Managementagenten für DHCP Server [Demm98], Mail Transfer Hubs [Coeh98] und PC Switchs [Allg98] entwickelt, die die Voraussetzungen für eine Zusammenarbeit mit der Managementanwendung erfüllen. Um ein umfassendes Management zu ermöglichen, ist es selbstverständlich notwendig, weitere Agenten zu erstellen.

Einer der interessantesten Bereiche für das NoS-Management ist der des Accountings, der in dieser Arbeit nur genannt und nicht weiterverfolgt werden konnte. Das Accounting stellt eine nicht triviale Aufgabe an das MS, da nicht nur der gemanagte Bereich betroffen ist, in dem das NoS angeschlossen wird, sondern zusätzlich auch der Heimatbereich des NoS. Hier muß ein wirksames Konzept entwickelt werden, das einerseits den Datenaustausch zwischen den Managementanwendungen regelt und andererseits verschiedene Abrechnungsarten unterstützt.

Ein weiterer Aspekt, der nicht weiterverfolgt werden konnte, ist das Sicherheitsmanagement. Es tritt auf mehreren Ebenen der Bedarf der Authentifizierung von Objekten auf:

- *Authentifizierung der nomadischen Systeme*: U.a. wurde bereits im Beispielszenario die Notwendigkeit der Authentifizierung der nomadischen Systeme begründet, um z.B. darauf abstützend die Ressourcennutzung zu steuern. Dies könnte durch bereits existierende Authentifizierungsserver wie z.B. RADIUS [RFC2138] geschehen. Eine weitere Möglichkeit besteht in der Implementierung des *CORBA Security Service* ([OMG97b], S.15-1 ff), der diesen Aspekt in der Spezifikation miteinbezieht.

- *Authentifizierung der Managementanwendung gegenüber den Managementagenten*: Hier offenbart sich eine ernstzunehmende Sicherheitslücke im Managementsystem. Da die Managementanwendung das Verhalten der NoS steuert, indem es auf den Managementagenten agiert, muß sichergestellt werden, daß die Managementanwendung auch tatsächlich dafür autorisiert ist. Um dies zu gewährleisten, muß ein entsprechendes Authentifizierungsprotokoll zwischen Anwendung und Agenten eingeführt werden. Es existieren mehrere standardisierte Protokolle (siehe [Eck97]), die in das MS integriert werden könnten. Dieser Aspekt würde allerdings ebenfalls durch eine Implementierung des oben bereits erwähnten *CORBA Security Services* abgedeckt werden.

- *gegenseitige Authentifizierung der Managementanwendungen*: Um z.B. sicherheitssensible Daten zwischen zwei Managementanwendungen aus verschiedenen Managementdomänen auszutauschen, ist eine vorherige gegenseitige Authentifizierung notwendig. Dies könnte entweder durch die Integration eines eigenen Authentifizierungsprotokolls gelöst werden oder ebenfalls durch den *CORBA Security Service*.

- *Authentifizierung der User der Anwendung*: Es liegt auf der Hand, daß nicht jeder Policies erstellen darf, der Zugang zum System hat. Hier wäre eine Benutzerverwaltung mit Authentifizierung notwendig. Allerdings würde dieser Bereich ebenfalls durch den *Security Service* abgedeckt werden.

- *Gegenseitige Authentifizierung der Komponenten der Anwendung*: Auch an dieser Stelle eröffnet sich eine Sicherheitslücke. Momentan ist es z.B. noch möglich, daß ein nicht autorisiertes Objekt auf eine Datenbank des Persistence Service zugreift und deren Inhalt verändert. Um solche Aktionen ausschließen zu können, muß auch an dieser Stelle eine Authentifizierung der Objekte stattfinden. Auch dies wäre durch den *Security Service* realisierbar.

Wie aus dieser kurzen Darstellung zu ersehen ist, würde sich die Umsetzung des *CORBA Security Service* in eine konkrete Implementierung anbieten, um alle genannten Sicherheitslükken zu schließen.

Das bisherige Kommandozeilen-Interface der Anwendung, u.a. zur Erstellung der Policies, ist selbstverständlich noch unbefriedigend. Es wird aber bereits im Rahmen eines Fortgeschrittenenpraktikums [Goli98] eine GUI in Form von Java-Applets für die Komponenten der Anwendung erstellt.

Ein weiterer Bereich, der im Rahmen dieser Arbeit unbetrachtet blieb, ist die Verwaltung der Operationen der Managementagenten, die bisher einfach als bekannt vorausgesetzt wurden. Hier wäre es denkbar, Operationsaufrufe zu standardisieren indem eine Menge von Standard-IDL-Interfaces definiert werden, von denen die Agenten jeweils eine Untermenge tatsächlich implementieren.

