INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

CORBA-basiertes Management von UNIX-Workstations mit Hilfe von ODP-Konzepten

Bearbeiter:	Tobias Müller
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Alexander Keller Dr. Bernhard Neumair

INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

CORBA-basiertes Management von UNIX-Workstations mit Hilfe von ODP-Konzepten

Bearbeiter:	Tobias Müller
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Alexander Keller Dr. Bernhard Neumair

Abgabetermin: 15. Februar 1998

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

München, den 15. Februar 1998

....

(Unterschrift des Kandidaten)

Zusammenfassung

Heutige DV-Systemlandschaften bestehen meist aus gewachsenen Netzen von Workstations und Personal Computers. Die oft heterogenen Netzkomponenten und Endsysteme bilden ein verteiltes System, in denen einzelne Server bestimmte Dienste erbringen. Software-Komponenten kooperieren nach dem Client/Server-Prinzip und realisieren somit verteilte Anwendungen. Das Management eines solchen heterogenen Systems ist sehr komplex. Daher ist ein integriertes Management von Netzkomponenten, Endsystemen und Anwendungen erforderlich. Hierzu wird ein einheitliches Managementmodell benötigt, das mächtig genug ist, die Managementinformation und -funktionalität komplexer Ressourcen zu beschreiben.

In dieser Diplomarbeit wird ein Modell für das Management von UNIX-Workstations und Systemdiensten entwickelt. Es basiert auf der *Object Modeling Technique* (OMT), die einen objektorientierten Modellierungsansatz bietet, der unabhängig von einer bestimmten Managementarchitektur ist. Damit das Modell auf unterschiedlichste Ressourcen abgebildet werden kann, werden generische Objektklassen gesucht, die allgemeingültige Managementinformation bereitstellen. Hierzu wird das standardisierte *Reference Model of Open Distributed Processing* der ISO betrachtet, welches abstrakte, systemunabhängige Sichten zum Entwurf von verteilten Anwendungen spezifiziert. Die Analyse der Konzepte des *Computational* und *Engineering Viewpoint* unter Managementgesichtspunkten führt zu generischen Objektklassen. Diese werden gemäß den Anforderungen verschiedener Managementfunktionsbereiche mit Information (Attributen) und Funktionalität (Methoden) ausgestattet. Die Basisklassen werden im nächsten Schritt für das Management von UNIX-Systemen verfeinert. Nach Einführung von generischen Klassen für das Dienstmanagement werden die konkreten Systemdienste NFS und NIS modelliert. Das entstandene Modell erlaubt ein Management auf diversen Ebenen. Auf der abstrakten Ebene der Basisklassen kann eine Anwendung viele unterschiedliche Ressourcen überwachen. Auf der untersten Ebene der verfeinerten Klassen kann ein spezielles Werkzeug für eine bestimmte Ressource arbeiten.

Da die allgemeine Modellierungstechnik OMT eingesetzt wurde, läßt sich das Modell auf die Informationsmodelle verschiedener Managementarchitekturen abbilden. In dieser Arbeit wird das Modell auf Basis von CORBA prototypisch implementiert. Ein CASE-Tool erzeugt aus dem Objektmodell IDL-Beschreibungen der Managementschnittstellen für einen CORBA-Agenten. Der Agent wird in Java mit Hilfe der CORBA-Entwicklungs- und Laufzeitumgebung *VisiBroker for Java* realisiert. Zusätzlich wird ein Management-Applet entwickelt, welches den Zugriff auf den Agenten (Abruf von Information, Steuerung von Ressourcen) über eine graphische Oberfläche ermöglicht.

Die Arbeit zeigt, daß CORBA für das integrierte System- und Anwendungsmanagement geeignet ist. Die generischen Basisklassen des zugrundeliegenden Managementmodells können für viele unterschiedliche Endsysteme und Anwendungen eingesetzt werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Zwei der wichtigsten Schlagworte der IT-Branche innerhalb des letzten Jahrzehnts waren «*Downsizing*» und «*Client/Server*». Diese drücken den Wandel aus, der sich in der DV-Systemlandschaft vollzogen hat. Monolithische Anwendungen auf zentralen Großrechnern wurden durch verteilte Anwendungen ersetzt, deren Software-Komponenten nach dem Client/Server-Prinzip miteinander kooperieren. Ermöglicht wurde dieser Trend durch immer leistungsfähigere Workstations und zunehmende Vernetzung auf Basis standardisierter Kommunikationsprotokolle. Auslöser war aber das bessere Preis-/Leistungsverhältnis der verteilten Systeme gegenüber der klassischen Mainframe-Technologie. Ein Paradebeispiel ist der Erfolg der Standard-Software SAP R/3 für Unternehmen, der Nachfolger der Großrechneranwendung R/2.

Wie so oft wurde aber in der Kostenrechnung ein Faktor unterschätzt, der mit zunehmender Größe der verteilten Systeme einen ebenso immer größeren Einfluß erhält: die Komplexität des technischen Managements. Diese resultiert aus der großen Zahl beteiligter Hardware- und Software-Komponenten, deren räumlicher Verteilung und Abhängigkeiten untereinander und vor allem aus der Heterogenität der Endsysteme und Netzkomponenten. Die Kosten hierfür drohen inzwischen die erhofften Einsparungen wieder aufzufressen.

Es liegt auf der Hand, daß nur ein integriertes Management, d.h. ein einheitliches Management von Netzkomponenten, Endsystemen und Anwendungen innerhalb einer standardisierten Architektur, die Problematik lösen kann. Beim System- und Anwendungsmanagement werden aber die gleichen Fehler wie beim Netzmanagement wiederholt. Dort wurden zwar Protokoll- und Dienstschnittstellen genormt, um die Interoperabilität der Komponenten sicherzustellen, standardisierte Managementschnittstellen wurden aber beim Entwurf meist vergessen. Zwar liefern die großen Hersteller inzwischen Werkzeuge für das Management ihrer Endsysteme mit oder bieten entsprechende Produkte an. Damit kann aber keine einheitliche Gesamtlösung für eine individuelle heterogene Umgebung zusammengesetzt werden, wie sie das integrierte Management fordert. Noch schlechter sieht es im Bereich des Anwendungsmanagements aus.

Beim Netzmanagement konnten sich im wesentlichen zwei verschiedene Managementarchitekturen etablieren. Auf der einen Seite das mächtige OSI-Management, welches sich aber aufgrund seiner Komplexität nur für sehr große Telekommunikationsnetze durchsetzen konnte. Auf der anderen Seite das in der Funktionalität eher eingeschränkte Internet-Management der IAB/IETF, welches aber gerade wegen seiner Einfachheit aus Kostengründen im LAN-Bereich sehr weite Verbreitung fand. Beim System- und Anwendungsmanagement präsentiert sich die Ausgangssituation nicht so einfach. Hier sind neben den zwei bereits erwähnten weitere Architekturen wie das *Distributed Management Environment* der OSF, die *Object Management Architecture* mit CORBA der OMG, das *Desktop Management Interface* der DMTF und das *Web Based Enterprise Management* vorhanden oder im Entstehen.

Ein für die Belange des integrierten Managements aus mehreren Gründen vielversprechender Ansatz, der in dieser Arbeit weiter verfolgt wird, scheint CORBA zu sein. Dieser Meinung sind auch die Autoren von [OHE96]: „*We believe CORBA is the answer to the distributed systems management nightmare.*“

1.2 Aufgabenstellung

Ein wichtiger Teil einer Managementarchitektur ist das Informationsmodell. Dieses schafft eine einheitliche Basis zur Beschreibung der Managementinformation für Ressourcen und legt Operationen zum Zugriff auf die Information fest. Da CORBA keine reine Managementarchitektur ist, sondern nur eine standardisierte Kommunikationsinfrastruktur für verteilte Objekte bereitstellt, besitzt es kein eigenes Informationsmodell. Es definiert allerdings ein Objektmodell. Mit der zugehörigen Beschreibungssprache IDL werden die Schnittstellen der Objekte, also die an der Oberfläche sichtbaren Attribute und Operationen, festgelegt.

Für integriertes Management wird ein mächtiges Informationsmodell benötigt, das in der Lage ist, die Komplexität der zu managenden Ressourcen angemessen zu modellieren. Diese Mächtigkeit bietet der objektorientierte Ansatz, bei dem Objektklassen als Abstraktionen der realen Ressourcen für das Management dienen. Attribute der Klasse definieren die Managementinformation, Methoden erlauben den Zugriff auf die Information und das Ausführen von Aktionen auf der Ressource. Die direkte Benutzung von IDL als Modellierungssprache für das Managementmodell würde aber in eine Sackgasse bzw. wiederum zu einer isolierten Lösung führen. Da CORBA, wie oben beschrieben, nur *einen* möglichen Ansatz für das System- und Anwendungsmanagement darstellt, der außerdem noch nicht weit verbreitet ist, sollte sich das Modell zur Schaffung von Übergängen zwischen verschiedenen Architekturen auf die unterschiedlichen Informationsmodelle abbilden lassen. Ein generisches Modell ermöglicht nicht nur die Integration von Altsystemen, sondern gewährleistet auch die Zukunftssicherheit. Hierzu gehört auch die Anwendbarkeit des Modells auf unterschiedlichste Ressourcen in heterogener Umgebung. Es müssen daher Basisklassen mit generischer Managementinformation und -funktionalität für das Objektmodell gefunden werden, die einerseits die Heterogenität der Ressourcen verschatten, aber trotzdem ein effizientes Management erlauben. Die generischen Basisklassen können für spezielle Ressourcen anschließend verfeinert werden. Das standardisierte *Reference Model for Open Distributed Processing* (RM-ODP) der ISO definiert unterschiedliche abstrakte Sichten auf ein verteiltes System und bildet somit ein Rahmenwerk zum Entwurf von verteilten, objektorientierten Anwendungen unabhängig von der darunterliegenden Systemplattform. Die Konzepte des *Computational* und *Engineering Viewpoint* sollen die Grundlage für die Definition der generischen Basisklassen des Objektmodells bilden. CORBA wiederum gilt als wichtigste Realisierung des RM-ODP.

Aufgabe dieser Arbeit ist also die Erstellung eines Objektmodells für das Systemmanagement von UNIX-Workstations über ein Top-Down-Vorgehen, welches den formulierten Anforderungen genügt. Zusätzlich soll auch das eher zum Anwendungsmanagement zählende Management von Systemdiensten in das Modell mit einbezogen werden. Zur Modellierung soll die verbreitete *Object Modeling Technique* (OMT) eingesetzt werden. Mit Hilfe eines CASE-Tools sollen anschließend aus dem Objektmodell IDL-Beschreibungen der Managementschnittstellen für einen CORBA-Agenten erzeugt werden. Die Tragfähigkeit des Modells soll durch die Implementierung eines Prototypen auf Basis eines CORBA-konformen *Object Request Broker* erfolgen. Der Prototyp soll sowohl einen CORBA-Agenten als auch eine Managementanwendung mit graphischer Oberfläche zum Zugriff auf die vom Agenten bereitgestellte Managementinformation beinhalten. Bei der Implementierung soll außerdem die Wiederverwendbarkeit von Code eines bestehenden SNMP-Agenten überprüft werden.

1.3 Gliederung und wichtigste Ergebnisse

Diese Einleitung ist das erste von sieben Kapiteln der Arbeit. Im folgenden Kapitel 2 wird nach einer Abgrenzung des System- und Anwendungsmanagements und einer Analyse der Anforderungen an ein Managementmodell das weitere Vorgehen zur Gewinnung des Objektmodells und zur Implementierung des Prototypens beschrieben. Kapitel 3 beleuchtet das Umfeld der Arbeit. Hierzu bietet es kurze Einführungen in Client/Server-basierte Systemdienste im allgemeinen und in die verteilten Dienste NFS und NIS im speziellen, die in Kapitel 5 genauer betrachtet werden. Anschließend wird ein Überblick über die *Object Modeling Technique* gegeben. Es folgt eine Einführung in das *Reference Model of Open Distributed Processing* (RM-ODP) mit näherer Betrachtung der für diese Arbeit relevanten Konzepte des *Computational* und *Engineering Viewpoint*. Hierauf wird ein Blick auf die wichtigsten Komponenten der CORBA-Architektur geworfen. Das Kapitel schließt mit der Vorstellung des State-of-the-Art des Systemmanagements. Dabei werden Schwächen bestehender Managementlösungen aufgedeckt, die mit dem in dieser Arbeit verfolgten Ansatz vermieden werden sollen. An diesem Punkt liegen mit den Anforderungen an das Modell, der Vorgehensweise zur Realisierung und den Grundlagen der verwendeten Techniken und Architekturen alle Voraussetzungen vor, um zum Kern der Arbeit, nämlich zur Erstellung des Modells überzugehen.

Im zentralen Kapitel 4 wird anhand eines Top-Down-Vorgehens das Objektmodell für das Management von UNIX-Systemen spezifiziert. Hierzu werden der *Computational* und *Engineering Viewpoint* des RM-ODP analysiert und aus deren Konzepten generische Managementobjektklassen abgeleitet. Diesen wird gemäß den Anforderungen von Funktionsbereichen möglichst allgemeingültige Managementinformation und -funktionalität in Form von Attributen und Methoden hinzugefügt. Damit liegt ein Managementmodell vor, das wichtige Bereiche des System- und Anwendungsmanagements abdeckt und aufgrund seiner Allgemeingültigkeit auf sehr viele unterschiedliche Systeme angewendet werden kann. Die generischen Basisklassen werden im nächsten Schritt für das Management von UNIX-Systemen verfeinert. Hierzu wird ein bestehender Prototyp eines Modells, der auf einer SNMP-MIB für das Management von Endsystemressourcen beruht, in das Modell integriert. Die erfolgreiche Integration rechtfertigt die Spezifikation der Basisklassen. Weiterhin werden generische Klassen für das Management von Client/Server-basierten Systemdiensten eingeführt. Die Tragfähigkeit des entstandenen Modells wird schließlich durch einen Vergleich mit existierenden SNMP-MIBs gezeigt.

Kapitel 5 erweitert das Objektmodell um Klassen für das Management der konkreten Systemdienste NFS und NIS. Dazu werden die generischen Klassen für das Dienstmanagement verfeinert. Bottom-up wird überprüft, ob die geforderte Managementinformation von den betrachteten Diensten zur Verfügung gestellt wird, was leider nicht vollständig der Fall ist. Am Beispiel von NFS wird außerdem demonstriert, wie sich das dynamische Modell von OMT zur Modellierung asynchroner Ereignismeldungen verwenden läßt. Insgesamt kann der Schluß gezogen werden, daß die Modellierungstechnik OMT und damit auch der Nachfolger UML mit Einschränkungen als architekturunabhängiges Informationsmodell eingesetzt werden kann.

Kapitel 6 beschreibt die prototypische Implementierung des entwickelten Modells für eine IBM-Workstation. Das OMT-Modell wird mittels des CASE-Tools StP in IDL-Objektbeschreibungen für einen CORBA-Agenten überführt. Dieser Agent wird in der CORBA-Entwicklungs- und Laufzeitumgebung *VisiBroker* für die Programmiersprache Java implementiert. Es wird gezeigt, wie der CORBA-Agent Code eines bestehenden, in C programmierten SNMP-Agenten wiederverwenden kann. Außerdem wird ein Management-Applet entwickelt, welches aus jedem Web-Browser heraus über eine graphische Oberfläche den Zugriff auf die Managementinformation des Agenten ermöglicht. Als Ergebnis liegt ein Prototyp einer CORBA-basierten Lösung für das Management von Endsystemressourcen und Systemdiensten vor.

Das abschließende Kapitel 7 faßt die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf noch offene Fragestellungen.

Kapitel 2

Vorgehensmodell

2.1 Präzisierte Aufgabenstellung

Ziel dieser Arbeit ist es, ein Objektmodell für das Management von UNIX-Systemen und Systemdiensten zu entwickeln. Bei der Modellierung soll ein Top-Down-Vorgehen benutzt werden. Dies bedeutet, daß die Definition von Managementinformation nicht darauf basiert, welche Information von einer Ressource zur Verfügung gestellt wird, sondern welche Information benötigt wird, um Managementanwendungen entwickeln zu können, die den Anforderungen der Betreiber eines Systems gerecht werden. Außerdem soll das Modell für das integrierte Management geeignet sein. Das heißt, das Modell soll das Management beliebiger Ressourcen in einer heterogenen, verteilten Systemlandschaft ermöglichen. Die Tragfähigkeit des Modells soll durch die Implementierung eines Prototypen gezeigt werden.

Diese wenigen Sätze werfen eine Reihe von Fragen auf. Was versteht man unter Management? Welche Aufgaben zählen zum Systemmanagement? Wozu wird ein Modell benötigt und was wird modelliert? Welche Anforderungen werden durch die Ziele des integrierten Managements an das Modell gestellt? Was zeichnet die gewählte Modellierungstechnik aus? Wie soll der Prototyp realisiert werden und welche Werkzeuge werden für die Implementierung genutzt?

Diese Fragen werden in den folgenden Abschnitten beantwortet. Nachdem geklärt ist, was modelliert werden soll und was das Modell zu leisten hat, wird erläutert, welcher Ansatz in dieser Arbeit zur Erreichung des Ziels verfolgt wird. Das Vorgehensmodell erläutert die Schritte vom Beginn der Modellierung bis zur Implementierung des Prototypen.

2.1.1 Funktionsbereiche des Managements

Im letzten Jahrzehnt gab es eine große Veränderung in der Systemlandschaft der Datenverarbeitung. Durch immer leistungsfähigere Workstations bei ständig fallenden Preisen und gleichzeitiger Etablierung neuer Netztechniken mit großer Kommunikationsbandbreite verlagerte sich die Datenverarbeitung von zentralen Großrechnern auf eine Systemumgebung bestehend aus verteilten, heterogenen und autonomen Komponenten. In dieser Umgebung setzen sich Anwendungen aus Modulen zur Datenhaltung, -verarbeitung und Präsentation zusammen, die über das Client/Server-Prinzip miteinander kooperieren. Natürlich ist ein Management in dieser verteilten Umgebung im Vergleich zum Großrechner komplexer.

Allgemein wird das Management in die folgenden Funktionsbereiche eingeteilt (vgl. [HA93], [HNW95]).

Konfigurationsmanagement: Ziel ist es, durch Beeinflußung von Parametern eine Ressource so einzurichten, daß sie in Kooperation mit anderen im Normalbetrieb den gewünschten Dienst erbringt.

Fehlermanagement: Seine Aufgabe ist das Aufspüren und Beheben von Fehlern. Dazu gehört auch die Verarbeitung von Fehlermeldungen.

Leistungsmanagement: Sammeln und Auswerten von Performance-Daten, um die Gesamtleistung eines Systems durch Optimierung seiner Ressourcen zu verbessern.

Abrechnungsmanagement: Protokollierung der Nutzung von Betriebsmitteln, um diese mit Benutzern abrechnen zu können.

Sicherheitsmanagement: Teil der Benutzerverwaltung, der die Zugriffsrechte verwaltet (Autorisierung). Zweite wichtige Aufgabe ist die Zugangskontrolle (Authentisierung).

2.1.2 Abgrenzung des Systemmanagements

Als nächstes soll betrachtet werden, welche Aufgaben das Systemmanagement umfaßt. Bevor sich Rechnernetze durchgesetzt haben, bedeutete Systemmanagement die Verwaltung der Ressourcen einzelner, voneinander getrennter Rechner. Hierzu zählen die folgenden Aufgabenbereiche eines Systemadministrators:

- Verwaltung der Speicherressourcen (Hauptspeicher, Hintergrundspeicher, Plattenplatz)
- Datenverwaltung (Filesysteme, Backups, Archivierung)
- Verwaltung von Ein-/Ausgabegeräten (Spooling, Drucker etc.)
- Benutzerverwaltung und Zugangskontrolle
- Installation und Konfiguration von Hardware-Komponenten
- Installation und Konfiguration von Betriebssystem und Anwendungen
- Überwachung der laufenden Prozesse
- Operating: Fehler- und Leistungsmanagement

Anschließend begann das Zeitalter der Vernetzung. Immer mehr einst isolierte Rechner wurden miteinander verbunden. Ziel war die gemeinsame Nutzung von Ressourcen (Dateisysteme, Ausgabegeräte, etc.) im Netz. Zusätzlich wurden die ersten verteilten Anwendungen wie Electronic Mail und Verzeichnisdienste entwickelt. Die Aufgaben des Systemmanagements wurden natürlich aufgrund der neuen Bereiche vielfältiger. Trotzdem konnte das Systemmanagement noch deutlich vom Netz- und Anwendungsmanagement abgegrenzt werden. Dies wird anhand der Abbildung 2.1 (links) erläutert.