Die PDL erweist an einigen Stellen Erweiterungsmöglichkeiten, wie z.B. daß im Aktionsteil if-then-else-Anweisungen erlaubt werden könnten. In diesem Fall wäre auch die Spezifizierungen von Variablen denkbar. Es ist jedoch sicher, daß der Bedarf an Anpassungen der PDL aus dem laufenden Testbetrieb entstehen werden.

Ein nicht unerhebliches Problem bei der Formulierung von Policies stellt die Lösung von Konflikten dar. In dieser Arbeit konnten nur Ansätze und Denkanstöße betrachtet werden, die einer weiteren Untersuchung würdig wären. Zudem konnten die bereits erarbeiteten Konfliktlösungsstrategien nicht in den *Policy Service* integriert werden, so daß dies in weiteren Arbeiten noch umgesetzt werden sollte.

Ebenfalls wurde die große Bedeutung von Meta-Policies zur Erstellung und Verwaltung von Policies deutlich. Im Bereich der Policy-Erstellung werden sich die meisten Meta-Policies aus der Erfahrung im laufendem Betrieb ergeben. Der Bereich der Policy-Verwaltung mit Meta-Policies wird bereits am Lehrstuhl im Rahmen einer Diplomarbeit [Avit98] untersucht.

Abkürzungsverzeichnis

AE	aggregate entity
API	Application Programming Interface
BOA	Basic Object Adapter
CASE	Computer aided Software Engineering
CORBA	Common Object Request Broker Architecture
DB	Datenbank
DDNS	Dynamic Domain Name System
DHCP	Dynamic Host Configuration Protocol
DIB	Directory Information Base
DII	Dynamic Invocation Interface
DIT	Directory Information Tree
DNS	Domain Name System
DSI	Dynamic Skeleton Interface
EO	Enforcement Object
EOF	Enforcement Object Factory
EOFMA	Enforcement Object Factory Mobile Agent
FQDN	Fully Qualified Domain Name
GUI	Graphical User Interface
IDL	Interface Definition Language
IP	Internet Protocol
IR	Interface Repository
JDK	Java Development Kit
LAN	Local Area Network

MA	Managementarchitektur
MASIF	Mobile Agent System Interoperability Facilities
MO	Managementobjekt (Managed Object)
MS	Managementsystem
NoS	nomadische(s) System(e)
NoCS	Nomadic Computing System
OA	Object Adapter
OMA	Object Management Architecture
OMG	Object Management Group
OMT	Object Modeling Technique
OO	Objektorientierung, objektorientiert
ORB	Object Request Broker
PDA	Personal Digital Assistant
PDL	Policy Description Language
PFMA	Policy Factory Mobile Agent
PID	Persistent Object Identifier
PO	Policy Object
PPP	Point-to-Point Protocol
PPTP	Point-to-Point Tunneling Protocol
PSMA	Persistence Service Mobile Agent
RFC	Request for Comment
RFP	Request for Proposals
SS	Schnittstelle
StP	Software through Pictures
TCP	Transmission Control Protocol
TQL	Topology Query Language
UDP	User Datagram Protocol
UI	User Interface
UML	Unified Modeling Language
VLAN	Virtuelles LAN
VM	(Java) Virtual Machine
WAN	Wide Area Network

Literaturverzeichnis

[AIC97a] American Internet Corporation, *Network Registrar Getting Started Guide*, Release 1.1, Juni 1997.

[AIC97b] American Internet Corporation, *Network Registrar User's Guide*, Release 1.1, Juli 1997.

[Allg98] Peter Allgeyer, „Entwurf einer Managementschnittstelle für einen PC-basierten Switch“, Diplomarbeit, Technische Universität München, August 1998.

[Aoni96] Aonix, *Software through Pictures/Object Modeling Technique - Getting Started with StP/OMT Release 3.0*, Februar 1996.

[Aoni96a] Aonix, *Software through Pictures/Object Modeling Technique - Creating OMT Models Release 3.0*, Februar 1996.

[Aoni97] Aonix, *StP Core - Fundamentals of StP Release 2.4*, September 1997.

[Avit98] Marco Avitabile, „An Examination of Requirements for Meta-Policies in Policy-Based Management“, Diplomarbeit, Technische Universität München, November 1998.

[BEA98] BEA Systems, DSTC, Expersoft Corporation, Fujitsu Limited et al., „Notification Service, Joint Revised Submission“, 1998.