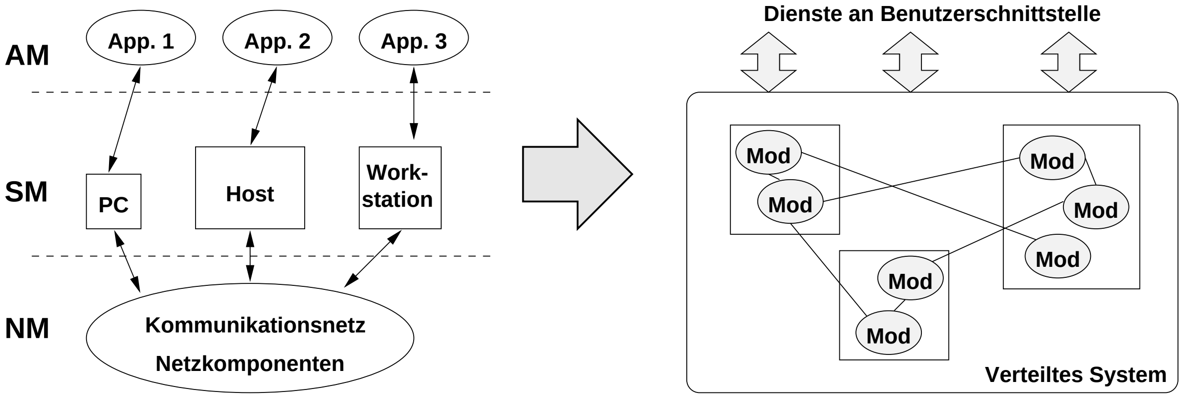


Abbildung 2.1: Wandel des Systemmanagements

Die Basis ist das Netzmanagement (NM). Dieses soll durch Überwachung und Steuerung der Netzkomponenten sicherstellen, daß eine Kommunikation der Endsysteme möglich ist. Das Systemmanagement baut in vernetzten Umgebungen auf dem Netzmanagement auf bzw. setzt dieses voraus. Das Augenmerk liegt auf der Verwaltung der Endsysteme und deren Bestandteile, erweitert um Aspekte wie gemeinsame Nutzung von Ressourcen und globale Benutzerverwaltung. Noch eine Ebene höher ist das Anwendungsmanagement (AM) angesiedelt. Hier geht es um die Kontrolle und Steuerung von (verteilten) Anwendungen.

Der Trend geht aber weiter zu offenen verteilten Systemen (*Open Distributed Processing*). Durch Schaffung von Transparenzen in verteilten Umgebungen verliert der Benutzer die Sicht auf die einzelnen Endsysteme, auf denen die Anwendungen laufen. Mittlerweile sind verteilte Systeme Realität, die einem Benutzer Dienste anbieten, ohne daß dieser feststellen kann, welche Komponenten des Systems die Rechen-, Speicher- und Kommunikationsressourcen zur Erbringung der Dienste bereitstellen. Zur Schaffung dieser Transparenzen wurden Systemdienste wie verteilte Dateisysteme (z.B. NFS, DFS), Verzeichnisse (z.B. NIS, DNS) und Informationssysteme (z.B. WWW) geschaffen. Hierauf können verteilte Benutzeranwendungen aufbauen. Der rechte Teil von Abbildung 2.1 zeigt ein verteiltes System, welches an der Benutzerschnittstelle Dienste zur Verfügung stellt. Diese Dienste sind durch kooperierende Software-Module auf vernetzten Endsystemen realisiert und stellen somit selbst verteilte Anwendungen dar. Aus der Abbildung läßt sich auch erkennen, welchen Stellenwert das Systemmanagement in Zukunft einnehmen wird. In einem verteilten System ist die Abgrenzung von Netz-, System- und Anwendungsmanagement nicht mehr möglich. Zu den „klassischen“ Aufgabenfeldern kommen neue wichtige Anforderungen hinzu, die sich aus der Administration verteilter Systemdienste, der Software-Verteilung, der Lastbalanzierung und aus der globalen Benutzerverwaltung (Autorisierung, Authentisierung und Abrechnung) ergeben. Die folgenden Abschnitte geben einen Überblick über die Szenarien Software- und Dienstmanagement.

2.1.3 Aufgaben des Software-Managements

Da die Installation und Konfiguration der Software ein wichtiger Aspekt des Systemmanagements ist, werden hier die Aufgaben des Software-Managements kurz vorgestellt:

Software-Verteilung und Installation: Unter Software-Verteilung versteht man die koordinierte und automatisierte Installation von Software-Komponenten auf mehreren Rechnern einer verteilten Umgebung. Bei Einrichtung eines neuen Dienstes werden neue SW-Komponenten installiert, bei einem Upgrade nur eine neue Version einer bereits vorhandenen Komponente. Berücksichtigt werden müssen Abhängigkeiten zwischen den Versionen verschiedener SW-Komponenten und Anforderungen von SW-Komponenten an die Hardware des Systems.

Überwachung von bestehenden Installationen: Durch Eingriffe von Administratoren und Benutzern oder durch Hardware-Defekte wie fehlerhafte Sektoren von Festplatten können bestehende SW-Installationen Schaden nehmen, so daß die Erbringung des Dienstes durch die SW-Komponente nicht mehr gewährleistet ist. Diese Fehler sollen ggf. automatisch entdeckt werden.

Konfigurationsänderungen: Diese treten z.B. bei Verlagerung eines Dienstes auf einen anderen Rechner auf. Dies zieht Modifikationen bei allen betroffenen Systemen nach sich.

Kontrolle der Startprozedur: Beim Start von SW-Komponenten sind Abhängigkeiten in der Startreihenfolge und korrekte Übergabe von Parametern wichtig.

Lizenzverwaltung: Diese Aufgabe schließt die Überwachung von Lizenzvereinbarungen für SW-Komponenten und die Einrichtung sog. Lizenzserver ein.

Die Installation von SW-Komponenten für Systemdienste und Applikationen sollte möglichst einheitlich und unabhängig vom Hersteller der Komponente sowie der Hardware-Architektur und des Betriebssystems sein, damit die skizzierten Aufgaben des Software-Managements auch in einer heterogenen verteilten Umgebung von einer Management-Applikation erfüllt werden können. Einen guten Ansatz bieten hier die in der UNIX-Welt verbreiteten Packages für SW-Komponenten, obwohl die verwendeten Paketformate unter verschiedenen UNIX-Derivaten wie z.B. AIX und HP-UX leider (noch) nicht übereinstimmen. Dementsprechend schlecht geeignet für das SW-Management sind Installationsmethoden wie proprietäre Setup-Programme und Installationsskripte, die meist in der Windows-Welt anzutreffen sind.

Das *Software Working Commitee* der *Desktop Management Task Force* (DMTF) setzt sich aus Mitarbeitern großer Software-Hersteller zusammen. Es hat sich zur Aufgabe gemacht, die für das SW-Management benötigten Informationen über SW-Komponenten zu standardisieren. Diese auf Anforderungen von Administratoren und Endbenutzern basierenden Informationen sollen in erster Linie SW-Management-Applikationen unterstützen. Die *Software Standard Groups Definition* [DMT95] teilt die Managementinformation in mehrere Gruppen von Attributen ein. Unterschieden werden statische Daten, die vom SW-Hersteller bei der Entwicklung festgelegt werden, und dynamische Daten, deren Werte sich erst bei der Installation oder sogar danach ergeben. Beispiele für statische Daten sind Produktname und Version der SW-Komponente. Diese sind wichtig, um Abhängigkeiten bei der Installation identifizieren zu können. Dynamisch sind beispielsweise der Installationsort und der Status der Installation. Solche Informationen sind für die Überwachung von SW-Installationen nötig. Die Ansätze der DMTF sollen bei der Modellierung der Objektklassen berücksichtigt werden.

2.1.4 Anforderungen an das Dienstmanagement

Wie oben bereits beschrieben wurde, spielen verteilte Systemdienste eine wichtige Rolle bei den vernetzten Systemumgebungen heutiger DV-Versorgungsstrukturen. Die Administration dieser Dienste fällt sowohl unter das System- als auch unter das Anwendungsmanagement. An dieser Stelle sollen die Anforderungen an das Dienstmanagement aufgezeigt werden, um später eine Top-Down-Modellierung entsprechender Objektklassen zu erlauben. Hierbei hat sich die Vorgehensweise von Markus Gutschmidt bewährt, die in seiner Arbeit „Ein Objektmodell für ein integriertes Management von Systemdiensten mit Client/Server-Struktur" [Gut95] beschrieben ist. Er analysiert die Betriebsaspekte *Steuerung* und *Überwachung* für beliebige Dienste anhand einiger Funktionsbereiche, die in der folgenden Aufzählung kurz erklärt werden.

Leistungsmanagement: Aufgabe des Leistungsmanagements ist es, nicht nur den Betrieb einer Ressource zu gewährleisten, sondern ein möglichst „gut" laufendes Gesamtsystem zu erzielen. Bezogen auf einen Dienst bedeutet dies vor allem, daß der Server alle an ihn gestellten Aufträge mit einer zufriedenstellenden Antwortzeit bearbeiten kann. Es sollten aber auch bei kurzzeitigen Spitzenbelastungen keine Aufträge wegen eines Überlaufs von Warteschlangen, etc. verlorengehen.

Information des Leistungsmanagements sind demnach Meßgrößen für die Dienstgüte (QoS) wie z.B. Durchsatz, Auslastung und Antwortzeit. Diese Größen müssen zur Laufzeit für einen ordentlichen Betrieb des Dienstes überwacht werden, aber auch für eine langfristige Planung über einen längeren Zeitraum statistisch ausgewertet werden.

Fehlermanagement: (In [Gut95] nicht explizit untersucht.) Im Rahmen des Fehlermanagements müssen Fehler aufgespürt und diagnostiziert werden. Anschließend sollte natürlich eine Behebung des Fehlers angestrebt werden. Bei Diensten können Störungen durch fehlerhafte Aufträge oder durch interne Fehler bei der Auftragsbearbeitung auftreten.

Sicherheitsmanagement: (In [Gut95] nicht untersucht.) Das Sicherheitsmanagement umfaßt neben der Überwachung eines Dienstes im Hinblick auf Sicherheitsangriffe auch die Zugangskontrolle (Authentifizierung) zum Dienst und evtl. ein Verschlüsseln von schutzwürdigen Informationen und Daten. Ein wichtiger Punkt ist die Identifikation von Clients, die trotz fehlender Autorisierung zur Nutzung Aufträge an den Server stellen.

Statusmanagement: Die Überwachung des Status einer Ressource gehört zu den wichtigsten Aufgaben des Managements überhaupt. Bezogen auf einen Systemdienst geht es dabei im wesentlichen um die Frage, ob alle Prozesse laufen, die zur Erbringung eines Dienstes beitragen, d.h. einen Server bzw. einen Client realisieren. Neben der Ermittlung und Überwachung sieht das Management auch Funktionalität zum Ändern des Status vor. Dies ist beispielsweise notwendig, wenn eine Änderung an der Konfiguration des Dienstes das Stoppen und erneute Starten des Servers erfordert.

Auftragsmanagement: Einige Dienste lassen die Beobachtung und Steuerung einzelner Aufträge zu. Dies ist meist dann der Fall, wenn Aufträge sequentiell abgearbeitet werden und eine längere Bearbeitungszeit im Sekunden- oder Minutenzeitraum aufweisen. Da Aufträge Ressorcen im Server binden, kann bei Fehlern oder Leistungsengpässen ein Eingreifen durch das Management erforderlich sein. Dazu bedarf es natürlich auch an Beobachtungsmöglichkeiten für Einzelaufträge. Die Auswertung von Einzelaufträgen kann auch für eine Abrechnung der Dienstnutzung erforderlich sein.

Für Server, die sehr viele Aufträge mit sehr kurzer Bearbeitungszeit (Millisekundenbereich) bearbeiten, ist dagegen ein Management für Einzelaufträge wenig sinnvoll bzw. unmöglich. Hier sollten aber zumindest die Meßgrößen des Leistungsmanagements (siehe Kapitel 4.4.1) zur Verfügung gestellt werden.

Konfigurationsmanagement: Dieses hat die Aufgabe, einen Dienst so anzupassen und zu konfigurieren, daß er die gewünschte Leistung in vollem Umfang erfüllt. Neben dem Auslesen der Konfiguration muß auch die Änderung der Konfiguration durch das Management möglich sein. Die Konfiguration eines Dienstes kann meist über dienstspezifische Parameter beeinflußt werden. Für den Fall, daß der Dienst von mehreren Servern erbracht wird, gehört auch die Plazierung und Replikation von Servern zum Konfigurationsmanagement.

2.1.5 Managementmodelle

Damit eine Managementanwendung auf einer Systemressource operieren kann, wird ein für das Management relevantes Modell der Ressource benötigt, welches von den übrigen Eigenschaften abstrahiert. Im wesentlichen treten dabei folgende Interaktionen zwischen Managementanwendung und Ressource auf:

- Abfrage (*polling*) von Statusinformationen, Leistungsdaten, etc.
- Aktives Einwirken auf die Ressource: Setzen von Konfigurationsparametern, Veranlassen einer Zustandsänderung (z.B. *reset, stop*)
- Senden von asynchronen Ereignismeldungen (*notifications / events*) an den Manager

Das Informationsmodell stellt daher die Abstraktion einer Ressource für Managementzwecke dar und definiert die Managementinformation und -funktionalität dieser Ressource an ihrer Managementschnittstelle. In das Modell werden nur die für das Management relevanten Eigenschaften der Ressource aufgenommen. Damit nicht jede auf dem Markt vorhandene Ressource neu modelliert werden muß, wird versucht, möglichst allgemeingültige Information und Funktionalität zu finden. Dadurch lassen sich die Definitionen für Klassen von Ressourcen wiederverwenden.

Zur Festlegung eines Informationsmodells muß eine Modellierungstechnik und eine geeignete Syntax zur Beschreibung der Managementinformation gewählt werden. Im Managementbereich wurden für die Modellierung u.a. der Entity-Relationship-Ansatz, der Datentypansatz oder der objektorientierte Ansatz eingesetzt, wobei sich letzterer als am leistungsfähigsten herausgestellt hat.

Bei Verwendung des objektorientierten Ansatzes werden die Ressourcen durch Managementobjektklassen repräsentiert. Die Anwendungen operieren dann über die Managementschnittstelle (*Managed Object Boundary*) auf Instanzen dieser Objektklassen, den sog. *Managed Objects* (MO). Siehe hierzu Abbildung 2.2.

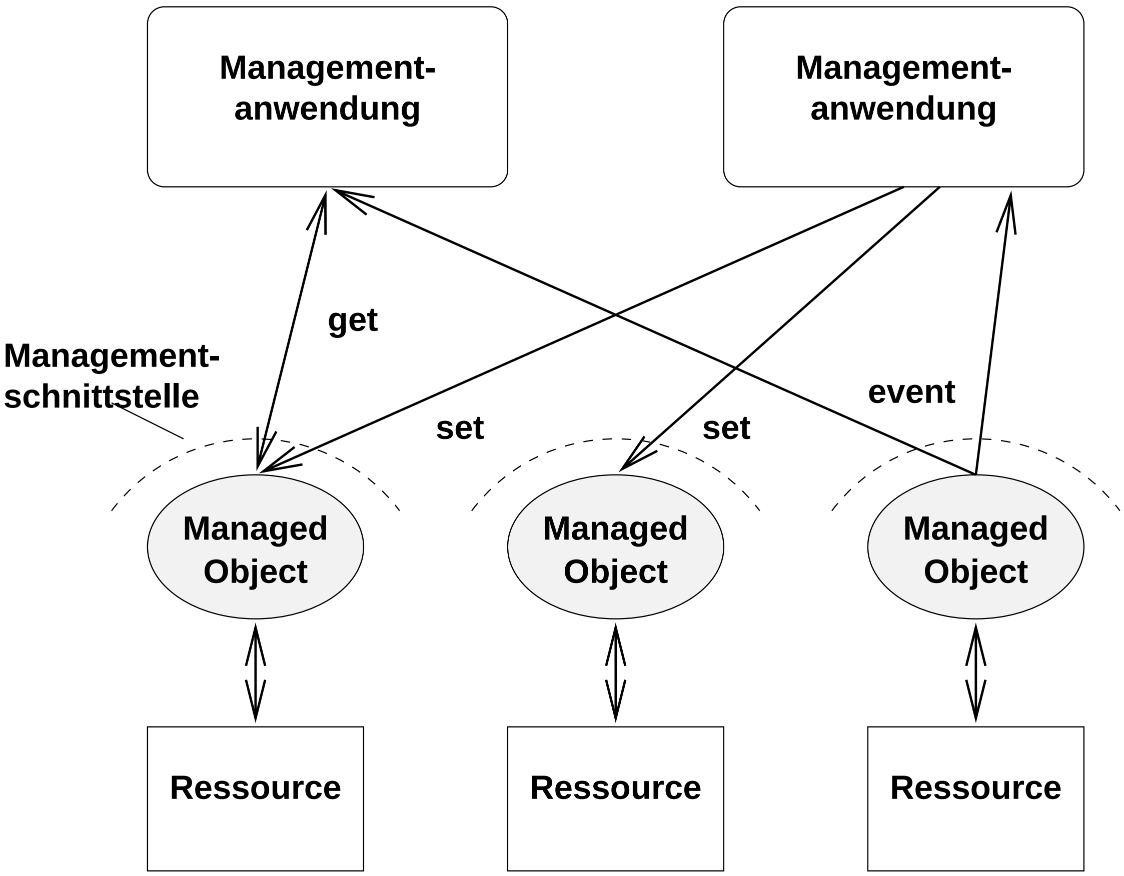


Abbildung 2.2: Managementmodelle zu Ressourcen

Managementinformation und -funktionalität für die Objekte werden durch Attribute und Methoden der Klasse modelliert. Attribute legen die Eigenschaften und das Verhalten einer Ressource fest. Sie können durch den Manager ausgelesen und evtl. verändert werden. Die Methoden definieren komplexe Aktionen, die auf dem Objekt ausgeführt werden können. Asynchrone Meldungen sind Botschaften über ein aufgetretenes Ereignis, die das MO an seinen Manager senden kann.

Im objektorientierten Modell können Enthaltenseins- und Vererbungshierarchien zwischen den Klassen definiert werden. Enthaltensein bedeutet, daß ein Managed Object aus mehreren anderen aufgebaut sein kann. Bei der Vererbung erbt eine Unterklasse alle Attribute und Methoden ihrer Oberklasse(n) und kann durch Hinzufügen weiterer Attribute und Methoden spezialisiert werden. Diese Eigenschaft unterstützt einen Top-Down-Ansatz bei der Modellierung besonders gut. Dadurch können generische Klassen für Ressourcen eingeführt werden, die nur von den Anforderungen des Managements an die Klasse dieser Ressourcen geprägt sind. Durch Spezialisierung der generischen Basisklassen können im Modell aber auch konkrete Ausprägungen von Ressourcen berücksichtigt werden.

Allomorphie erlaubt die Behandlung eines Objekts einer spezifischen Klasse entweder als Instanz von dieser Klasse oder als Instanz von einer seiner Oberklassen. Dies erlaubt die Entwicklung von Managementanwendungen für unterschiedliche Ebenen des gleichen Modells. Auf der Ebene der Basisklassen kann eine Anwendung mit eingeschränktem Funktionsumfang auf viele unterschiedliche Ausprägungen einer Ressource zugreifen. Auf der Ebene der spezifischen Klassen arbeitet ein Werkzeug für eine spezielle Ressource.

2.2 Anforderungsanalyse für das Modell

In den folgenden Abschnitten soll diskutiert werden, welche Anforderungen an das Informationsmodell zu stellen sind, die bei der späteren Modellierung berücksichtigt werden müssen.

Ziele des integrierten Managements

Integriertes Management soll es ermöglichen, alle Ressourcen einer verteilten Systemumgebung auf einheitliche Art und Weise verwalten und steuern zu können. Aus einzelnen Modulen soll auf Basis einer Managementarchitektur eine Managementlösung für das Gesamtsystem entstehen können. Integriertes Management liegt *nicht* vor, wenn Ressourcen aus dem Netz-, System- und Anwendungsmanagement isoliert behandelt werden und ein Nebeneinander von Werkzeugen für spezielle Ressourcen besteht.