[Coeh98] Christian Coehn, „Integriertes Management verteilter Email-Systeme auf der Basis flexibler Managementagenten“, Diplomarbeit, Ludwig-Maximilians-Universität München, August 1998.

[CoMa97] Peter Coad und Mark Mayfield, *Java Design: Building Better Apps and Applets*, Prentic Hall PTR, 1997.

[Demm98] Simone Demmel, „Implementierung eines CORBA-basierten Managementagenten für DHCP Server“, Fortgeschrittenenpraktikum, Ludwig-Maximilians-Universität München, 1998.

[Eck97] Prof. Dr. C. Eckert, „Praktikum: Sichere Rechensysteme“, Vorlesungsskript, Technische Universität München, Sommersemester 1997.

[Flan97] David Flanagan, *Java in a Nutshell, Second Edition*, O'Reilly & Associates, 1997.

[FrRu96] Franz Fricke und Klaus-Hartwig Rube, *Betriebswirtschaftslehre*, Bayerischer Schulbuchverlag, 3. Auflage, 1996.

[Gars98] Markus Garschhammer, „Entwicklung eines Managementkonzepts für LDAP-integrierte Verzeichnisdienste“, Diplomarbeit, Technische Universität München, August 1998.

[GDL96] Ben Lancki Vipul Gupta, Abhijit Dixit, „Linux MobilIP“, http://anchor.cs.binghapton.edu/~mobileip/, Mai 1996.

[GHW 99] B. Gruschke, S. Heilbronner und N Wienold, „Managing Groups in Dynamic Networks“, In *To be published*, Juli 1998.

[Goh 98] Che Goh, „A Generic Approach to Policy Description in System Management“, Technischer Bericht, System Management Department, Hewlett Packard Laboratories, 1998.

[Goli98] Dimitrios Golias, „Entwicklung und Implementierung einer GUI für das Policy-basierte Management von nomadischen Systemen“, Fortgeschrittenenpraktikum, Technische Universität München, 1998.

[GoYe96] James Gosling, Frank Yellin und The Java Team, *The Java Application Programming Interface, Volume 1*, Addison-Wesley Publishing Company, 1996.

[Hagg96] Prof. Dr. R. Haggenmüller, „Methoden der Software-Entwicklung“, Vorlesungsskript, Ludwig-Maximilians-Universität München, Sommersemester 1996.

[Hals96] Fred Halsall, *Data Communications, Computer Networks and Open Systems*, Addison Wesley, 4. Auflage, 1996.

[HeAb93] Heinz-Gerd Hegering und Sebastian Abeck, *Integriertes Netz- und Systemmanagement*, Addison-Wesley, 1. Auflage, 1993.

[Heil97] S. Heilbronner, „DHCP und das Management nomadischer Systeme in Intranets“, In *Proceedings of the Opennet' 97*, Berlin, Germany, November 1997, Internet Society German Chapter.

[Heil98a] S. Heilbronner, „Requirements for Policy-based Management of Nomadic Computing Systems“, In *Proceedings of the 8th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 98)*, Newark, DE, USA, Oktober 1998.

[HePa97] Hewlett-Packard Company, „Response to Topology Service RFP, Revised Submission“, 1997.

[HLKB97] Stephen Howard, Hanan Lutfiyya, Michael Katchabaw und Michael Bauer, „Supporting Dynamic Policy Change Using CORBA System Management Facilities“, Technischer Bericht, Departement of Computer Science, The University of Western Ontario, 1997.

[JaCC98] Sun Microsystems, *Description of the JavaCC Grammar File*, Januar 1998.

[JNDI97] JavaSoft, *JNDI: Java Naming and Directory Interface*, 1997.

[JSPI97] JavaSoft, *JNDI SPI: Java Naming and Directory Service Provider Interface*, 1997.

[Kemp98] Bernhard Kempter, „Entwurf eines Java/CORBA-basierten Mobilen Agenten“, Diplomarbeit, Technische Universität München, August 1998.

[Kern95] Helmut Kerner, *Rechnernetze nach OSI*, Addison Wesley, 3. Auflage, 1995.

[Klem95] Anders Klemets, „IP Mobility Support version 11“, http://it.kth.se/pub/klemets, Juli 1995.

[Koch96] Thomas Koch, *Automated Management of Distributed Systems*, Dissertation, FernUniversität Hagen, Oktober 1996.

[KoKr96] Thomas Koch und Bernd Krämer, „Rules and agents for automated management of distributed systems“, *Distributed System Engineering*, (3):110-114, 1996.

[Krel95] Christoph Krell, „Policy-basiertes Management von verteilten Systemen“, Diplomarbeit, Fernuniversität Hagen, 1995.

[Krieg94] Prof. Dr. H.-P. Kriegel, „Informatik I“, Vorlesungsskript, Ludwig-Maximilians-Universität München, Wintersemester 1994.

[LePe96] Laura Lemay und Charles Perkins, *Teach Yourself JAVA in 21 Days*, Sams.net Publishing, 1996.

[LSY97] E. Lupu, M. Sloman und N. Yialelis, „Policy Based Roles for Distributed Systems Security“, In *HP-Openview University Association Workshop*, S. 1-12, Madrid, 1997.

[LuSl96] E. Lupu und M. Sloman, „Towards a Role-based Framework for Distributed Systems Management“, *Journal of Network and Systems Management*, 1996.

[LuSl97a] E. Lupu und M. Sloman, „Conflict Analysis for Management Policies“, Technischer Bericht, Imperial College, Department of Computing, 1997.

[LuSl97b] Emil Lupu und Morris Sloman, „A Policy Bases Role Object Model“, In *First International Enterprise Distributed Object Computing Workshop (EDOC'97)*, S. 1-12, Gold Coast, Queensland, Australia, 1997, IEEE CS Press.

[Mari97] Damian A. Marriott, *Policy Service for Distributed Systems*, PhD thesis, Imperial College London, 1997.

[May98] Hussein Mayhu, „Erstellung eines Testprogramms für das DHCP Server Management“, Fortgeschrittenenpraktikum, Ludwig-Maximilians-Universität München, Juli 1998.

[McCl96] Stuart McClure, „Product Comparison: Dynamic IP addressing software“, *InfoWorld Electric*, 18(43), October 1996.

[Meta98] MetaInfo, „MetaIP/Manager: Einfaches Management von DNS und DHCP Daten“, http://www.metainfo.de, Mai 1998.

[Mich96] LDAP development Team, „The SLAPD and SLURPD Administrator's Guide“, 1996, Release 3.3.

[MMS94] D. Marriott, M. Mansouri-Samani und M. Sloman, „Specification of Management Policies“, In *Fifth IFIP/IEEE International Workshop on Distributed Systems: Operations & Management*, Toulouse (France), 1994, IFIP/IEEE.

[MoSl94] M.S. Sloman J.D. Moffett, „Specification of Management Policies and Discretionary Access Control“, In *Network and Distributed Systems Management*, Kapitel 17, S. 455-479, Addison-Wesley, 1994.

[OHE 96] Robert Orfali, Dan Harkey und Jeri Edwards, *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, Inc., 1996.

[OMG97a] OMG, „A Discussion of the Object Management Architecture“, Januar 1997.

[OMG97b] OMG, „CORBAservices: Common Object Services Specification“, November 1997.

[OMG97c] GMD FOKUS, „Mobile Agent System Interoperability Facilities Specification“, Technischer Bericht, Object Management Group, 1997.

[Orch98] Orchestream, „The Orchestream Solution“, http://www.orchestream.com/solution.html, 1998.

[PiFu97] Evaggelia Pitoura und Ioannis Fudos, „Tracking Mobile Users Using Hierarchical Location Databases“, Technischer Bericht, Department of Computer Science, University of Ioannina, 1997.

[PTHV96] Gurdeep Singh Pall, Jeff Taarud, Kory Hamzeh, William Verthein und W. Andrew Little, „Point-to-Point Tunneling Protocol (PPTP) Technical Specification“, http://www-ms.eunet.re/workshop/prog/prog-gen/pptp.htm, February 1996.

[RFC951] W. J. Croft und J. Gilmore, „RFC 951: Bootstrap Protocol“, Technischer Bericht, IETF, September 1985.

[RFC1533] S. Alexander und R. Droms, „RFC 1533: DHCP Options and BOOTP Vendor Extensions“, Technischer Bericht, IETF, Oktober 1993.

[RFC1534] R. Droms, „RFC 1534: Interoperation Between DHCP and BOOTP“, Technischer Bericht, IETF, Oktober 1993.

[RFC1541] R. Droms, „RFC 1541: Dynamic Host Configuration Protocol“, Technischer Bericht, IETF, Oktober 1993.

[RFC1548] W. Simpson, „RFC 1548: The Point-to-Point Protocol (PPP)“, Technischer Bericht, IETF, Dezember 1993.

[RFC1558] T. Howes, „RFC 1558: A String Representation of LDAP Search Filters“, Technischer Bericht, IETF, Dezember 1993.