Hieraus lassen sich bereits zwei wichtige Anforderungen an das Informationsmodell aufstellen. Da sich das Systemmanagement, wie in 2.1.2 beschrieben, in verteilter Umgebung nicht mehr vom Netz- und Anwendungsmanagement abgrenzen läßt, darf das Modell nicht auf eine Disziplin beschränkt sein. Es muß also beispielsweise möglich sein, so unterschiedliche Objekte wie eine offene Kommunikationsverbindung, eine Benutzerkennung und einen Server für einen Systemdienst mit dem selben Modellierungsansatz beschreiben zu können. Hier liegt bereits die Verwendung des objektorientierten Ansatzes auf der Hand. Durch dessen Kapselung von Daten und zugehöriger Funktionalität ist damit eine Abstraktion von beliebigen in der Realität beobachtbaren Objekten möglich. Für das Management relevante Information und Funktionalität kann einheitlich durch Attribute und Methoden modelliert werden.

Die zweite Anforderung ergibt sich aus der Heterogenität der Komponenten eines verteilten Systems. Für das integrierte Management muß die Heterogenität verschattet werden. Auch dies ist mit dem objektorientierten Ansatz möglich. Managementobjektklassen definieren Information und Funktionalität für Klassen von Ressourcen und abstrahieren somit von proprietären Details. Durch Spezialisierung sind aber sehr spezifische Werkzeuge nicht ausgeschlossen.

Zukunftssicherheit

Sowohl das Modell als auch die Modellierungstechnik soll zukunftssicher sein. Das Modell sollte also möglichst auch auf zukünftige Systeme anwendbar und darf somit nicht auf bestimmte Systemarchitekturen oder Betriebssysteme beschränkt sein. Obwohl später ein konkretes Modell für das Management von UNIX-Systemen und UNIX-Systemdiensten entwickelt wird, werden durch das Top-Down-Vorgehen generische Basisklassen eingeführt, die sich für beliebige existierende und zukünftige Systeme bzw. Dienste verwenden lassen. Da sich die Objektorientierung in vielen Bereichen, z.B. bei den Datenbanken, Programmiersprachen (C++, Java) und bei verteilten Client/Server-Umgebungen (CORBA), durchzusetzen beginnt oder bereits etabliert ist, stellt der objektorientierte Modellierungsansatz ebenfalls eine zukunftssichere Wahl dar. Dieser gewährleistet außerdem die einfache Implementierbarkeit des Modells mit einer objektorientierten Programmiersprache.

Zur Zukunftssicherheit gehört aber auch eine weitere wichtige Anforderung an die Modellierung, die im nächsten Abschnitt beschrieben wird.

Unabhängigkeit von der Managementarchitektur

Managementarchitekturen bilden die Basis für integrierte Managementlösungen. Das Problem „Management" wird in verschiedene Teilmodelle strukturiert. Neben dem hier besprochenen Informationsmodell gibt es noch das Organisationsmodell für Akteure und Rollen im Management, das Kommunikationsmodell zur Beschreibung der Kommunikationsinfrastruktur und das Funktionsmodell zur Definition der Funktionalität. Für eine detaillierte Beschreibung von Architekturen sei auf [HA93] und [HNW95] verwiesen.

Obwohl für die Integration eher kontraproduktiv, haben sich mehrere völlig unterschiedliche Managementarchitekturen auf dem Markt etabliert und weitere werden spezifiziert. Die Konzepte und Techniken der Architekturen bewegen sich eher auseinander. Es kann nicht davon ausgegangen werden, daß sich eine davon in näherer Zukunft durchsetzen wird. Die folgende Tabelle faßt Merkmale der wichtigsten Architekturen zusammen.

Architektur	Informationsmodell	Protokoll	Funktionsmodell
ISO/OSI	objektorientiert, GDMO-Templates	CMIP	Systems Management Functions
IETF Internet	objektbasiert, ASN.1 Datentypen	SNMP (v3)	nicht vorhanden
OMG CORBA	objektorientiert, IDL	IOP	CORBA Services, Facilities, Domain Interfaces
WBEM	objektorientiert, CIM	HMMP / DCOM	??

Tabelle 2.1: Übersicht über Managementarchitekturen

Hieraus resultiert die Anforderung, daß das Modell architekturunabhängig sein soll, um die Zukunftssicherheit nicht durch Festlegung auf eine Architektur zu gefährden. Für eine breite Anwendbarkeit sollte es vielmehr auf die Informationsmodelle der unterschiedlichen Architekturen abbildbar sein. Im optimalen Fall wird diese Abbildung durch ein Werkzeug unterstützt.

Zusammenfassung der Anforderungen

Folgende Liste faßt die Anforderungen an ein Managementmodell für Endsysteme und verteilte Systemdienste nochmals zusammen. Im nächsten Abschnitt wird beschrieben, wie in dieser Arbeit vorgegangen wird, um ein Modell zu gewinnen, welches den Anforderungen möglichst gerecht wird.

- Das Modell soll sich auf verschiedenste Ressourcen der drei Disziplinen System-, Software- und Dienstmanagement abbilden lassen.
- Die Modellierungstechnik muß mächtig genug sein, um auch komplexe Managementinformation und Funktionalität aller Funktionsbereiche modellieren zu können.
- Das Modell soll die Heterogenität der Ressourcen verschatten und unabhängig von konkreten Systemen bzw. Diensten sein.
- Eine flexible Abbildung auf konkrete Programmiersprachen ist Voraussetzung.
- Die Anwendbarkeit und Erweiterbarkeit des Modells auf neue Systemumgebungen und neue Ressourcen soll die Zukunftssicherheit gewährleisten.
- Das Modell soll möglichst unabhängig von einer Managementarchitektur sein. Gefordert wird allerdings die leichte Abbildbarkeit auf die Syntax verbreiteter Informationsmodelle.
- Eine möglichst gute Unterstützung durch Werkzeuge (CASE-Tools) für die Modellierung, Abbildung auf konkrete Informationsmodelle und ggf. Erzeugung von Agentencode ist wünschenswert.

2.3 Gewinnung des Objektmodells

Abbildung 2.3 zeigt die Vorgehensweise zur Gewinnung des Objektmodells. Die Einzelschritte, die bei der Top-Down-Modellierung durchgeführt werden, sind im folgenden erklärt.

Analyse

Aufgrund der oben beschriebenen Vorteile wird der objektorientierte Ansatz bei der Modellierung verfolgt. Die erste Phase der objektorientierten Modellierung ist die Analyse. Durch die Analyse soll unter Berücksichtigung der Problemstellung ein Modell der realen Welt entwickelt werden. Unter dem Managementgesichtspunkt bedeutet Analyse die Identifikation geeigneter Managementobjektklassen und deren Beziehungen untereinander.

Die Suche nach Managementobjektklassen stützt sich auf die Anforderungen des Betreibers für das Systemmanagement. Durch Untersuchung der Aufgabenbereiche eines Systemadministrators (siehe 2.1.2) sollen die für das Management relevanten Objekte gefunden werden. In der ersten Phase ist die Frage nach dem «*Was*» d.h. „Welche Komponenten werden für das Management benötigt?“ wichtiger als nach dem «*Wie*» d.h. „Welche spezifische Information und Funktionalität wird für die Ressource benötigt?“.

Eine bedeutende Anforderung an das Modell ist die angestrebte Anwendbarkeit der Klassen auf beliebige heterogene, verteilte Systeme. Gleichzeitig soll das Modell Aspekte aller Managementdisziplinen abdecken. Beispielsweise gehört zum Management verteilter Systemdienste neben der Installation und Konfiguration der Software auch die Überwachung des Betriebs, also das Monitoring von Prozessen und offenen Kommunikationsverbindungen. Um für all diese Aspekte generische Basisklassen zu finden, werden die Konzepte und Viewpoints des RM-ODP (siehe Kapitel 3.5) analysiert. RM-ODP ist ein abstrakter Architekturrahmen zur Spezifikation und Entwurf von Anwendungen in offener, verteilter Systemumgebung. Dieser

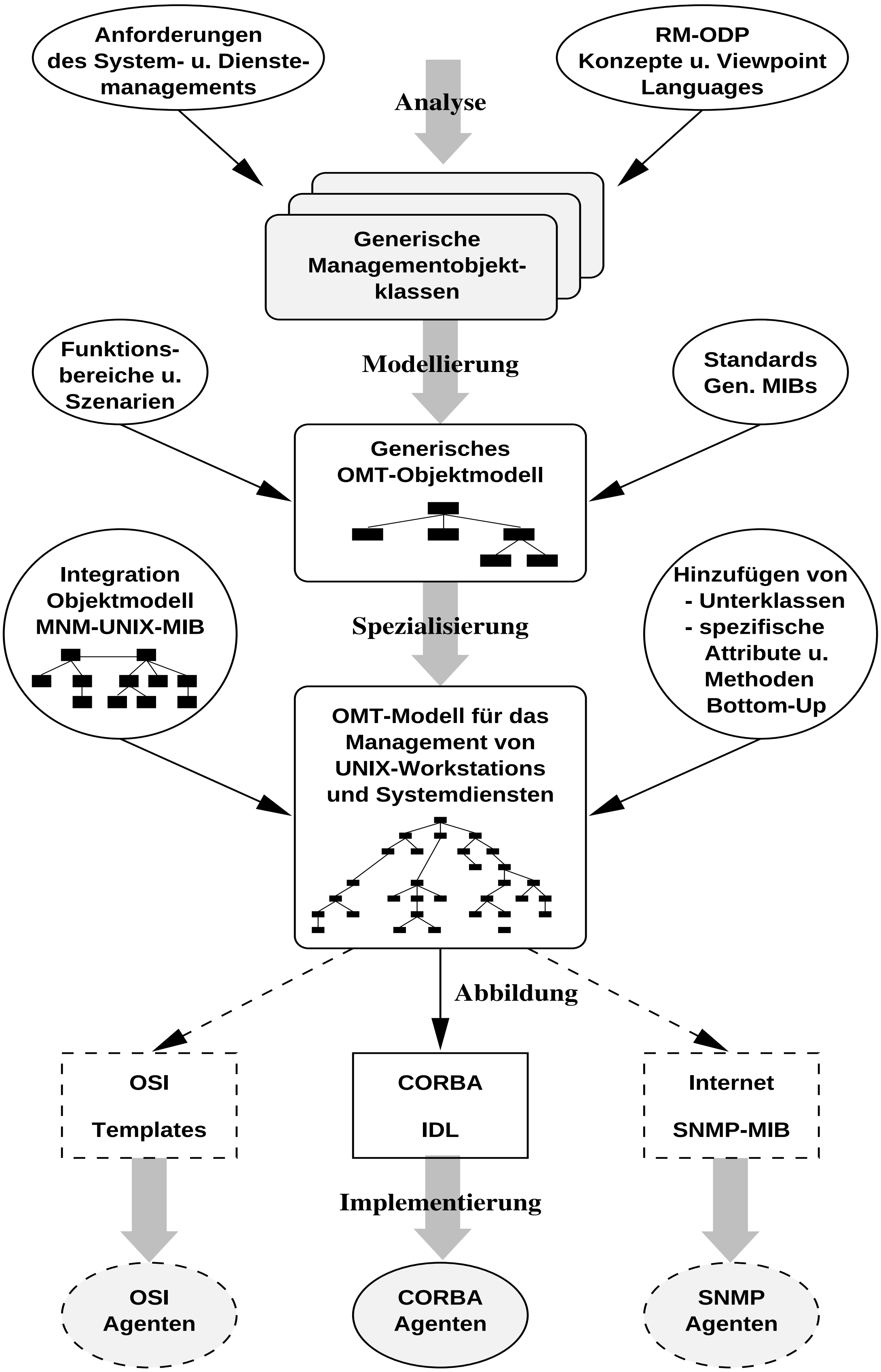


Abbildung 2.3: Gewinnung des Objektmodells durch Top-Down-Modellierung

Ansatz erscheint vielversprechend, da dieses Modell nicht nur Komponenten zum Entwurf von verteilten Anwendungen, sondern auch die erforderlichen Infrastrukturobjekte zur Realisierung auf einer verteilten Plattform spezifiziert. Da das Management im RM-ODP nicht im Vordergrund steht, müssen die Objekte des Modells natürlich auf diesbezügliche Relevanz überprüft werden. Weiterhin ist die Tatsache wichtig, daß es sich bei RM-ODP um einen offenen ISO-Standard handelt, der von der OMG mitgetragen wird. Es ist zu hoffen, daß Modelle, die auf diesem Standard basieren, breite Akzeptanz bei den bedeutenden Software-Herstellern finden.

Ergebnis der Analysephase ist eine Menge von generischen – vorerst noch leeren – Objektklassen, die möglichst viele Aufgabenbereiche des Systemmanagements abdecken.

Modellierung

Der nächste Schritt ist die Anordnung der Klassen in einem Objektmodell auf Basis der *Object Modeling Technique* (OMT). Nähere Informationen zu OMT finden sich in Kapitel 3.4 und in [RBP+91]. Für den Einsatz von OMT sprechen folgende Gründe. OMT ist seit langem etabliert, weit verbreitet und dementsprechend gut dokumentiert. Die für diese Arbeit genutzten Techniken von OMT finden sich außerdem praktisch unverändert in UML wieder, welche sich aufgrund der Verschmelzung der wichtigsten Modellierungsmethoden zum Quasi-Standard entwickeln wird. Darüber hinaus wurde am Lehrstuhl das CASE-Tool *Software through Pictures* (StP) angeschafft, welches die Erstellung von OMT-Modellen unterstützt und aus diesen für diverse objektorientierte Programmiersprachen Code-Gerüste erzeugen kann.

Das Objektmodell beschreibt die statische Struktur der Managementobjekte eines verteilten Systems und ihre Relationen. Die ungeordneten Basisklassen aus der Analysephase können im Objektmodell durch Assoziationen, Enthaltenseins- und Vererbungshierarchien zueinander in Beziehung gesetzt werden.

Zur Modellierung der Basisklassen gehört natürlich auch die Definition von Managementinformation und -funktionalität. Basisklassen machen nur dann einen Sinn, wenn sie nicht als leere Container für die Wurzel des Vererbungsbaums dienen, sondern einer Managementanwendung, die auf der Ebene der Basisklassen operiert, bereits soviel Information bereitstellen, um eine große Anzahl verschiedenster, abgeleiteter Ressourcen *gleichartig* behandeln zu können. Beispielsweise ist die Überwachung des Status von Dienstkomponenten in einem verteilten System eine wichtige Aufgabe. Enthält eine generische Klasse *Software-Komponente*, von der alle Dienste und Anwendungen abgeleitet werden können, demnach ein Attribut für den Betriebszustand, kann ein *einziges* Werkzeug, welches die Basisklasse für jeden Dienst instantiiert, den Status aller Dienste eines verteilten Systems gleichzeitig überwachen.

Es geht also darum, den Basisklassen möglichst viel Information und Funktionalität zuzuordnen, um ein effizientes Management von Ressourcen auf dieser Ebene zu ermöglichen. Trotzdem darf die Allgemeinheit der Klasse, d.h. die Anwendbarkeit auf unterschiedlichste Objekte nicht eingeschränkt werden. Da OMT strikte Vererbung fordert, muß nämlich sichergestellt werden, daß zumindest ein großer Teil der in der Basisklasse definierten Information auch von einer spezifischen Ressource bereitgestellt wird. Sonst ist eine Implementierung eines Agenten auf Basis dieses Modells für die Ressource unmöglich.

Attribute und Methoden für die Basisklassen können durch Analyse der Managementfunktionsbereiche und Erstellung von Szenarien abgeleitet werden. Weiterhin können die für einige Felder des System- und Software-Managements bereits existierenden Standards und generischen MIBs als Quelle dienen. Eine dritte Möglichkeit ist ein Bottom-Up-Vorgehen. Hierbei werden bestehende Modelle oder MIBs für spezielle Ressourcen dahingehend untersucht, ob die definierte Information oder Funktionalität evtl. in allgemeinerer Form in Ober- bzw. Basisklassen verlagert werden kann.

Ziel dieses Schritts ist ein OMT-Objektmodell aus *wenigen* Basisklassen, die über einen *hohen* Informationsgehalt verfügen und somit einem Werkzeug die Überwachung und Manipulation *vieler* Managed Objects ermöglicht.

Spezialisierung

Der nächste Modellierungsschritt ist die Spezialisierung der generischen Basisklassen. Diese soll zu einem Modell für das Systemmanagement einer UNIX-Workstation führen. Zusätzlich werden die Basisklassen, die für das Anwendungsmanagement eingeführt wurden, zu Klassen für das Management von (verteilten) Systemdiensten unter UNIX verfeinert (vgl. Kapitel 5).

Die bisherige Modellierung war systemunabhängig. Nun werden ein konkretes System und ausgewählte Systemdienste untersucht und modelliert, um die Anwendbarkeit der Basisklassen und somit die Tragfähigkeit des generischen Objektmodells zu überprüfen. Von den Basisklassen werden hierzu Unterklassen abgeleitet und um spezifische Attribute und Methoden ergänzt. Dazu wird eine Bottom-Up-Analyse konkreter Ressourcen durchgeführt. Hierbei wird die für das Management der Ressource erforderliche Information und Funktionalität ermittelt. Nach der Abbildung der von den Oberklassen ererbten Attribute und Methoden auf die Ergebnisse der Bottom-Up-Analyse werden die noch fehlenden Attribute und Methoden zur Unterklasse hinzugefügt.

In diesem Modellierungsschritt wird außerdem ein bestehender Prototyp eines Objektmodells für das integrierte Systemmanagement von Workstations (siehe [Sir96]), welcher auf der MNM-UNIX-MIB [KH96] basiert, in das Modell integriert. Gelingt die Integration ohne großen Aufwand, rechtfertigt dies ebenso die Definition der Basisklassen, da Modelle für konkrete Systeme wiederverwendet werden können. Die Möglichkeit der Wiederverwendung (von Klassen und Code) ist gerade ein Aspekt, den objektorientierte Entwurfstechniken für sich beanspruchen.

Im optimalen Fall ist nach diesem Schritt ein Objektmodell entstanden, welches den in Abschnitt 2.2 formulierten Anforderungen gerecht wird. Auf die geforderte Unabhängigkeit von einer Managementarchitektur wird weiter unten eingegangen. Anhand der Basisklassen sollte die Entwicklung generischer Managementanwendungen möglich sein. Die spezifischen Unterklassen erlauben die Konstruktion von Werkzeugen, die auf konkrete Ressourcen abgestimmt sind.

Zu erwähnen bleibt noch, daß die skizzierte Vorgehensweise über die Schritte *Analyse*, *Modellierung* und *Spezialisierung* nicht streng sequentiell zu verstehen ist. Beispielsweise kann die Untersuchung konkreter Systeme zu Änderungen von Basisklassen oder sogar zur Neueinführung führen. Die Managementfunktionsbereiche und existierende MIBs vor allem aus der Welt des Internet-Managements beeinflussen natürlich auch die Modellierung der spezifischen Unterklassen.

Abbildung auf Managementarchitekturen

Das OMT-Modell erfüllt ebenfalls die geforderte Unabhängigkeit von einer Managementarchitektur. An dieser Stelle wird kurz skizziert, wie sich das Modell auf die Syntax der drei wichtigsten Architekturen abbilden läßt.

OSI-Management: Das OSI-Informationsmodell ist ebenso objektorientiert. Die Objektklassen mit Attributen und Methoden können problemlos auf *GDMO-Templates* der OSI-Metasprache zur Beschreibung von Managementobjekten abgebildet werden. Vererbung und Enthaltenseinshierarchien werden direkt unterstützt. Auch die Definition von komplexen Attributtypen ist möglich. Einer werkzeugunterstützten Abbildung steht daher nichts im Wege.

OMG CORBA: Obwohl CORBA (siehe Kapitel 3.6) keine reine Managementarchitektur darstellt, sondern nur die Infrastruktur zur Kommunikation verteilter Objekte standardisiert und durch den ORB bereitstellt, soll es in dieser Arbeit als Realisierungsumgebung für das Objektmodell dienen. Objekte werden in CORBA durch die Sprache IDL beschrieben. Das Case-Tool StP kann aus dem OMT-Objektmodell direkt die IDL-Spezifikation der Klassen generieren. Auf das weitere Vorgehen zur Implementierung des Modells und auf die Vorteile von CORBA wird im folgenden Abschnitt 2.4 eingegangen.

Internet-Management (SNMP): Die Abbildung des Objektmodells ist hier bei weitem schwieriger, da das Informationsmodell nicht objektorientiert ist, sondern auf dem Datentypansatz beruht. Vererbung, Enthaltenseinbeziehungen, komplexe Attribute und Methoden fehlen vollständig. Es ist möglich, Objektklassen konkreter Modelle manuell auf SNMP-Tabellen und -Gruppen abzubilden. Ein Beispiel hierfür findet sich in [Hai95]. Außerdem ist die Implementierung des Modells auf Basis einer anderen Architektur und die Verwendung eines Management-Gateway für den Zugriff von SNMP-basierten Anwendungen möglich.

Der letzte Schritt nach der Abbildung des Modells auf eine bestimmte Architektur ist die Implementierung von Agenten für die Managed Objects mit Hilfe einer entsprechenden Entwicklungsumgebung.

2.4 Realisierung des Prototypen

Agenten realisieren ein Managementmodell, indem sie Objektklassen des Modells instantiieren und somit Managed Objects bereitstellen, welche Abstraktionen von realen Ressourcen für eine Managementanwendung sind. Die Anwendung kommuniziert mit dem Agenten, um Informationen über eine Ressource zu erlangen oder auf diese einzuwirken. Die Low-Level-Interaktionen zwischen dem Agenten und der Ressource bleiben der Anwendung verschattet, die nur über die Attribute und Methoden der *Managed Object Boundary* auf das Objekt zugreifen kann. In diesem Abschnitt wird die prinzipielle Vorgehensweise vorgestellt, die in dieser Arbeit zu einem Prototypen für einen Agenten führt, der Teile des Objektmodells implementiert. Abbildung 2.4 illustriert sowohl die Erzeugung des Modells (linke Hälfte), die Implementierung der Agenten (untere Hälfte) und die Verwendung des Modells und der Agenten in einer Managementlösung (rechte, obere Hälfte).

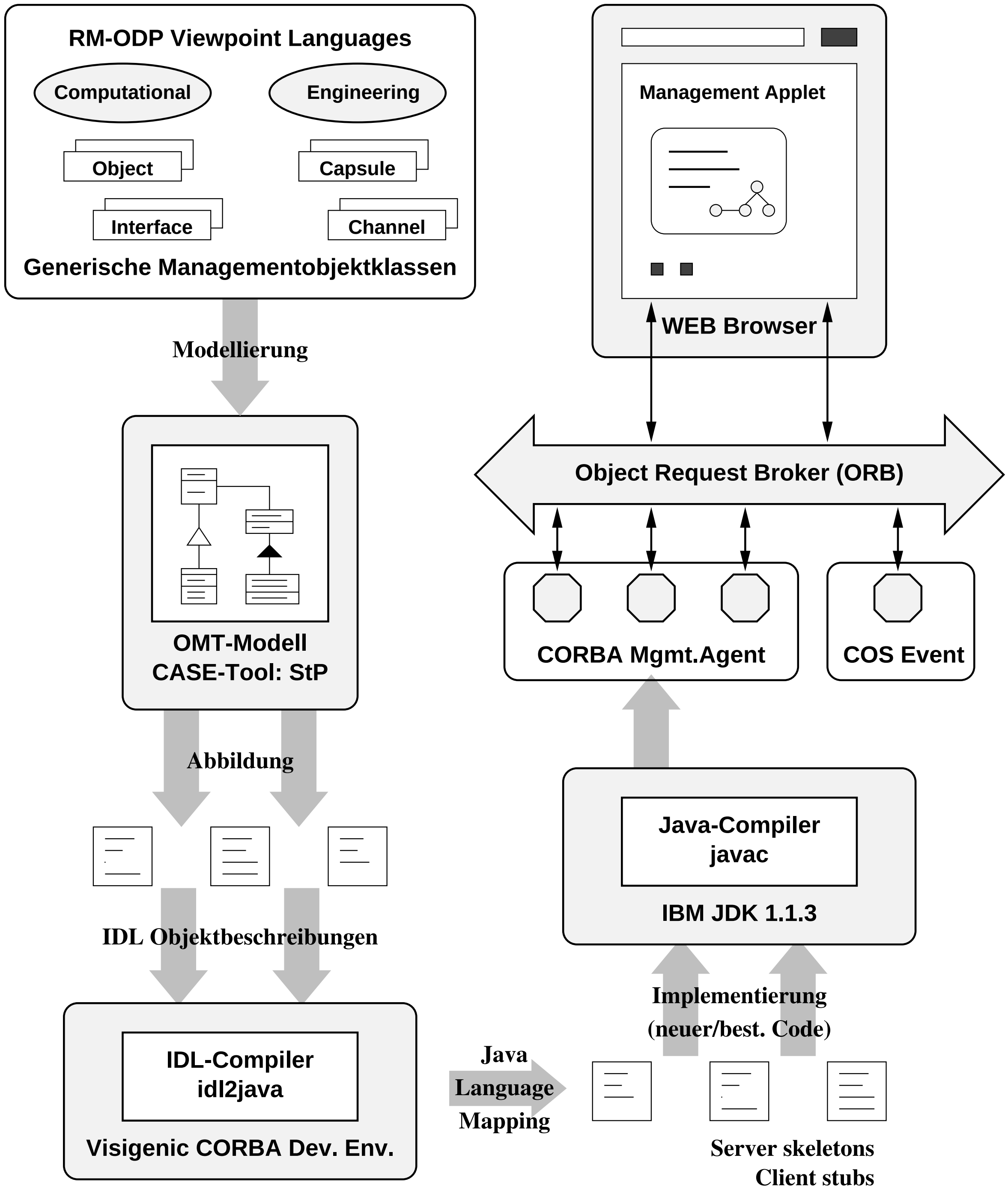


Abbildung 2.4: Vorgehensmodell für die Realisierung des Agentenprototyps

Die Top-Down-Modellierung beginnt, wie bereits beschrieben, mit der Suche nach relevanten Managementobjektklassen. Hierzu werden Konzepte aus den *RM-ODP Viewpoint Languages* – vor allem aus dem *Computational* und *Engineering* Viewpoint – für die Belange des Managements herangezogen. Die weitere Modellierung gemäß Kapitel 2.3 wird durch das CASE-Tool *Software through Pictures* (StP) [Int96b] unterstützt. Der Einsatz dieses Werkzeugs wird in

Kapitel 6.1.1 beschrieben. Das OMT-Modell wird im *Object Model Editor* von StP bearbeitet.

Der Agent soll auf Basis von OMG CORBA (siehe Kapitel 3.6) implementiert werden. Zur Beschreibung von Objektklassen benutzt CORBA die von konkreten Programmiersprachen unabhängige *Interface Definition Language*. Mittels IDL werden die Attribute und Methoden einer Klasse durch Typisierung und Angabe von Parametern festgelegt. IDL unterstützt Vererbung und die Definition komplexer Typen.

StP kann für die Klassen des Objektmodells Code in den Sprachen Ada, C++, IDL, Java und Smalltalk erzeugen. Hierzu ist es erforderlich, daß in der Klassentabelle von StP alle Attributtypen sowie die Argumente, Rückgabewerte und Ausnahmen der Methoden deklariert werden.

Aus den IDL-Objektbeschreibungen generiert ein IDL-Compiler im nächsten Schritt *Server skeletons* und *Client stubs* für eine konkrete Programmiersprache. Der IDL-Compiler ist Bestandteil der CORBA-Entwicklungsumgebung. Die Abbildung von IDL auf eine Programmiersprache wird von der OMG durch sog. *Language Mappings* standardisiert. Der hier eingesetzte IDL-Compiler *idl2java* gehört zum Produkt *VisiBroker for Java V3.0* der Firma Visigenic Software, Inc. und erzeugt Code-Gerüste für die Sprache Java (siehe Kapitel 6.3.1).

Der nächste Schritt ist die Implementierung der Server-Objekte, die einen CORBA-Agenten bilden. Hierzu werden die Code-Gerüste des IDL-Compilers mit Java-Code ergänzt, welcher die Attribute und Methoden der MOCs implementiert. Dieser Code beschafft die managementrelevante Information bzw. realisiert die Manipulationsmöglichkeiten der Ressource durch eine Anwendung. Eine Wiederverwendung von bestehendem C-Code eines SNMP-Agenten ist möglich, falls dieser in einzelne Prozeduren bzw. Module für das Lesen und Schreiben von Attributen gegliedert ist. Die Module können durch das *Java Native Interface* (siehe Kapitel 6.3.4) in die Server-Objekte integriert werden. Dadurch geht natürlich die Plattformunabhängingkeit des Java-Codes verloren. Da aber der Code eines Agenten für das Systemmanagement durch die Heterogenität der Systeme in den seltensten Fällen plattformunabhängig ist, wird der Verlust durch die Einsparung der Neuimplementierung aufgewogen. Eine Kapselung des SNMP-Codes in Objektklassen eignet sich prinzipiell auch für die Migration von Altsystemen (*legacy systems*) sowie zur Realisierung eines multiarchitekturellen Agenten [HKN96].

Nach Abschluß der Implementierung und Übersetzen der Server-Objekte durch einen Java-Compiler liegt der Prototyp eines CORBA-Agenten vor. Die Laufzeitumgebung von *VisiBroker for Java V3.0* besteht aus einem *Object Request Broker* (ORB) nach der CORBA-Spezifikation 2.0 und den CORBA-Services *Naming* und *Event*. Asynchrone Meldungen der Managed Objects werden über den Event-Service den Managementanwendungen zugestellt.

Um eine graphische Oberfläche für den Zugriff auf die Managed Objects bereitzustellen, wird für den Prototypen eine Anwendung in Form eines Java-Applet implementiert. Das Management-Applet ruft als Client Methoden der Server-Objekte des Agenten auf. Der ORB realisiert die Kommunikation zwischen den verteilten Objekten. Die Implementierung des Managers als Java-Applet hat den Vorteil, daß die Anwendung system- und ortsunabhängig in jedem Web-Browser, der Java unterstützt, ausgeführt werden kann. Weiterhin sind Änderungen und Erweiterungen leicht möglich, da sie keine Neuinstallationen der Software auf den Managersystemen nach sich ziehen. Der *Navigator* der Firma Netscape integriert seit Version 4.0 zudem den ORB – allerdings in der alten Version 2.5 – von Visigenic.

2.4.1 Vorteile von CORBA

In diesem Abschnitt soll die Verwendung von CORBA als Realisierungsumgebung für den Prototypen des Agenten motiviert werden. Hierzu werden die Vorteile von CORBA für das System- und Anwendungsmanagement gegenüber dem in diesen Bereichen bisher dominierenden Internet-Management zusammengefaßt.

- Wie bereits erwähnt, sollen die Konzepte des RM-ODP dazu benutzt werden, geeignete generische Objektklassen für das System- und Anwendungsmanagement zu spezifizieren. CORBA ist eine standardisierte Architektur zur Bereitstellung der Infrastruktur für verteilte objektorientierte Programmierung. Sie gilt als die Referenzrealisierung für einige Konzepte aus dem Engineering Viewpoint des RM-ODP. Die Standardisierungsgremien der ISO und OMG arbeiten eng zusammen, um die Modelle untereinander konsistent zu halten. Da es sich bei der OMG um eine Vereinigung großer Software-Hersteller handelt, ist davon auszugehen, daß die Konzepte des RM-ODP mit Hilfe von CORBA auch den Weg in konkrete Produkte finden werden. Es ist daher zu erwarten, daß in Zukunft integrierte Managementlösungen, also Agenten, Plattformen *und* Anwendungen, basierend auf CORBA entwickelt werden. Ein Beispiel für eine bereits existierende, kommerzielle Managementlösung auf Basis von CORBA ist Tivoli TME.

- Der objektorientierte Ansatz hat sich gegenüber dem Informationsmodell des Internet-Managements als geeigneter für die Definition von Managed Objects herausgestellt. Bei der Verwendung von SNMP beschränkt sich das Management auf das Lesen und Schreiben einfacher Variablen. Außer zweidimensionalen Tabellen gibt es bisher keine komplexeren Datentypen. Beziehungen zwischen Objekten müssen aufwendig über Indexvariablen und Indextabellen konstruiert werden. Das Auslösen von Aktionen auf einer Ressource ist durch sog. „Pushbutton-Variablen“ unnatürlich modelliert.

  CORBA erlaubt durch die Sprache IDL die objektorientierte Modellierung der Ressourcen. IDL unterstützt die Definition komplexer Datenstrukturen für Attribute. Die Methoden, die die Funktionalität für den Zugriff darstellen, werden gemeinsam mit den Datenstrukturen in einem CORBA-Objekt gekapselt. Managementaktionen auf Ressourcen können ebenfalls anschaulich durch Methoden modelliert werden. Weiterhin ist durch Vererbung die Wiederverwendung definierter Information und Funktionalität möglich. Hierdurch wird die unübersichtliche, redundante und proprietäre Definition von Information im Internet-Registrierungsbaum vermieden.

- Da das Internet-Management kein Funktionsmodell besitzt, welches die Delegierung von Funktionalität an die Agenten erlaubt, müssen Aufgaben wie das Sammeln und Auswerten von Leistungsdaten, das Führen von Logs und die Verarbeitung von asynchronen Meldungen zentral durch die Managementplattform oder Anwendung erledigt werden. In großen Systemen kann dies aber zu Engpässen bei den Rechen- und Kommunikationsressourcen führen. CORBA erlaubt die Implementierung leistungsfähiger Agenten, die eigene Funktionalität für obige Aufgaben besitzen können. Darüber hinaus können diese Funktionen aufgrund der Infrastruktur von CORBA auch von verteilten Managementkomponenten (Server-Objekten) erbracht werden.

- Von CORBA erhofft man sich auch erhebliche Effizienzsteigerungen beim Zugriff auf die Managementinformation der Ressourcen. Tabellen können bei SNMP nur sequentiell durch zahlreiche Einzelanforderungen (`GetRequest, GetNextRequest`) ausgelesen werden. Bei CORBA ist neben dem wahlfreien Zugriff auf Einzelattribute auch die Übertragung einer großen Menge an Information durch *eine* Anforderung bis zur Kopie ganzer Objektinstanzen möglich. Da sich aber CORBA-konforme *Object Request Broker* der Version 2 erst kurze Zeit auf dem Markt befinden, muß die tatsächliche Effizienz der Implementierungen evaluiert werden.

- Die ebenfalls standardisierten CORBA-Services, die nach und nach von den ORB-Herstellern angeboten werden, vereinfachen die Implementierung einer Managementlösung auf Basis von CORBA.

  Der *life cycle service* vereinheitlicht das Anlegen, Kopieren und Löschen von Objekten. Der *naming service* erlaubt die hierarchische Anordnung von Objekten in einem Registrierungsbaum und ermöglicht somit eine effiziente Suche nach Objekten. Möglicherweise könnte er auch zur Implementierung des vom OSI-Kommunikationsmodell bekannten *Scoping* eingesetzt werden. Der *event service* definiert ein Kommunikationsmodell für das Senden und Empfangen von asynchronen Meldungen. Darüber hinaus können spezielle Server-Objekte die Events filtern und korrelieren. Beziehungen zwischen Managementobjekten, wie z.B. Enthaltenseinshierarchien, können auf den *relationship service* abgebildet und somit besser kontrolliert werden. Der *security service* könnte nicht nur Anforderungen aus dem Sicherheitsmanagement abdecken, sondern auch Sicherheitsmechanismen innerhalb des Managementsystems implementieren.

- Wie oben bereits erwähnt, kann durch Kapselung von Code bestehender Agenten in CORBA-Objekte sowohl die Wiederverwendung des Codes als auch der Anschluß von Altsystemen (*legacy systems*) an neue, objektorientierte Systemumgebungen realisiert werden. Hierzu ist es allerdings erforderlich, daß die Agentenprozeduren modularisiert und der Quellcode offen zugänglich ist.

- CORBA soll die Entwicklung verteilter Anwendungen in heterogenen Systemumgebungen erleichtern. Damit eignet es sich implizit auch als Realisierungsumgebung für verteilte Anwendungen für das integrierte System- und Anwendungsmanagement. Anwendungen und Management nutzen die gleiche Kommunikationsinfrastruktur. CORBA-Management ist auch, anders als SNMP, nicht auf eine Protokollwelt beschränkt, da das Inter-ORB-Protocol IOP auf verschiedenen Transportprotokollen aufsetzen kann. Ebenso ist die Spezifikation von Dienst- und Managementschnittstellen der Anwendungskomponenten einheitlich.

- Die Kooperation von Agenten verschiedener Hersteller auf einem System ist aufgrund des offenen Standards von CORBA problemlos möglich. Hierzu waren für SNMP Erweiterungen wie DPI oder DMI der DMTF erforderlich.

- Da die Beschreibung von CORBA-Objekten durch IDL unabhängig von einer Programmiersprache ist, ermöglicht der ORB die Kooperation von Objekten, die in unterschiedlichen Sprachen implementiert sind. Java erleichtert die Entwicklung portabler und ortsunabhängiger Managementanwendungen mit aufwendigen graphischen Benutzeroberflächen. Diese können über den ORB transparent auf Agenten zugreifen, die aufgrund ihrer Systemnähe möglicherweise in C++ programmiert wurden.

Kapitel 3

Umfeld und State-of-the-Art

3.1 Verteilte Systemdienste mit Client/Server-Prinzip

Nach einer kurzen Beschreibung des Client/Server-Prinzips werden in den folgenden Abschnitten die konkreten Dienste NFS und NIS eingeführt. In Kapitel 5 wird ein Objektmodell für das Management dieser Dienste entwickelt.

Die meisten Systemdienste sind nach dem Client/Server-Prinzip aufgebaut. Ein Dienstnehmer (*Client*) übergibt einen Auftrag an den Diensterbringer *Server* und wartet im allgemeinen auf seine Erfüllung. Der Server kann den Auftrag akzeptieren und bearbeiten. Nach erfolgreicher Bearbeitung sendet der Server dem Client die Antwort auf den Auftrag zu. Abbildung 3.1 verdeutlicht das Kooperationsprinzip. Sie zeigt nur den Standardfall der synchronen Bearbeitung eines Auftrags, bei der der Client bis zum Empfang der Antwort blockiert.

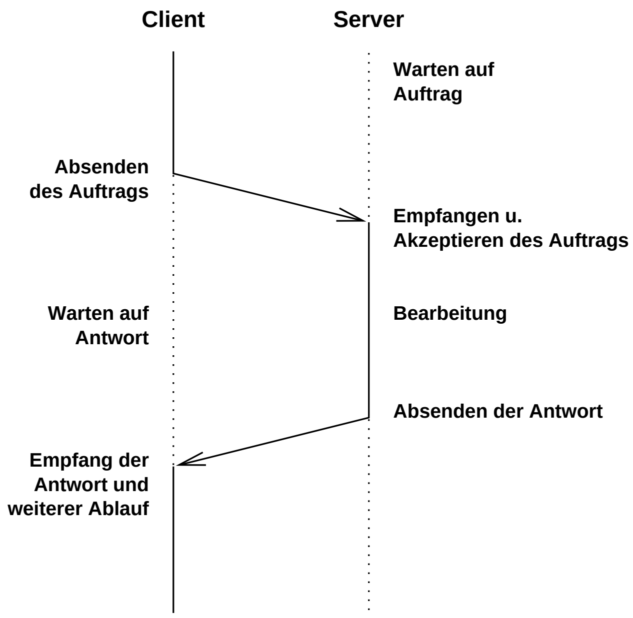


Abbildung 3.1: Das Client/Server-Prinzip

Bei verteilten Diensten befinden sich Client und Server meist nicht auf dem gleichen Rechensystem. Das Senden des Auftrags und die Übertragung der Antwort wird über ein Rechnernetz abgewickelt. Im allgemeinen nutzen mehrere Clients einen Dienst. Ein Dienst kann von einem Server allein zur Verfügung gestellt werden oder in Kooperation von mehreren Servern erbracht werden. Aus Gründen der Lastverteilung und Redundanz können auch mehrere Server den gleichen Dienst in einem Netz realisieren. Beispiele für verteilte Systemdienste oder systemnahe Dienste, die nach dem Client/Server-Modell arbeiten, sind:

Dateidienste: NFS (*Network File System*), AFS (*Andrew File System*), DFS (*Distributed File System*), FTP (*File Transfer Protocol*)

Verzeichnisdienste: DNS (*Domain Name Service*), NIS (*Network Information Service*), X.500 (*OSI-Directory*)

Nachrichtendienste (*electronic mail*): SMTP (*Simple Mail Transfer Protocol*), X.400 (*OSI-MHS*)

Druckdienste: Berkeley-lpd

Informationsdienste: WWW, gopher

3.2 Einführung in das Network-File-System (NFS)

Verteilte Dateisysteme ermöglichen Benutzern den gemeinsamen Zugriff auf Dateien im Netz, ohne daß sich die Benutzer jeweils auf dem Rechner anmelden müssen, auf dem sich eine Datei physikalisch befindet, bzw. ohne Duplizieren der Dateien auf mehrere Rechner. Die wichtigste Motivation für verteilte Dateisysteme ist die Bereitstellung der gleichen Arbeitsumgebung für den Benutzer unabhängig vom Arbeitsplatzrechner, an dem er angemeldet ist. Der einst sehr teure Plattenplatz verbot ein Duplizieren von Dateien und eine Installation großer Software-Pakete auf jedem Rechner von selbst. Aber auch heute gewinnen verteilte Dateisysteme bei zunehmender Vernetzung zusätzlich an Bedeutung, da die konsistente Datenhaltung vereinfacht wird.