[RFC1777] W. Yeong, T. Howes und S. Kille, „RFC 1777: Lightweight Directory Access Protocol“, Technischer Bericht, IETF, März 1995.

[RFC1778] T. Howes, S. Kille, W. Yeong und C. Robbins, „RFC 1778: The String Representation of Standard Attribute Syntaxes“, Technischer Bericht, IETF, März 1995.

[RFC1779] S. Kille, „RFC 1779: A String Representation of Distinguished Names“, Technischer Bericht, IETF, März 1995.

[RFC1823] T. Howes und M. Smith, „RFC 1823: The LDAP Application Program Interface“, Technischer Bericht, IETF, August 1995.

[RFC1959] T. Howes und M. Smith, „RFC 1959: An LDAP URL Format“, Technischer Bericht, IETF, Juni 1996.

[RFC2002] C. Perkins, „RFC 2002: IP Mobility Support“, Technischer Bericht, IETF, Oktober 1996.

[RFC2131] R. Droms, „RFC 2131: Dynamic Host Configuration Protocol“, Technischer Bericht, IETF, März 1997.

[RFC2138] C. Rigney, A. Rubens, W. Simpson und S. Willens, „RFC 2138: Remote Authentication Dial In User Service (RADIUS)“, Technischer Bericht, IETF, April 1997.

[RFC2251] M. Wahl, T. Howes und S. Kille, „RFC 2251: Lightweight Directory Access Protocol (v3)“, Technischer Bericht, IETF, Dezember 1997.

[RFC2252] M. Wahl, A. Coulbeck, T. Howes und S. Kille, „RFC 2252: Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions“, Technischer Bericht, IETF, Dezember 1997.

[RFC2253] M. Wahl, S. Kille und T. Howes, „RFC 2253: Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names“, Technischer Bericht, IETF, Dezember 1997.

[RFC2254] T. Howes, „RFC 2254: The String Representation of LDAP Search Filters“, Technischer Bericht, IETF, Dezember 1997.

[RFC2255] T. Howes und M. Smith, „RFC 2255: The LDAP URL Format“, Technischer Bericht, IETF, Dezember 1997.

[RFC2256] M. Wahl, „RFC 2256: A Summary of the X.500(96) User Schema for use with LDAPv3“, Technischer Bericht, IETF, Dezember 1997.

[Rieg96] Gernot Rieger, „Erstellung einer Managementschnittstelle für DHCP-Server“, Diplomarbeit, Technische Universität München, 1996.

[Roel98] Harald Rölle, „Implementierung des CORBA Topology Service“, Fortgeschrittenenpraktikum, Technische Universität München, 1998.

[Schm97] Douglas Schmidt, „An Overview of OMG CORBA Event Services“, Vortragsfoliensammlung, 1997.

[Scho92] Uwe Schöning, Logik für Informatiker, BI-Wissenschaftsverlag, Mannheim, 3. Auflage, 1992.

[Sun98] Inc. Sun Microsystems, „The Source for Java Technology", http://www.javasoft.com, 1998.

[Telx95] Telxon, „First Integrated MobileIP Wireless Solution", http://www.telxon.com/tech1.htm, September 1995.

[Visi97] Visigenic, *Visibroker for Java Reference Manual Version 3.0*, September 1997.

[Visi97a] Visigenic, *Visibroker for Java Programmer's Guide Version 3.0*, September 1997.

[Visi97b] Visigenic, *Visibroker Naming and Event Services Programmer's Guide Version 3.0*, September 1997.

[Wies94a] R. Wies, „Policies in Network and Systems Management -- Formal Definition and Architecture", *Journal of Network and Systems Management*, 2(1):63 - 83, 1994, M. Malek (edt.), Plenum Publishing Corp., New York.

[Wies94b] R. Wies, „Policy Definition and Classification: Aspects, Criteria, and Examples", In *Proceedings of the IFIP/IEEE International Workshop on Distributed Systems: Operations & Management*, Oktober 1994.

[Wies95] R. Wies, „Using a Classification of Management Policies for Policy Specification and Policy Transformation", In Y. Raynaud und A. Sethi (Hrsg.), *Proceedings of the 4th International IFIP/IEEE Symposium on Integrated Network Management*, Mai 1995.

[Wies95b] Rene Wies, *Policies in Integrated Network and Systems Management: Methodologies for the Definition, Transformation and Application of Management Policies*, Dissertation, Ludwig-Maximilians-Universität München, 1995.

[X.500] ITU-T Study Group 7, „The Directory - Overview of Concepts, Models and Services", ITU-T Recommendation X.500, ITU-T, 1993.