Das Network-File-System (NFS) wurde seit 1984 von der Firma Sun Microsystems Inc. entwickelt und 1989 in [Sun89] standardisiert. Im UNIX-Bereich ist NFS das verteilte Dateisystem mit der größten Marktverbreitung, da Sun das zugrundeliegende Protokoll und eine Referenzimplementierung veröffentlicht hat. Praktisch jeder UNIX-Hersteller liefert NFS zusammen mit seinem Betriebssystem aus. NFS ermöglicht den transparenten Zugriff auf Dateisysteme, die sich physikalisch auf anderen Rechnern im Netz befinden. Hierzu stellen NFS-Server Teile ihrer lokalen Dateisysteme über das Netz anderen Computern zur Verfügung. Berechtigte Clients können diese Dateisysteme in ihr eigenes Dateisystem integrieren. Anwendungen sehen keinen Unterschied zwischen lokalen und entfernten Dateisystemen. Der Schreib- und Lesezugriff auf entfernte Dateisysteme ist transparent für Programme, die daher für NFS nicht geändert oder erweitert werden müssen. Hierzu wurde die Semantik des NFS-Filesystems möglichst eng an die eines normalen UNIX-Filesystems angelehnt. Eine umfassende Beschreibung von NFS bieten [San91] und [Ste91].

3.2.1 Das NFS-Protokoll und der RPC

Im OSI-Schichtenmodell ist NFS der Anwendungsschicht sieben zuzuordnen. Es basiert auf dem von Sun eigens für NFS entwickelten Remote-Procedure-Call (RPC Version 2), der in [Sun88] standardisiert ist[1]. Obwohl für NFS kein Transportprotokoll vorgeschrieben wurde, wird in der Praxis aus Performancegründen praktisch immer UDP verwendet. Der RPC benutzt für die Datenrepräsentation das Format XDR [Sun87], welches logisch auf der Präsentationsschicht sechs angesiedelt wird. In Abbildung 3.2 ist die NFS-Protokollschichtung nochmals zusammengefaßt.

NFS-Protokoll-Stack

Layer	Protokoll	Standard
Application	NFS, Mount	RFC 1094
Presentation	XDR	RFC 1014
Session	RPC	RFC 1050
Transport	UDP	RFC 768
Network	IP	RFC 791
Data link		IEEE 802.2
Physical		

Abbildung 3.2: Protokollschichtung bei NFS

Der RPC arbeitet exakt nach dem in Abschnitt 3.1 beschriebenen Client/Server-Prinzip. Über den RPC-Mechanismus kann ein Client Aufträge an den Server senden. Ein Auftrag setzt sich aus dem Aufruf einer vom Server angebotenen Operation mit aktuellen Parametern und dem Rücksenden des Ergebnisses zusammen. In RPC-Terminologie ist ein Netzdienst ein RPC-Programm, bestehend aus einer Menge vom Server implementierter Prozeduren (Operationen). Jedem Dienst wird durch die Datei /etc/rpc eine eindeutige Programmnummer zugeordnet. Analog besitzen die Prozeduren eines Dienstes eine eindeutige Prozedurnummer und außerdem eine Versionsnummer. Ein Auftrag eines Clients trägt im RPC-Header die Programm-, Prozedur- und Versionsnummer, um die aufzurufende Operation genau zu spezifizieren.

Ein NFS-Server implementiert 18 verschiedene Operationen für den Zugriff auf ein Dateisystem. Die wichtigsten sind das Lesen und Schreiben von Dateiblöcken und Dateiattributen sowie das Anlegen und Löschen von Dateien. Der NFS-Server arbeitet bezüglich einer Folge von Client-Aufträgen zustandslos. Das bedeutet, daß jeder Auftrag die auszuführende Lese- oder Schreiboperation komplett beschreiben muß. Es werden keine Informationen zu geöffneten Dateien oder Zeiger für die Schreib- und Leseposition innerhalb einer Datei im Server gehalten. Ein Auftrag ist daher mit dem Absenden der Antwort an den Client, auf welche

[1] Die ISO siedelt „ihren“ RPC ebenfalls auf Schicht sieben an. Der abgebildete Protokollstack für NFS zeigt eher die logische Schichtung, da die Schichten fünf bis sieben in der IP-Welt ohnehin nicht klar spezifiziert sind.

dieser synchron wartet, vollständig abgeschlossen. Die Zustandslosigkeit hat mehrere Vorteile. Nach einem Systemabsturz kann der NFS-Server einfach wieder gestartet werden, ohne daß der Zustand vor dem Absturz wiederhergestellt werden muß. Die Clients bemerken außer einem temporären Ausfall des Dienstes keine Veränderung. Weiterhin kann der zustandslose Datagrammdienst UDP mit seinem geringen Protokoll-Overhead eingesetzt werden. Hierdurch wird die maximale Datenmenge für Lese- oder Schreiboperation allerdings auf 8192 Bytes pro Auftrag begrenzt. Da die meisten NFS-Operationen idempotent sind, kann der Client bei Verlust des Anforderungs- bzw. Antwortpakets den Auftrag, sobald ein Timeout beim Warten auf die Antwort aufgetreten ist, einfach wiederholen. Für nicht idempotente Operationen wie das Löschen einer Datei unterhält der Server einen Cache für die zuletzt ausgeführten Operationen. Der Server kann Duplikate von Aufträgen daher erkennen und verwerfen.

Da NFS zustandslos arbeitet, sind Systemaufrufe wie `open()` und `close()` zum Öffnen und Schließen von Dateien auf NFS-Dateisystemen nicht sinnvoll und auch nicht implementiert. Damit Anwendungen aber transparent auf Dateien unterschiedlicher Dateisysteme zugreifen können, arbeiten die Systemaufrufe auf einer vom Filesystem unabhängigen Schicht. Die Systemaufrufe werden im Kernel vom *virtuellen Filesystem* (VFS) bearbeitet und je nach Zieldateisystem in spezifische, vom Filesystem abhängige Operationen übersetzt. Ein Leseaufruf `read()` wird bei einer lokalen Datei direkt an den Gerätetreiber weitergeleitet, bei einer Datei auf einem NFS-Dateisystem hingegen dem NFS-Client-Code übergeben, der einen entsprechenden RPC auf dem NFS-Server veranlaßt. Abbildung 3.3 verdeutlicht dieses Konzept.

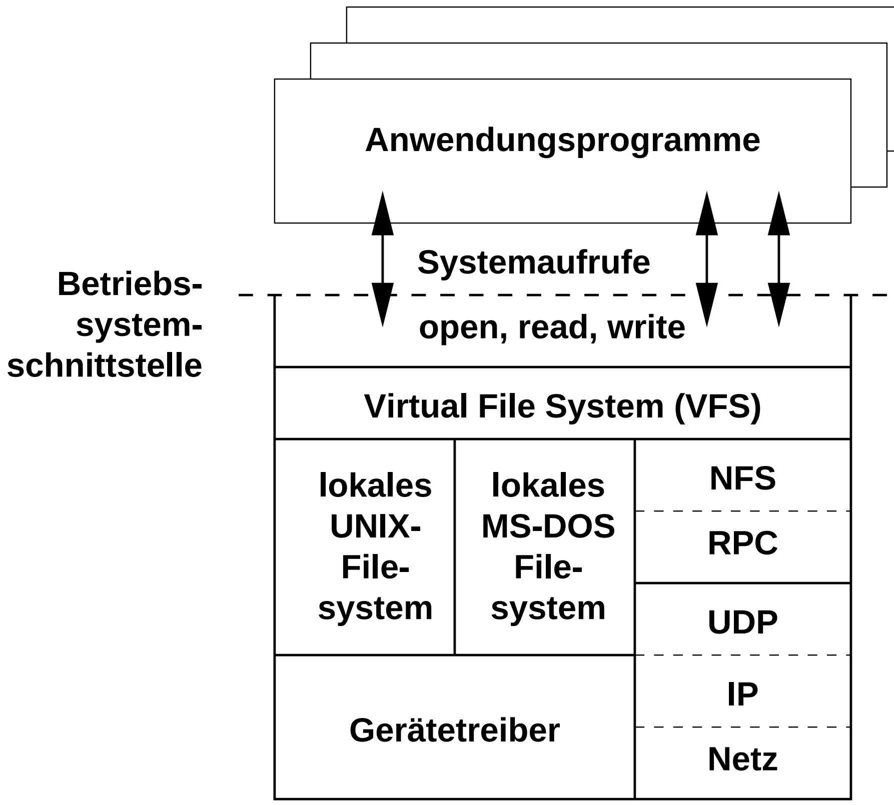


Abbildung 3.3: Integration von NFS in den Client-Kernel

3.2.2 Die Prozesse des NFS-Servers und Clients

- Server

nfsd: Ein NFS-Server wird durch den Hintergrundprozeß (Dämon) nfsd implementiert. Dieser Prozeß empfängt die Aufträge der Clients und führt die Operationen auf dem lokalen Dateisystem aus. Die Bedienzeit des Auftrags ist abhängig von der Dauer der zugrundeliegenden Dateioperation und liegt in der Regel im Bereich einiger 10ms. Da der nfsd während der Dateioperation durch das System blockiert ist, kann er Aufträge nur streng seriell bearbeiten. Um den Durchsatz des NFS-Dienstes zu steigern, werden normalerweise mehrere gleichartige Serverprozesse (nfsd) gestartet. Somit können mehrere NFS-Anforderungen gleichzeitig bearbeitet werden. Der eigentliche Code des NFS-Server befindet sich wiederum aus Performancegründen im Kernel. Die nfsd-Benutzerprozesse sind im Prinzip also nur leere Hülsen für den Scheduler, um ein Multitasking innerhalb des NFS-Dienstes zu ermöglichen. Unter AIX 4.2.0 werden mehrere NFS-Serverprozesse durch Threads innerhalb des Kernels realisiert. In diesem Fall gibt es nur noch einen NFS-Benutzerprozeß nfsd. Die Serverprozesse besitzen keine eigenen Warteschlangen für Aufträge. Alle Serverprozesse binden sich an den gleichen Socket. Trifft ein neuer Auftrag ein, wird ein schlafender Serverprozeß vom Scheduler aufgeweckt, der den Auftrag übernimmt. Sind alle NFS-Serverprozesse beschäftigt, kann eine begrenzte Anzahl von Aufträgen in dem Socketpuffer gesammelt werden.

rpc.mountd: Dieser Prozeß gehört ebenfalls zum NFS-Server. Damit ein Client auf ein von einem Server bereitgestelltes Dateisystem zugreifen kann, muß es dieses mittels eines `mount`-Kommandos in seinen lokalen Dateisystembaum einhängen. Ein `mount` auf einem NFS-Dateisystem führt zu einem RPC auf dem rpc.mountd des Servers. Dieser kontrolliert, ob der Client berechtigt ist, auf das Dateisystem des Servers zuzugreifen. Ist dies der Fall, übergibt er dem Client ein sog. *handle*, welches der Client für zukünftige Dateioperationen als Parameter benötigt. Durch diese Aktion gehen Client und Server die Dienstbeziehung ein. Von nun an kann der Client den Dienst, also ein konkretes NFS-Dateisystem nutzen. Der rpc.mountd implementiert ein eigenes RPC-Programm mit eigener RPC-Dienstnummer.

portmapper: Dieser Prozeß wird nicht nur für den NFS-Dienst, sondern für alle RPC-basierten Dienste benötigt. Es gibt i.a. keine feste Zuordnung zwischen RPC-Programmen und den Kommunikations-Ports, an denen die zugehörigen Server ihren Dienst anbieten. Damit ein Client einen Dienst nutzen kann, muß ihm aber neben der IP-Adresse des Servers auch der UDP- oder TCP-Port des Dienstes bekannt sein. Der Portmapper unterhält einen Verzeichnisdienst, der RPC-Programmnummern in Portnummern auflöst. Möchte ein Client einen RPC-basierten Dienst nutzen, sendet er eine Anfrage mit der entsprechenden Dienstnummer an den Portmapper des Serverrechners. Dieser gibt als Antwort den zugehörigen UDP- oder TCP-Port zurück, falls ein Server für den angegebenen Dienst auf diesem Rechner existiert. Hierfür muß ein Server den ihm vom Betriebssystem bei Start des Dienstes zugewiesenen Port beim Portmapper registrieren. Der Portmapper selbst bietet seinen Dienst immer an dem *well known* Port 111 (UDP/TCP) an.

- Client

biod: Beim Zugriff einer Anwendung auf eine Datei eines NFS-Dateisystems wird ein RPC-Request für den zugehörigen Server erstellt. Bei kleinen Dateioperationen wie Lesen oder Schreiben von Dateiattributen wird die Leistung der Anwendung durch das Blockieren beim Warten auf die Auftragserfüllung kaum beeinträchtigt. Anders verhält es sich, wenn große Datenmengen gelesen oder geschrieben werden müssen. Solche Dateioperationen werden unter UNIX über einen Puffer im Speicher (*file buffer cache*) abgewickelt. Die Pufferverwaltung sorgt beim Lesen durch vorzeitiges Laden (*prefetching*) von Dateiblöcken dafür, daß der Puffer immer gefüllt ist. Umgekehrt werden Schreiboperationen solange verzögert (*delayed write*), bis neuer Platz im Puffer benötigt wird. Dieser Mechanismus kann auch bei Lese- und Schreiboperationen auf NFS-Dateisystemen eingesetzt werden, wenn mehrere gleichartige biod-Prozesse auf dem System laufen. Diese Prozesse können für die Pufferverwaltung gleichzeitig mehrere Lese- oder Schreibaufträge mit maximaler Datenmenge (8 Kilobytes) an den NFS-Server stellen. Ein Client kann auch ohne biod-Prozesse betrieben werden. Dann muß aber mit einem sehr geringen Durchsatz beim Lesen und Schreiben gerechnet werden. Wie beim nfsd befindet sich der Code des biod im Kernel. Die Benutzerprozesse werden wiederum nur für das Scheduling benötigt.

- Client und Server

rpc.lockd und rpc.statd: Diese beiden Prozesse müssen sowohl auf dem Client wie auch auf dem Server vorhanden sein. Hiermit werden Dateisperren (*locks*) auf NFS-Dateisystemen realisiert. Da Dateisperren das zustandslose Arbeitsprinzip eines NFS-Servers verletzen, sind für ihre Verwaltung eigene Hintergrundprozesse nötig. Auf den komplizierten Mechanismus wird an dieser Stelle nicht weiter eingegangen.

3.3 Einführung in den Network-Information-Service (NIS)

Eines der Hauptziele bei verteilten Rechensystemen ist es, daß ein Benutzer unabhänging vom Rechner, an dem er sich anmeldet, immer die gleiche Arbeitsumgebung vorfindet. In einem heterogenen Netz von UNIX-Workstations genügt es hierfür nicht, daß der Benutzer auf jeder Maschine die gleiche Kennung besitzt. Es muß außerdem sichergestellt werden, daß er von jeder Maschine Zugriff auf andere Rechner im Netz hat und die gewohnten Dienste und Anwendungen nutzen kann.

Unter UNIX sind hierfür gemeinsame, einheitliche Konfigurationsdateien für Kennungen (/etc/passwd), Benutzergruppen (/etc/group) oder Dienste (/etc/services) erforderlich, von denen jeder Rechner im Netz eine Kopie besitzt. Die Wartung dieser Dateien stellt in einem größeren Netz ein ernstes Problem dar. Änderungen, wie das Einrichten einer neuen Benutzerkennung oder das Entfernen eines Rechners aus dem Netz, erfordern eine konsistente Modifikation der Konfigurationsdateien auf jeder Workstation im Netz. Aufgrund dieser Problematik hat Sun Microsystems, Inc. den *Network Information Service*[2] entwickelt.

[2] NIS war früher unter dem Namen „Yellow Pages“ bekannt, der aber ein Warenzeichen der *British Telecom* verletzte. Daher stammt der Präfix „yp“ der meisten NIS-Kommandos.

NIS ersetzt die Kopien der Konfigurationsdateien auf jedem Rechner durch einen Verzeichnisdienst, der die Information aus den Dateien jedem teilnehmenden Rechner zur Verfügung stellt. Für jede Konfigurationsdatei existiert eine Datenbank[3] auf einem zentralen Server. Änderungen an der Konfiguration werden ausschließlich in der Datenbank auf dem Server durchgeführt und sind anschließend sofort für jeden NIS-Client sichtbar. Das Anlegen eines neuen Benutzers erfordert somit nur einen neuen Eintrag in der Datenbank für die Datei /etc/passwd anstatt der Änderung der Paßwortdatei auf jedem Rechner. Eine detaillierte Einführung in NIS findet sich in [Ste91].

Bei NIS handelt es sich ebenfalls um einen Systemdienst nach dem Client/Server-Prinzip. Er basiert genau wie NFS auf dem RPC-Mechanismus und benutzt UDP als Transportprotokoll. Findet ein Client eine benötigte Information nicht in seiner lokalen Datei, befragt er über einen RPC-Auftrag den NIS-Server. Dieser greift auf die entsprechende Datenbank (*map*) zu und sendet das Ergebnis als Antwort. Die Verfügbarkeit des Dienstes muß außerordentlich hoch sein, da die von NIS bereitgestellte Konfigurationsinformation essentiell für den Betrieb der Clients ist. Da auf die Information mancher Maps oft zugegriffen wird, hat ein NIS-Server in einem Netz mit vielen Clients eine hohe Last zu verkraften. Aus diesen beiden Gründen werden neben dem Master-Server ein oder mehrere Slave-Server betrieben. Slave-Server besitzen Kopien der Maps des NIS-Masters und haben nur die Aufgabe, Anfragen von Clients zu beantworten. Die Verwaltung und Aktualisierung der Datenbanken wird weiterhin ausschließlich auf dem Master-Server durchgeführt. Nach Änderung einer Map auf dem Master, muß diese zur Sicherstellung der Konsistenz explizit an alle Slaves verteilt (*push*) werden. Abbildung 3.4 zeigt die Beziehungen zwischen Master, Slaves und Clients.

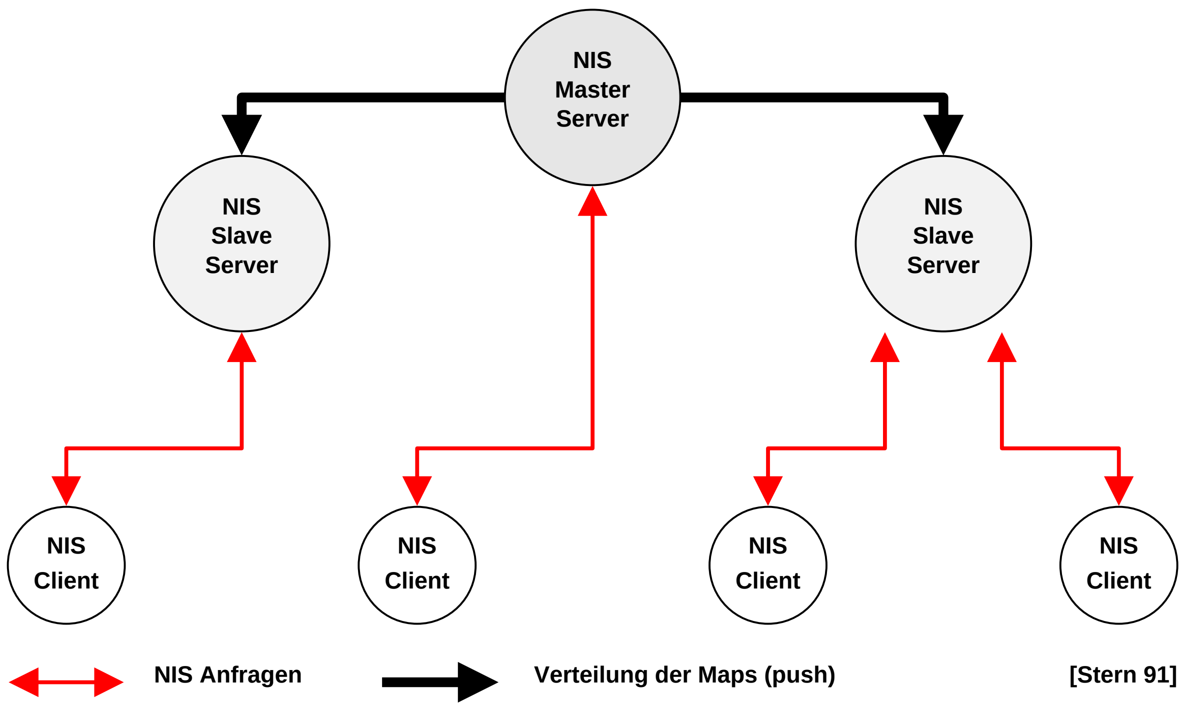


Abbildung 3.4: Master, Slaves und Clients bei NIS

[3]Diese sog. Maps sind Dateien im DBM-Format, welche die schnelle Suche von Einträgen durch einen Index erlauben.

Die Rechner eines Clusters können bezüglich der Administration und Konfiguration in unterschiedliche Domänen eingeteilt werden. Eine Domäne ist eine Menge von NIS-Maps und wird durch den Domänennamen (*domain name*) eindeutig beschrieben. Anschaulicher, wenn auch nicht ganz korrekt, ist eine Domäne eine Gruppe von Rechnern, die sich *einen* Satz NIS-Maps teilen. Das bedeutet, daß ein Client im Normalfall nur zu einer Domäne gehört, aus deren NIS-Datenbanken er alle benötigten Konfigurationsinformationen bezieht. Ein Master-Server ist „Eigentümer" der Maps einer Domäne. Da eine Verwaltung mehrerer unterschliedlicher Map-Sätze nicht vorgesehen ist, kann ein Master-Server diese Rolle nur für *eine* Domäne übernehmen. Theoretisch kann ein Slave-Server Anfragen zu mehreren Domänen bearbeiten. Dieser Fall ist aber in der Praxis eher selten. Für die weitere Betrachtung des Dienstes in dieser Arbeit werden daher folgende, die Allgemeinheit nur unwesentlich einschränkende Vereinbarungen getroffen. Master- und Slave-Server stellen den NIS-Dienst für genau *eine* Domäne zur Verfügung. Ein Client wird per Default an mindestens *eine* Domäne gebunden, von der er alle für den Betrieb des Systems benötigten Informationen beziehen kann. Spezielle Anwendungen können den Client veranlassen, eine Bindung zu einer weiteren Domäne einzugehen.

Auf dem Master liegen die Konfigurationsdateien weiterhin im ASCII-Format vor. Ein spezielles Werkzeug (`makedbm`) transformiert eine ASCII-Datei in eine Map im DBM-Format. Da diese Dateien einfach indiziert sind, kann auf sie nur über *einen* Schlüssel zugegriffen werden. Jeder Map-Eintrag ist daher ein Tupel («key», «information»). Zu einigen Dateien gibt es daher mehrere Maps, die die gleiche Information enthalten aber unterschiedlich indiziert sind. Zur Datei /etc/passwd existieren beispielsweise die Maps passwd.byname mit dem Schlüssel Benutzername und passwd.byuid mit dem Schlüssel User-ID. Einige NIS-Maps ersetzen die entsprechende lokale Datei auf dem Client völlig, andere erweitern nur die lokale Datei durch die Einträge der Map. Der Server stellt RPC-Prozeduren zum sequentiellen Durchlauf aller Einträge einer Map (`yp_get_first(), yp_get_next()`) sowie zum gezielten Suchen eines Eintrags über den Schlüssel (`yp_match()`) bereit. Hierzu gibt es auch die Kommandozeilen-Tools `ypcat` und `ypmatch`. Für Anwendungen auf dem Client ist der Einsatz von NIS transparent, da NIS die UNIX-Systemaufrufe, die normalerweise aus den Konfigurationsdateien lesen, entsprechend modifiziert.

3.3.1 Der NIS-Master-Server

Der NIS-Domänenname muß über das Kommando `domainname` gesetzt sein. Die NIS-Datenbanken werden mit Hilfe eines mitgelieferten Makefiles aus den Konfigurationsdateien im Verzeichnis /var/yp/«domainname» erstellt. Mit dem Kommando `yppush` wird die Verteilung einzelner oder aller Maps an die Slave-Server veranlaßt. Die Rechnernamen aller Slave-Server werden vom Master in der Map ypservers verwaltet. Folgende Prozesse gehören zu einem NIS-Master:

ypserv: Der zentrale Prozeß eines NIS-Servers ist ypserv. Dieser Dämon bearbeitet alle Anfragen der Clients. Die Bearbeitungszeit beträgt wenige Millisekunden. Der Server arbeitet wie bei NFS zustandslos. Das bedeutet, daß alle Aufträge in sich abgeschlossen sind. Neben den bereits erwähnten Prozeduren für den sequentiellen Durchlauf einer Map und für die Suche eines Eintrags über einen Schlüssel gibt es auch RPC-Aufrufe, die den Namen des Master-Servers einer Map oder den Zeitstempel der letzten Aktualisierung einer Map (*order number*) zurückliefern. Jede Anfrage muß den Namen der

Domäne und der Map beinhalten. Erreicht den Server eine Anfrage zu einer fremden Domäne, wird keine Antwort zurückgesendet. Betrifft die Anfrage eine nicht existente Map oder enthält eine Map nicht den gesuchten Schlüssel, wird eine Fehlermeldung an den Client zurückgesendet. Prinzipiell kann jeder NIS-Client im Netz Anfragen an einen Server senden, wenn ihm der Domainname des Servers bekannt ist. Über die Konfigurationsdatei /var/yp/securenets kann der Zugriff auf den Dienst aber auf bestimmte IP-Subnetze oder einzelne Rechner beschränkt werden. Optional wird von ypserv ein Logging fehlerhafter oder zurückgewiesener Anfragen unterstützt.

rpc.yppasswdd: Normalerweise werden Änderungen und Aktualisierungen der NIS-Datenbanken nur vom Administrator durchgeführt. Nach einer Änderung veranlaßt er die Neuerstellung der Maps und die Verteilung an die Slave-Server. Der Dämon yppasswdd ermöglicht es Benutzern, ihr Paßwort über das Kommando `yppasswd` auf dem Server zu ändern, ohne sich auf dem NIS-Master anmelden zu müssen. Der Dämon kann so konfiguriert werden, daß nach jeder Paßwortänderung die passwd-Map neu erstellt und verteilt wird.

rpc.ypupdated: Dieser für einen Master optionale Dämon veranlaßt die automatische Verteilung neu erstellter Maps an seine Slave-Server.

Portmapper: Wie bereits bei NFS erklärt, arbeiten RPC-basierte Dienste nicht mit festgelegten (*well known*) Ports. Damit ein Client auf einen NIS-Server zugreifen kann, generiert er einen Broadcast mit der RPC-Dienstnummer für NIS. Die Portmapper, bei denen ein NIS-Server registriert ist, senden eine Antwort mit der Portnummer für ypserv zurück.

3.3.2 Der NIS-Slave-Server

Bei Einrichtung eines Slave-Servers werden alle Maps vom Master übertragen und Kopien davon unter /var/yp/«domainname» angelegt. Auf dem Slave befinden sich *keine* Kopien der ASCII-Konfigurationsdateien. Der Zugriff auf die Informationen ist daher hier nur über RPC-Aufträge oder die oben erwähnten Tools möglich. Für den Betrieb eines Slave-Servers ist neben dem Portmapper nur der Serverprozeß ypserv erforderlich. Initiiert ein Master die Verteilung einer aktualisierten Map, wird ypserv auf jedem Slave durch einen RPC-Auftrag aufgefordert, einen linearen Durchlauf durch alle Einträge der Map durchzuführen, um somit die lokale Kopie der Map neu aufzubauen. Die Übertragung einer Map kann durch das Tool `ypxfr` auch von einem Slave ausgelöst werden.

3.3.3 Der NIS-Client

Für den Betrieb eines NIS-Clients ist es erforderlich, daß der Name der NIS-Domäne, auf die der Client per Default zugreifen soll, durch das Kommado `domainname` gesetzt ist. Zusätzlich muß auf jedem Client der Hintergrundprozeß ypbind gestartet werden.

ypbind: Dieser Dämon ist für die Lokalisierung von NIS-Servern für eine Domäne zuständig. Nach dem Start generiert ypbind einen Broadcast, der den Namen der Domäne enthält. Alle NIS-Server, die für diese Domäne zuständig sind, antworten auf den Broadcast. Der

Client wird an den Server gebunden, dessen Antwort als erstes eintrifft. Diese implizite Lastverteilung ist einer der Vorteile von NIS.

Benötigt ein Anwendungsprozeß auf dem Client Information aus einer NIS-Map, muß er eine RPC-Anfrage an einen Server stellen. Damit nicht jeder Anwendungsprozeß einen Server selbst lokalisieren muß, erhält er die IP-Adresse und den Port des Servers von ypbind. Diese Bindung benutzt der Prozeß für alle weiteren NIS-Anfragen.

Schlägt eine Anfrage fehl, weil der Server nicht verfügbar oder überlastet ist, wendet sich die Anwendung erneut an ypbind. Dieser versucht durch einen erneuten Broadcast, einen anderen Server zu lokalisieren.

Durch das Kommando `ypwhich` kann abgefragt werden, an welchen Server der Client derzeit gebunden ist. Die Nutzung von NIS auf einem Client kann durch Stoppen von ypbind beendet werden.

3.4 Überblick über die Object Modeling Technique (OMT)

Die *Object Modeling Technique* ist eine objektorientierte Software-Engineering-Methodologie, die unter der Leitung von James Rumbaugh bei der Firma General Electric seit 1987 entwickelt worden und 1991 veröffentlich worden ist. Da OMT auf den bekannten Entwurfstechniken wie den Entity-Relationship-, Zustands- und Datenflußdiagrammen basiert, errang sie nicht zuletzt auch wegen der Unterstützung durch zahlreiche CASE-Tools weite Verbreitung. Weitere bekannte objektorientierte Entwurfsmethoden haben Grady Booch und Ivar Jacobson entwickelt. Die Methoden, die an sich ähnliche Konzepte beschreiben, verwenden unterschiedliche Diagramme mit jeweils eigener Notation zur Modellierung. Die Stärken dieser drei Methoden werden derzeit in der *Unified Modeling Language* [Rat97] vereint.

Die OMT-Methodologie unterstützt den gesamten Software-Entwicklungsprozeß. Die unterschiedlichen Phasen sind im klassischen Wasserfallmodell in Abbildung 3.5 dargestellt.

Während der *Analyse* werden die Anforderungen an das zu erstellende System in einer Problembeschreibung formuliert. Ergebnis der Analyse sind grobe Teilmodelle für die drei essentiellen Aspekte des Systems. Das *Objektmodell* strukturiert die Objekte und ihre Relationen. Das *dynamische Modell* modelliert durch Zustände, Ereignisse und Aktionen den dynamischen Kontrollfluß innerhalb des Systems. Die funktionale Transformation der Daten wird durch das *funktionale Modell* abgebildet.

Die zweite Phase ist der *Systementwurf*, der die Gesamtarchitektur des Systems festlegt, indem Objekte des Objektmodells zu funktionalen Teilsystemen zusammengefaßt werden. Hierbei werden auch Grundsatzentscheidungen zur Realisierung und Implementierung getroffen.

Beim *Objektentwurf* werden die Grobmodelle der Analysephase erweitert und verfeinert. Ergebnis sind genaue Spezifikationen der Objektklassen, die in der folgenden Phase der *Implementierung* durch Code-Module einer bestimmten Programmiersprache und durch Wahl geeigneter Algorithmen realisiert werden. Die letzte Phase des Entwicklungsprozesses ist ein intensiver *Test* aller Teilsysteme. Natürlich werden die einzelnen Phasen nicht streng sequentiell durchlaufen. Während des Entwurfs können neue Erkenntnisse immer Rückschritte in

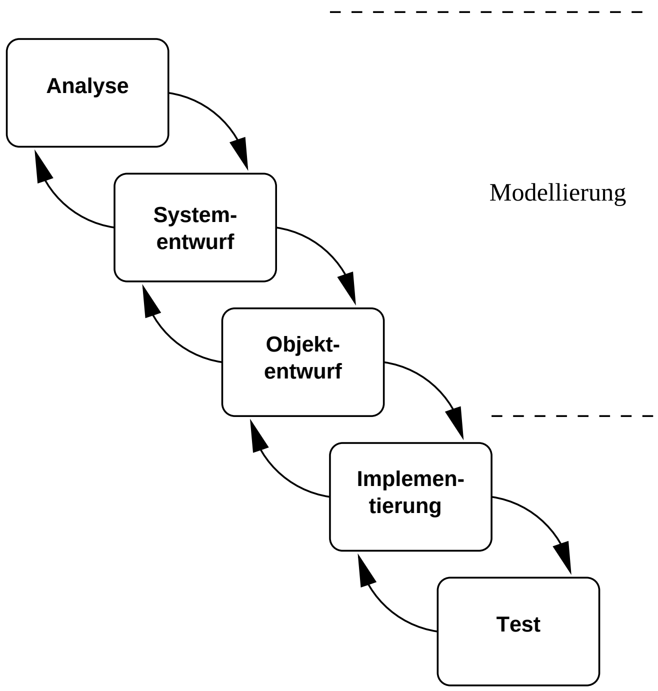


Abbildung 3.5: Der Entwurfsprozeß der Object Modeling Technique

zurückliegende Phasen erfordern, um Änderungen an den Modellen oder der Implementierung vorzunehmen.

Die einzelnen Phasen werden durch CASE-Tools für OMT unterstützt. Diese stellen diverse Diagrammeditoren für die Modellierung zur Verfügung. Aus den Diagrammen können i.a. automatisch Code-Gerüste für die Implementierung in verschiedenen objektorientierten Programmiersprachen erzeugt werden. Das in dieser Arbeit eingesetzte Werkzeug StP wird in Kapitel 6.1.1 vorgestellt. Die folgenden Abschnitte beschränken sich auf eine kurze Darstellung der Konzepte des Objektmodells und des dynamischen Modells, da diese Techniken zum Entwurf der Managementmodelle in Kapitel 4 und 5 eingesetzt werden. Für eine detaillierte Einführung zu OMT sei auf [RBP+91] und [Der95] verwiesen.

3.4.1 Objektmodell

Das Objektmodell beschreibt durch Objektdiagramme die statische Struktur der Objekte eines Systems und ihre Beziehungen zueinander. Objekte mit Zuständen und eigenem Verhalten sind gut geeignet, um den Ausschnitt der realen Welt, den das System repräsentieren soll, in einem Modell abzubilden. Ein Ziel der Modellierung ist, Objekte mit gleichen Eigenschaften und Verhalten zu einer Klasse zusammenzufassen, welche nur die für die Anwendung wichtigen Eigenschaften der Objekte enthält. Die Attribute einer Klasse spezifizieren die Datenstruktur und somit die unterscheidbaren Eigenschaften eines Objekts. Die Methoden einer Klasse sind Operationen auf den Datenstrukturen und beschreiben das Verhalten eines Objekts. Eine Klasse kann in beliebig viele gleichartige Objekte mit eigener Identität instantiiert werden. Objekte kapseln die Datenstrukturen (Attribute) und Operationen für den Zugriff (Methoden), um die interne Realisierung vor dem Benutzer zu verbergen.

Das Objektmodell ist ein Graph, dessen Knoten Objektklassen sind und dessen Kanten Relationen zwischen den Klassen darstellen. Ein Objektdiagramm ist die graphische Repräsentation des Objektmodells. Abbildung 3.6 zeigt die Notation für eine allgemeine Klasse:

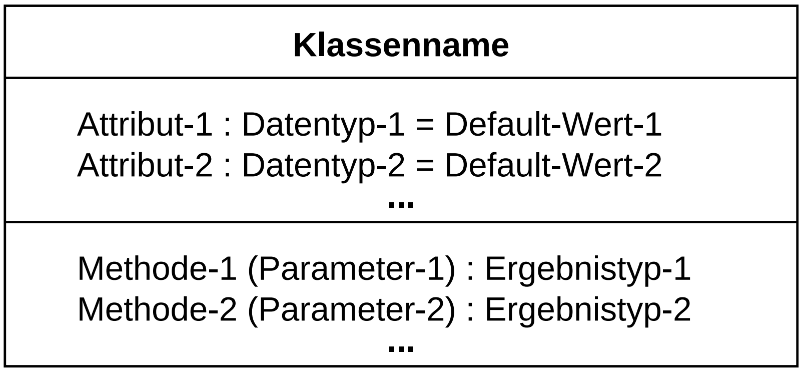


Abbildung 3.6: Notation für die Modellierung von Objektklassen

Eine Klasse ist durch Angabe des Namens, der Attribute und der Signatur der Methoden bestimmt. Namen sind innerhalb einer Klasse eindeutig, das heißt, verschiedene Klassen können Attribute und Methoden mit gleichem Namen besitzen. Attribute sind Datentypen mit einem festgelegten Wertebereich. Der Zustand einer Objektinstanz wird durch die aktuelle Belegung der Attribute mit Datenwerten ausgedrückt. Die Angabe des Typs und eines Default-Werts, welcher dem Attribut bei Instantiierung des Objekts zugewiesen wird, sind im Objektdiagramm optional.

Eine Operation ist eine Funktion, die von einem Objekt auf einem Zielobjekt angewendet werden kann. Die Operation wird von einer Methode des Zielobjekts implementiert. Durch Aufruf einer Methode kann der Zustand einer Objektinstanz verändert werden. Kann die gleiche Operation auf verschiedenen Klassen angewendet werden, ist sie polymorph. Die zugehörigen Methoden haben die gleiche semantische Bedeutung, sie sind aber in der Regel unterschiedlich implementiert. Die Methoden polymorpher Operationen sollten in jedem Fall eine einheitliche Signatur haben. Die Signatur legt den Operationsnamen, Anzahl und Typ der Parameter und den Ergebnistyp fest. Die Angabe der Parameter und des Ergebnistyps für die Methoden im Objektdiagramm ist wiederum optional.

Die Identifikation geeigneter Objektklassen aus der Problembeschreibung der Analyse ist ein wichtiges Ziel der Modellierung. Das OMT-Objektmodell betrachtet aber auch die Beziehungen zwischen den Objektklassen. OMT kennt die drei unterschiedlichen Beziehungen *Assoziation*, *Generalisierung* und *Aggregation*.

Assoziation: Assoziationen zwischen Objektklassen beschreiben Relationen zwischen Objektinstanzen. In Abbildung 3.7 zeigt ein Beispiel für die Assoziation «*läuft auf*», dargestellt durch die Linie zwischen der Klasse **UNIX_System** und der Klasse **UNIX_Process**.

Namen für Assoziationen werden meist aus Verben gebildet. Obwohl durch die Namen die Beziehung in der Regel nur in eine Richtung ausgedrückt wird, sind Assoziationen inhärent bidirektional. Die Umkehrung der Beziehung in obigem Beispiel könnte «*führt aus*» heißen. Assoziationsnamen sind optional.

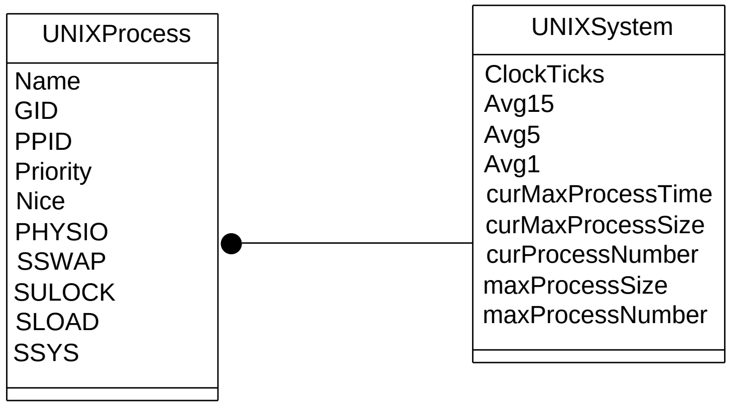


Abbildung 3.7: Beispiel für eine Assoziation

Neben den binären Assoziationen sind auch Assoziationen zwischen drei (ternär) oder mehreren (n-är) Klassen möglich. Dies wird im Diagramm durch ein Rautensymbol ausgedrückt, welches über Linien die in Beziehung stehenden Klassen verbindet. In der Praxis sind höhere Ordnungen als ternär aber selten, da diese Relationen schwieriger zu verstehen und zu implementieren sind. Assoziationen werden in der Regel durch Zeiger auf verknüpfte Objekte implementiert.

Ferner gibt es Symbole zum Bezeichnen der Multiplizität. Die Multiplizität spezifiziert, wieviele Instanzen einer Klasse mit *einer* Instanz der anderen Klasse verknüpft sein können. Möglich sind 1:1, 1:n, n:1 und n:m-Assoziationen. Für „n“ bzw. „m“ können auch feste Werte oder fixe Intervalle definiert sein. Ein ausgefüllter, schwarzer Punkt am Linienende steht für „n“, der Zusatz „1+“ bedeutet $n >= 1$. Ein transparenter Punkt steht für den Multiplizitätswert null oder eins. Fehlen Symbole oder Zahlen- bzw. Intervallangaben, handelt es sich um eine 1:1-Assoziation. In obigem Beispiel liegt eine n:1-Assoziation vor. *Mehrere* Prozesse laufen auf genau *einem* System.

Besitzt eine Assoziation eigene Attribute, so ist es sinnvoll, die Beziehung durch eine sog. Beziehungsklasse zu modellieren. Eine neue Verknüpfung zwischen zwei Objekten führt dann zu einer neuen Instanz der Beziehungsklasse. Abbildung 3.8 zeigt die Beziehung «*exportiert*» zwischen der Klasse **NFS_Server** und der Klasse **Filesystem**. Die Optionen für den Export werden durch die Attribute der Beziehungsklasse **ExportOptions** modelliert.

Generalisierung: Über die Generalisierung wird das bekannte objektorientierte Prinzip der Vererbung modelliert. Hierdurch können ähnliche Klassen gleiche Attributen bzw. Methoden aus einer gemeinsamen Oberklasse erben. Individuelle Merkmale werden in der Unterklasse durch Hinzufügen von Attributen und Methoden realisiert. Die Generalisierung („is-a“-Relation) ist also eine Beziehung zwischen einer Oberklasse und einer oder mehreren verfeinerten Unterklassen. Die OMT-Notation für die Generalisierung ist ein Dreieck, welches die Oberklasse mit ihren Unterklassen verbindet. Im Beispiel in Abbildung 3.9 erbt die Unterklasse **UNIX_Process** die Attribute und Methoden der allgemeineren Klasse **capsule**, welche sie durch eigene Attribute verfeinert.

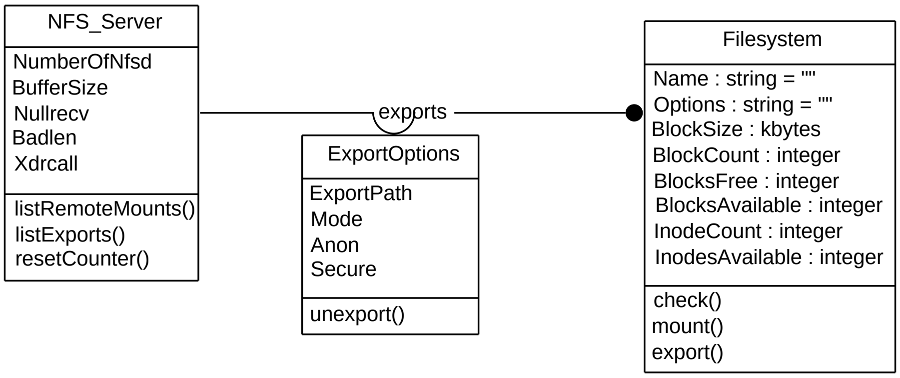


Abbildung 3.8: Beispiel für die Modellierung einer Assoziation als Klasse

Generalisierung und Vererbung sind transitiv. Unterklassen dürfen geerbte Methoden für Operationen überschreiben, d.h. eine eigene Implementierung für die Methode bereitstellen. Auch der Default-Wert eines geerbten Attributs kann in einer Unterklasse überschrieben werden. OMT geht allerdings von strikter Vererbung aus. Das bedeutet, daß Unterklassen geerbte Attribute oder Methoden nicht weglassen dürfen. Ferner sollte der Datentyp von Attributen sowie die Signatur von Methoden nicht verändert werden.

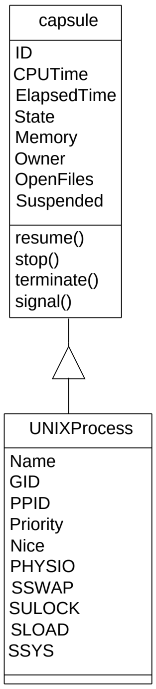


Abbildung 3.9: Beispiel für eine Generalisierung

Eine Einschränkung bzw. Erweiterung des Wertebereichs für ein Attribut muß abhängig vom Anwendungsfall vorsichtig vorgenommen werden.

Aggregation: Die Aggregation ist eine Spezialform der Assoziation. Objekte können aus anderen Objekten zusammengesetzt sein. Umgekehrt kann ein Objekt ein Teil oder eine Komponente eines anderen sein. Diese „part-of"-Beziehung oder Enthaltenseinsrelation wird *Aggregation* genannt. Wichtige Eigenschaften der Assoziation sind Transitivität und Antisymmetrie. Eine Aggregation liegt insbesondere dann vor, wenn Operationen auf das Ganze automatisch auch auf die Teile angewendet werden sollen. Dargestellt wird die Aggregation durch eine Assoziation, wobei die Klasse mit einer kleinen Raute gekennzeichnet ist, welche die Instanzen der verknüpften Klassen enthält. Auf der Seite der Komponentenklassen ist wieder die Angabe der Multiplizität (siehe oben) erforderlich. Das Beispiel in Abbildung 3.10 zeigt, daß sich eine Instanz der Klasse **Workstation** aus mehreren Instanzen der Klasse **Device** zusammensetzt.

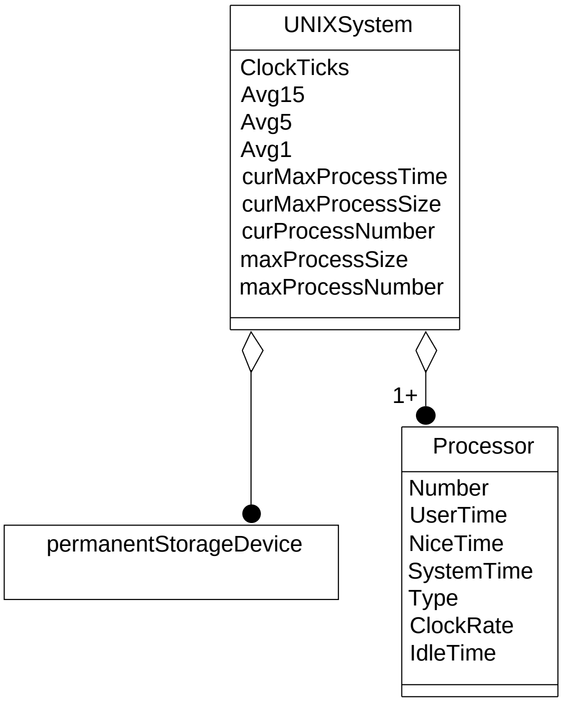


Abbildung 3.10: Beispiel für die Aggregation

Aus den Informationen des Objektmodells können Code-Gerüste zur Implementierung der Klassen in verschiedenen objektorientierten Programmiersprachen, wie z.B. C++, Java, Smalltalk etc., generiert werden.

3.4.2 Dynamisches Modell

Das Objektmodell teilt ein System in statische Objektklassen auf, die in bestimmten Beziehungen zueinander stehen. Es beschreibt, *welche* Attribute und Operationen ein Objekt besitzt, aber nicht, *wann* sich Attributwerte ändern oder in welcher Reihenfolge Operationen auf Objekten ausgeführt werden. Das *dynamische Modell* dient dazu, Interaktionen zwischen Objekten und die Ausführung von Operationen zeitlich anzuordnen. Weiterhin kann der Kontrollfluß in einem System mit mehreren, gleichzeitig aktiven Objekten gesteuert werden.

Zur dynamischen Modellierung kennt OMT mehrere Diagrammarten. Das wichtigste ist aber das Zustandsdiagramm (*state chart*), welches die Zustände eines Objekts und seine Reaktion auf Ereignisse in einer Notation beschreibt, die der Darstellung endlicher Zustandsmaschinen in der Automatentheorie ähnlich ist. Die Ereignisabfolge für ein bestimmtes Szenario kann durch einen Ereignispfad (*event trace*) beschrieben werden. Bei der Identifikation möglicher Ereignisse, die eine Gruppe von Objektklassen betreffen, helfen Ereignisflußdiagramme (*event flow*) und sog. *Use Cases*. An dieser Stelle werden nur die Zustandsdiagramme vorgestellt. Für eine Einführung in die anderen Diagramme wird nochmals auf die Literatur verwiesen.

Der *Zustand* eines Objekts ist durch seine Attributwerte und durch die Verknüpfungen zu anderen Objekten bestimmt. Das Auftreten eines *Ereignisses* kann den Zustand eines Objekts verändern. Ein Ereignis ist ein externer „Reiz“ auf ein Objekt und kann z.B. der Aufruf einer Methode sein. Ereignisse gehen immer von einem Quellobjekt aus und betreffen ein Zielobjekt. Die Reaktion auf ein Ereignis ist vom Zustand des Objekts abhängig. Möglich sind Zustandsänderungen, das Ausführen von Aktionen sowie das Senden eines neuen Ereignisses an das Quellobjekt oder an ein drittes Objekt. Ereignisse treten zu einem bestimmten Zeitpunkt auf und besitzen keine Dauer. Wenn zwischen Ereignissen kausale Zusammenhänge bestehen, können diese über die „nach“-Relation angeordnet werden. Voneinander unabhängige Ereignisse treten in beliebiger Reihenfolge parallel auf. Die Zeitspanne zwischen zwei Ereignissen, die ein Objekt betreffen, entspricht einem Zustand. Ein Zustandsdiagramm stellt Zustände und Ereignisse für eine Klasse dar. Das dynamische Modell beinhaltet alle Zustandsdiagramme, um die Aktivitäten und Interaktionen des gesamten Systems beschreiben zu können.

Eine Zustandsänderung, die durch ein Ereignis ausgelöst wird, heißt *Transition*. Das Zustandsdiagramm ist ein gerichteter Graph, dessen Knoten Zustände bezeichnen. Die Kanten sind die Transitionen und werden mit dem Namen des zugehörigen Ereignisses beschriftet. Abbildung 3.11 zeigt hierfür die OMT-Notation:

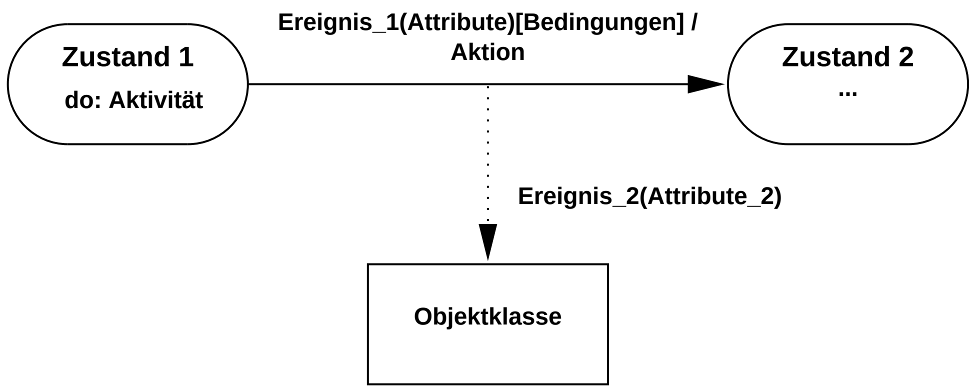


Abbildung 3.11: Die Notation für Zustandsdiagramme

Das Auftreten von «Ereignis_1» führt zu einem Übergang von «Zustand 1» nach «Zustand 2». Ereignisse können Attribute besitzen. In diesem Fall signalisieren sie dem Objekt nicht nur, daß etwas geschehen ist, sondern übermitteln durch die Attribute zusätzlich Datenwerte. Ist für einen Zustand keine Transition für ein bestimmtes Ereignis definiert, wird dieses bei Auftreten in dem Zustand ignoriert. Für Transitionen können Bedingungen durch eine boolesche Funktion formuliert werden. Falls vorhanden, findet der Zustandsübergang bei Auftreten des Ereignisses nur statt, wenn die Bedingungen erfüllt sind. Während eines Zustands kann ein Objekt eine bestimmte *Aktivität* ausführen. Ein Beispiel hierfür ist das Warten auf einen Tastendruck. Während eine Aktivität eine Zeitdauer besitzt, ist eine *Aktion* eine momentane Operation ohne Zeitdauer. Aktionen können bei einem Zustandsübergang ausgeführt werden. Beispiele für Aktionen sind interne Steueroperationen wie das Inkrementieren eines Zählers sowie das Generieren eines neuen Ereignisses. In Abbildung 3.11 generiert die Transition das Ereignis «Ereignis_2» für eine Objektklasse.

Zustandsdiagramme können zur Reduzierung der Komplexität verschachtelt werden. Ein Oberzustand wird dabei in mehrere detailliertere Unterzustände verfeinert, die in einem eigenem Diagramm dargestellt werden. Ein Objekt in einem Oberzustand befindet sich daher in genau einem Unterzustand im verschachtelten Diagramm. Weiterhin wird auch die Modellierung von Parallelität innerhalb eines Objekts unterstützt. Auch Ereignisse können in einer Generalisierungshierarchie angeordnet werden. Untereignisse erben dabei Attribute von Oberereignissen. Das Auftreten eines speziellen Ereignisses löst Transitionen für jedes Vorfahrenereignis aus.

Das dynamische Modell erleichtert die Implementierung der Objektklassen, da die Interaktionen der Menge paralleler Objekte eines Systems graphisch dargestellt werden könnnen. Eine Codegenerierung auf Basis des dynamischen Modells ist allerdings nicht vorgesehen.

3.5 Das Referenzmodell für Open Distributed Processing (RM-ODP)

3.5.1 Einführung

Seit langem existieren Standards und Architekturen, um Rechner miteinander zu vernetzen und Anwendern durch Kommunikationsverbindungen die Nutzung von Diensten auf entfernten Rechnern zu ermöglichen. Das wichtigste Rahmenwerk hierfür ist das OSI-Referenzmodell. Zu den einzelnen Schichten dieses Modells gibt es wiederum diverse Standards wie die IEEE-Normen 802.x für die Schicht 2 und IP für Schicht 3.

Bisher fehlte aber ein Standard, der die Entwicklung von Anwendungskomponenten erlaubt, die in offener, heterogener und verteilter Systemumgebung interagieren können. Die ISO begann 1987 mit der Arbeit an einem entsprechenden Modell und schloß sich 1991 mit der ITU-T zusammen, um eine „standardisierte logische Architektur für den Entwurf von objektorientierten verteilten Systemen in heterogener Umgebung“ [Far97] zu entwickeln. Ergebnis ist das *Reference Model Of Open Distributed Computing* (RM-ODP). Der Standard ISO 10746 aus dem Jahr 1995 besteht aus vier Teilen [ISO95b, ISO95c, ISO95d, ISO95e]. Tutorien zu diesem Modell sind u.a. [Ban96, Joy, Lin95, Ray95] und [Far97], an welchem sich diese Einführung orientiert.

Die Offenheit des ODP bedeutet, daß Programmierschnittstellen (APIs) und Kommunikationsprotokolle offengelegt und standardisiert sein müssen. Die Unterstützung von heterogenen, verteilten Systemen durch das Modell bezieht sich auf Rechner und Netzkomponenten verschiedener Hersteller, unterschiedliche Betriebssysteme und Kommunikationsprotokolle sowie den Einsatz unterschiedlicher Programmier- und Datenbanksprachen für die verteilten Anwendungen.

RM-ODP adressiert die Probleme, die sich bei der Organisation von Anwendungskomponenten ergeben. Fragestellungen sind hierbei z.B. „*Welche* Objekte erbringen einen benötigten Dienst und *wo* befinden sie sich?“, „*Wie* kann der Dienst genutzt werden?“, „*Wie* können Dienste organisiert werden, damit der Benutzer den Eindruck gewinnt, es handelt sich um eine einzige integrierte verteilte Anwendung?“ und „*Welche* Unterstützung benötigen Anwendungskomponenten von der darunterliegenden verteilten Systemplattform?“.

RM-ODP ist ein Meta-Modell, welches durch Konzepte, Regeln und die Defintion von Infrastrukturobjekten eine Architektur für verteilte Anwendungen in heterogener Systemumgebung beschreibt. Ziel ist es, beim Entwurf eines verteilten Systems die *Komponenten* (Objekte) identifizieren zu können, die an fest spezifizierten und typisierten *Schnittstellen* Dienste anbieten. Diese können durch Interaktionen an den Schnittstellen d.h. durch Kommunikation auf Anwendungsebene genutzt werden. Zwei weitere wichtige Aspekte sind *Portabilität* und *Transparenz*. Damit Anwendungskomponenten portabel sind, bedarf es standardisierter verteilter Plattformen. Weiterhin sollen Details der Verteilung vor den Komponenten verborgen bleiben. Hierzu sind Mechanismen erforderlich, die verschiedene Transparenzen (*access, concurrency, location, migration, replication, failure, etc.*) bereitstellen.

Aufgrund der Komplexität ist ein Entwurf eines verteilten Systems mittels einer einzelnen Spezifikation schwierig, wenn nicht sogar unmöglich. Deshalb führt das RM-ODP unterschiedliche Sichten (*Viewpoints*) auf das System ein, um die Komplexität nach dem Prinzip «divide and conquer» zu reduzieren. Diese spiegeln die Sichten wieder, die verschiedene Personen auf das System haben. Auftraggeber eines Unternehmens definieren die Anforderungen an das System. Software-Architekten erstellen hieraus einen Entwurf. Programmierer implementieren den Entwurf. Techniker installieren das fertige System. Die Viewpoints sind daher Projektionen auf unterschiedliche Aspekte des Systems. Um Redundanz zu vermeiden, soll die Zerlegung durch die Viewpoints möglichst disjunkt sein. *Viewpoint Languages* legen das Vokabular und die Grammatik fest, die zur Beschreibung der Information eines Viewpoints eingesetzt werden. Jeder Viewpoint enthält Konzepte, die das Vokabular der zugehörigen Sprache darstellen, und Regeln (*structuring rules*), die die Grammatik der Sprache bilden. Welche konkrete Sprache schließlich als Viewpoint Language eingesetzt wird, läßt das Modell offen. Voraussetzung ist lediglich, daß die Syntax und Semantik der konkreten Sprache die Konzepte und Regeln der Viewpoint Language ausdrücken können. RM-ODP definiert folgende fünf Viewpoints:

Enterprise Viewpoint: Hier wird die Gesamtumgebung für das System und sein Zweck beschrieben. Außerdem werden die Anforderungen (*requirements*) an das System, zu erfüllende Bedingungen (*constraints*), ausführbare Aktionen (*actions*) und DV-Zielvorgaben (*policies*) aus Unternehmenssicht definiert.

Information Viewpoint: Dieser Viewpoint legt die Struktur und Semantik der Informationen des Systems fest. Weitere Punkte sind die Definiton von Quellen und Senken von Information sowie die Verarbeitung und Transformation von Information durch das

System. Hierzu gibt es Integritätsregeln und Invarianten. Beispiele für *Information Languages* sind OMT und OSI GDMO.

Computational Viewpoint: Hier wird ein System in logische, funktionale Komponenten zerlegt, die für die Verteilung geeignet sind. Das Ergebnis sind Objekte, die Schnittstellen besitzen, an denen sie Dienste anbieten bzw. nutzen. Die meisten objektorientierten Sprachen, wie z.B. C++, IDL, Java und Smalltalk, eignen sich als *Computational Language*.

Engineering Viewpoint: Dieser Viewpoint beschreibt die erforderliche Systemunterstützung, um eine Verteilung der Objekte aus dem Computational Viewpoint zu erlauben. Hierzu werden generische Infrastrukturobjekte eingeführt, die die oben genannten Verteilungstransparenzen realisieren um eine Kommunikation der verteilten Objekte zu ermöglichen. So entsteht das Modell einer generischen, objektorientierten, verteilten Plattform.

Technology Viewpoint: Dieser Punkt beschreibt die Wahl konkreter Technologien zur Implementierung und Realisierung des Systems. Hierin enthalten ist die Spezifikation der Hardware (Hersteller und Modell der Rechner, Netzkomponenten, etc.) als auch der Software (Betriebssystem, Kommunikationsprotokolle und Programmiersprachen). Da die Implementierung eines ODP-Systems auf unterschiedlichen technischen Plattformen möglich sein soll, sollten für diesen Viewpoint keine Abhängigkeiten zu den übrigen bestehen.

Diese Arbeit beschränkt sich auf die Untersuchung des Computational und Engineering Viewpoints, da deren Konzepte gerade die Objekte beschreiben, die für das System- und Anwendungsmanagement relevant sind. Die nächsten beiden Abschnitte beschreiben daher die Konzepte dieser Viewpoints näher, aus denen in Kapitel 4.2 generische Managementobjektklassen abgeleitet werden.

3.5.2 Computational Viewpoint

Der Computational Viewpoint ist eine funktionale Zerlegung einer ODP-Anwendung in Komponenten, die für die Verteilung geeignet sind. In dieser Sicht besteht ein System aus sog. *Computational Objects*, die über fest definierte Schnittstellen (*Computational Interfaces*) interagieren. Die Objekte und Schnittstellen werden wiederum durch Vorlagen (*Templates*) spezifiziert. Im Computational Modell findet die Interaktion der Komponenten transparent von der Verteilung statt. Mechanismen und Konzepte zur Realisierung dieser Verteilungstransparenz definiert der Engineering Viewpoint.

Beispielsweise besteht der verteilte Systemdienst NFS aus den Computational Objects NFS-Server und NFS-Client. Der Client kann hierbei über die NFS-Dienstschnittstelle auf die vom Server exportierten Verzeichnisse und Dateien zugreifen. Weiterhin besitzt der Server eine Management-Schnittstelle, über die eine Managementanwendung zum Beispiel den Status des Servers überwachen kann.

Computational Object

Ein Computational Object ist eine Komponente eines verteilten Systemdienstes oder einer verteilten Anwendung. Die Komponenten interagieren über Schnittstellen. Ein Objekt kann eine oder mehrere Schnittstellen besitzen und nimmt bezüglich der Schnittstellen unterschiedliche Rollen ein. Mögliche Rollen sind zum Beispiel *Client* bzw. *Server* oder *managed* bzw. *managing*. Nach dem RM-ODP kann ein Computational Object u.a. folgende Aktivitäten ausüben:

- Objekte und Schnittstellen erzeugen.
- Objekte und Schnittstellen löschen.
- Bindungen zu Schnittstellen herstellen.
- Seinen Zustand lesen und verändern.
- Operationen bzw. Methoden eines anderen CO aufrufen.

Computational Objects werden beim Start aus ihrem zugehörigen *Computational Object Template* (siehe Seite 45) instantiiert. Die Abbildung 3.12 verdeutlicht nochmals die Zusammenhänge.

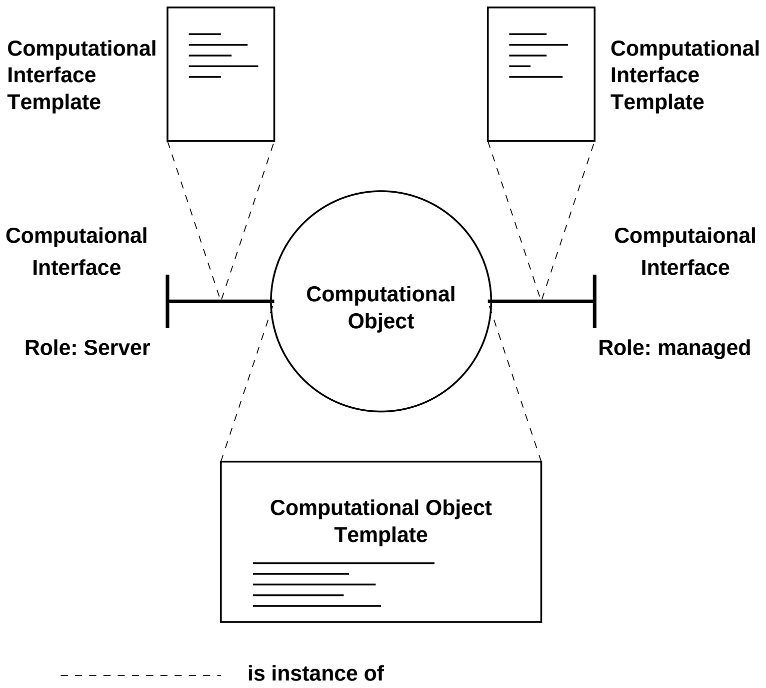


Abbildung 3.12: Modell des Computational Object

Computational Interface

Die Computational Interfaces stellen im RM-ODP die Punkte für das Bereitstellen von Diensten dar. Über diese Schnittstellen können die Computational Objects interagieren. Das

RM-ODP kennt drei verschiedene Formen von Interaktionen:

signals: Elementare atomare Interaktionen

operations: Prozedur- oder Methodenaufrufe

flows: Abstraktionen kontinuierlicher Datenflüsse (z.B. Audio- oder Videoströme)

Dementsprechend definiert das Modell auch drei verschiedene Arten von Computational Interfaces nämlich *Signal Interface*, *Operation Interface* und *Stream Interface*.

Alle Interaktionen, die an einer Signalschnittstelle auftreten können, sind Signale. Ein Signal könnte z.B. eine Meldung über ein Ereignis (*event*) oder eine Unterbrechungsaufforderung (*Interrupt*) sein, die ein CO an ein anderes sendet. Signale haben einen Namen und können Parameter besitzen. Sie führen zu einer atomaren, d.h. ununterbrechbaren Kommunikation in einer Richtung zwischen genau zwei Objekten, dem Initiator und dem Empfänger (*responder*). Es handelt sich also um einen eher primitiven Schnittstellentyp. Eine Signalschnittstelle kann für eine Menge von Signalen entweder die Rolle Initiator oder Empfänger einnehmen.

Beim Operation Interface sind die Interaktionen Operationen im Sinne des Client/Server-Modells. Ein Client ruft eine Prozedur bzw. eine Methode eines Servers auf. ODP unterscheidet zwei Typen von Operationen. Gibt der Server auf eine Anforderung (*request*) eines Clients eine Antwort (*termination*) zurück, wird die Operation *interrogation* genannt. Bei Aufrufen ohne Antwort handelt es sich um ein sog. *announcement*. Unter dem Begriff *termination* werden Ergebniswerte aber auch möglicherweise auftretende Ausnahmen zusammengefaßt. Als Rolle für die von der Schnittstelle unterstützte Menge von Operationen kommt natürlich nur Client oder Server in Frage.

Datenströme wie Audio- und Videoflüsse oder Dateiübertragungen werden über Stream Interfaces abgewickelt. Die exakte Semantik von Strömen wird im Computational Modell nicht spezifiziert, da sie sehr stark vom Typ der verteilten Anwendung abhängt. Beispielsweise kann eine Transaktion innerhalb eines Datenbanksystems als Folge von Einzeloperationen auf der Datenbank ebenfalls als *flow* interpretiert werden. Stream Interfaces können die Rollen Erzeuger oder Verbraucher besitzen.

Bindungen

Interaktionen zwischen gegebenen Schnittstellen von Computational Objects sind nur möglich, wenn eine Bindung (*binding*), also eine Art Kommunikationspfad zwischen ihnen hergestellt wurde. Für die Etablierung einer Bindung ist es erforderlich, daß die beiden beteiligten Schnittstellen vom gleichen Typ sind. Das Bindungsmodell des RM-ODP unterscheidet explizites und implizites Binden. Beim expliziten Binden muß der Kommunikationspfad explizit aufgebaut werden. Beispiel hierfür ist eine Telnet-Sitzung. Das CO «Telnet-Client» initiiert eine Bindung zum CO «Telnet-Server». Implizites Binden ist nur auf Operation Interfaces definiert. Ein entfernter Methodenaufruf zwischen zwei verteilten Objekten ist ein Beispiel für eine implizite Bindung. Auch unter CORBA sind Bindungen implizit. Hierzu muß aber zur Übersetzungszeit des Clients die Schnittstellenbeschreibung (IDL) des Serverobjekts zur Verfügung stehen, damit der IDL-Compiler statische Bindungen herstellen kann. Eine Ausnahme hiervon sind Bindungen die über das *Dynamic Invocation Interface* (DII) des ORB etabliert werden.

Bisher wurden nur primitive Bindungen, d.h. Bindungen zwischen genau zwei Schnittstellen betrachtet. Komplexe Bindungen (*compound binding*) zwischen mehreren Schnittstellen werden über ein *Binding Object* realisiert. Dieses Objekt koppelt die Schnittstellen mehrerer COs durch primitive Bindungen aneinander. Das Binding Object besitzt eine Kontrollschnittstelle, über welche Bindungen hinzugefügt, verändert (z.B. bezüglich QoS), gelöscht und überwacht werden können. Ein Konferenzsystem ist ein Beispiel, wo die Stream Interfaces mehrerer Verbraucher (Zuhörer) über ein Binding Object an das des Erzeugers (Sprecher) gebunden sind.

Computational Interface Template

Im RM-ODP wird ein Computational Interface durch Angabe der Signatur (*signature*), des Verhaltens (*behaviour*) und der Anforderungen an die Umgebung (*environment contract*) charakterisiert. Diese Informationen werden in einem Computational Interface Template zusammengefaßt, welches eine Instantiierung der Schnittstelle zur Laufzeit ermöglicht.

Ganz allgemein bestimmt die Signatur die möglichen Interaktionen an einer Schnittstelle. Sie beschreibt, auf welche Weise ein an der Schnittstelle angebotener Dienst genutzt werden kann. Abhängig von der Art der Schnittstelle enthält die Signatur eine Menge von Interaktionsbeschreibungen (*action templates*). Bei einem Operation Interface besteht die Signatur daher aus einer Menge von Beschreibungen der Operationen, die von der Schnittstelle unterstützt werden. Eine Operationsbeschreibung setzt sich aus dem Operationsnamen, der Anzahl der Parameter und gegebenenfalls der Angabe des Ergebnistyps zusammen. Für jeden Parameter muß Name und Typ spezifiziert sein.

Außerdem enthält das Template die Information, welche Rolle die instantiierte Schnittstelle einnimmt. Das Verhalten, welches vom internen Zustand des Objekts abhängig ist, legt die erlaubte Reihenfolge von Interaktionen an der Schnittstelle fest. Für eine Schnittstelle können im *environment contract* Anforderungen an die Dienstgüte (QoS) der Umgebung des Objekts definiert werden. Werden diese erfüllt, kann das Objekt seinerseits eine bestimmte Dienstgüte an der Schnittstelle garantieren.

In einer CORBA-Umgebung stellt die IDL-Spezifikation das Computational Interface Template eines Operation Interface dar. Sie enthält die Signaturen der Operationen. Eine IDL-Spezifikation besitzt immer die Rolle Server. Damit aber ein Objekt Operationen bzw. Methoden eines Server-Objekts aufrufen kann, muß die Schnittstellenbeschreibung des Servers zur Übersetzungszeit vorhanden sein oder zur Laufzeit das Interface Repository des ORB befragt werden, um einen Methodenaufruf über das DII absetzen zu können.

IDL besitzt keine formalen Konzepte, um das Verhalten eines Objekts und die Anforderungen an die Umgebung auszudrücken. Diese beiden Punkte werden aber ausdrücklich vom RM-ODP für eine Schnittstellenbeschreibung gefordert. Soll IDL als Computational Viewpoint Language eingesetzt werden, muß man hierfür auf zusätzliche Kommentare in der IDL-Spezifikation zurückgreifen.

Computational Object Template

Das *Computational Object Template* enthält alle Informationen, die zur Instantiierung eines Objekts zur Laufzeit erforderlich sind. Es umfaßt hierzu die Computational Interface Templates aller Schnittstellen, die das Objekt kreieren kann. Zusätzlich sollen eine Spezifikation des Objektverhaltens und seine Anforderungen an die Umgebung vorhanden sein. Unter dem Verhalten eines Objekts versteht man die Menge aller möglichen Zustandsänderungen, die das Objekt aufgrund von Aktionen und Ereignissen erfahren kann. Zur Modellierung des Verhaltens bietet OMT dynamische Modelle mit Zustandsdiagrammen und *Use Cases* an.

3.5.3 Engineering Viewpoint

Der Engineering Viewpoint im RM-ODP beschäftigt sich mit den Konzepten, die für eine verteilte Plattform erforderlich sind, um den im Computational Model spezifizierten verteilten Anwendungen eine Systemunterstützung bieten zu können. Er legt das Augenmerk auf die Frage, wie die Objektinteraktionen aus dem Computational Model erreicht werden können und welche Ressourcen dafür gebraucht werden. Hierzu werden Infrastrukturobjekte bereitgestellt, die Mechanismen für die Kommunikation zwischen Objekten und für die Unterstützung von Orts- und Verteilungstransparenz realisieren. Es gibt aber keine Beschränkung auf bestimmte Maschinenarchitekturen wie z.B. RISC, Betriebssysteme (UNIX) oder Kommunikationsprotokolle (TCP/IP). Vielmehr handelt es sich um ein maschinenunabhängiges Modell eines erweiterten verteilten Betriebssystems für vernetzte Rechner.

Im folgenden werden zunächst die Infrastrukturobjekte, die der Engineering Viewpoint beschreibt, vorgestellt.

Die verteilte Plattform des ODP Engineering Model besteht aus einer Anzahl vernetzter autonomer Rechner, den sog. *Nodes*. Der *Nucleus* kontrolliert die Rechen-, Speicher- und Kommunikationsressourcen eines Nodes und kann daher mit dem Betriebssystem eines Rechners gleichgesetzt werden. Ein Node beherbergt genau einen Nucleus und mehrere *Capsules*. Mit Capsule wird das Konzept des traditionellen geschützten Prozesses in einem Rechner beschrieben. Ein Capsule besitzt seinen eigenen Adressraum und nutzt Teile der Speicher- und Rechenressourcen, die der Nucleus verwaltet. Der Adressraum des Capsule wird durch das Betriebssystem geschützt. Außerdem stellt es die kleinste Einheit für unabhängiges Fehlverhalten dar. Ein Capsule wiederum setzt sich aus ein oder mehreren *Clusters* von *Basic Engineering Objects* zusammen. Die Objekte des Computational Model werden zur Laufzeit durch Basic Engineering Objects realisiert. Die Vorstellung für einen Cluster ist ein Segment des virtuellen Speichers, welches Objekte einer Anwendung enthält. Ein Basic Engineering Object ist ein Modul eines Programms, welches nicht isoliert ausgeführt werden kann. Ein Cluster verbindet mehrere solcher Module (*linked modules*) zu Einheiten, um daraus ein ausführbares Programm zu bilden. Die beschriebene Enthaltenseinshierarchie des Engineering Viewpoints wird durch die Abbildung 3.13 verdeutlicht.

Weiterhin definiert das Engineering Model Kommunikationskanäle (*Channels*), welche die Bindungen aus dem Computational Model darstellen. Im Computational Viewpoint können Objekte, deren Computational Interfaces aneinander gebunden sind, Interaktionen eingehen. Zu jedem Computational Interface gibt es genau ein Engineering Interface, also eine in einem

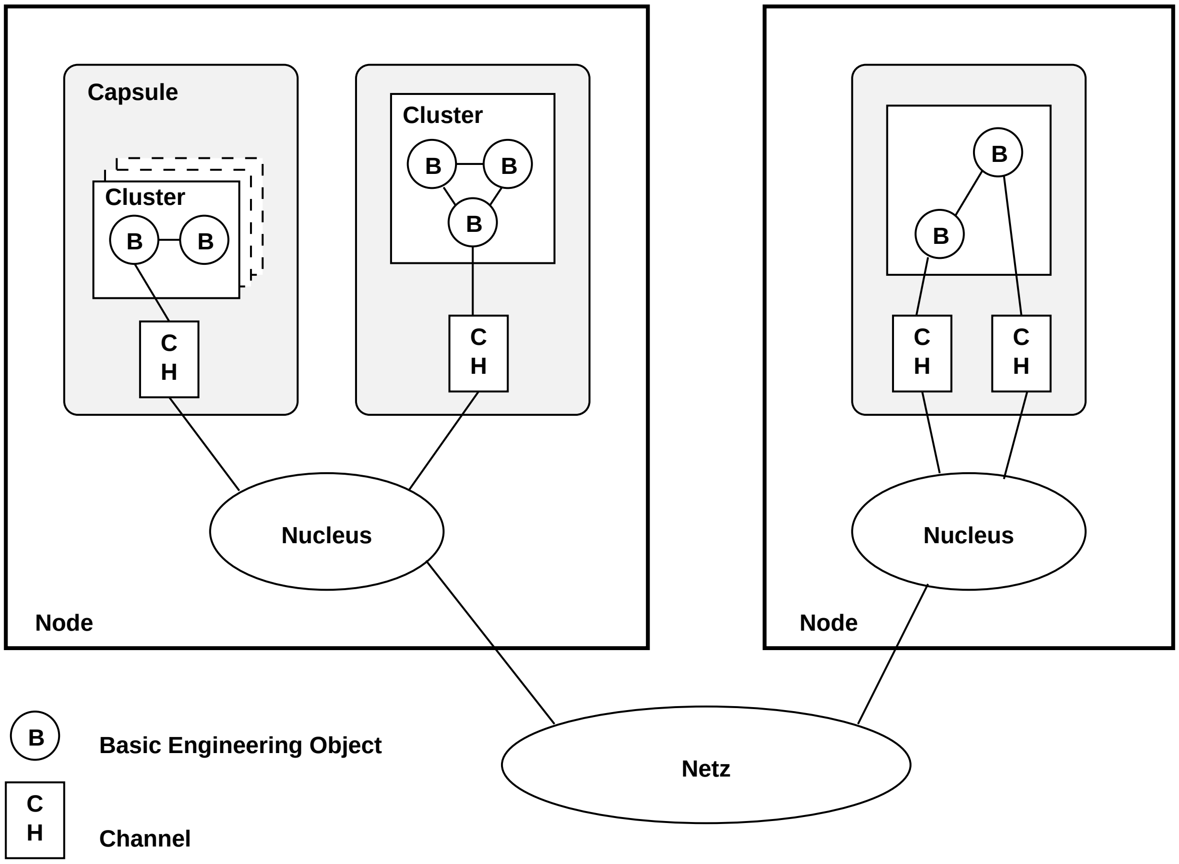


Abbildung 3.13: ODP Engineering Model

realen System identifizierbare Schnittstelle. Zum Instantiierungszeitpunkt eines Engineering Interface vergibt der Nucleus einen innerhalb einer bestimmten Domäne (*Engineering Interface Reference Management Domain*) eindeutigen Identifikator, die sog. *Engineering Interface Reference*. Anhand dieser Referenz kann ein Objekt eine verteilte Bindung eingehen. Besitzt also ein Objekt die Referenz einer Schnittstelle eines anderen Objekts, kann von den beteiligten Nucleus-Objekten ein Channel zwischen den Objekten aufgebaut werden, unter der Bedingung, daß die zugehörigen Computational Interfaces vom selben Typ sind.

Channels werden nur für Bindungen zwischen Objekten benötigt, die sich in Capsules unterschiedlicher Nodes befinden. Lokale Bindungen (*local bindings*) zwischen Objekten im selben Cluster oder Capsule bzw. zwischen Objekten zweier Capsules im gleichen Node sind nicht Gegenstand der Standardisierung von ODP. Diese Art der Interprozeßkommunikation soll aus Gründen der Performance durch den Compiler oder mit Mitteln des Betriebssystems (z.B. durch shared memory) realisiert werden. Ein Beispiel für einen Kanal zwischen zwei verteilten Objekten wäre eine Verbindung auf Schicht sieben des OSI Referenzmodells.

Der Aufbau eines Kanals ist in Abbildung 3.14 dargestellt. Einer der Hauptgründe, warum Kanäle im RM-ODP genau spezifiziert werden, ist das Ziel, verschiedene Transparenzen in der verteilten Umgebung einführen zu können. Hierzu gehören u.a. Orts-, Migrations- und Replikationstransparenz. Diese Konzepte benötigen ein mächtiges Kommunikationsmodell. Ein

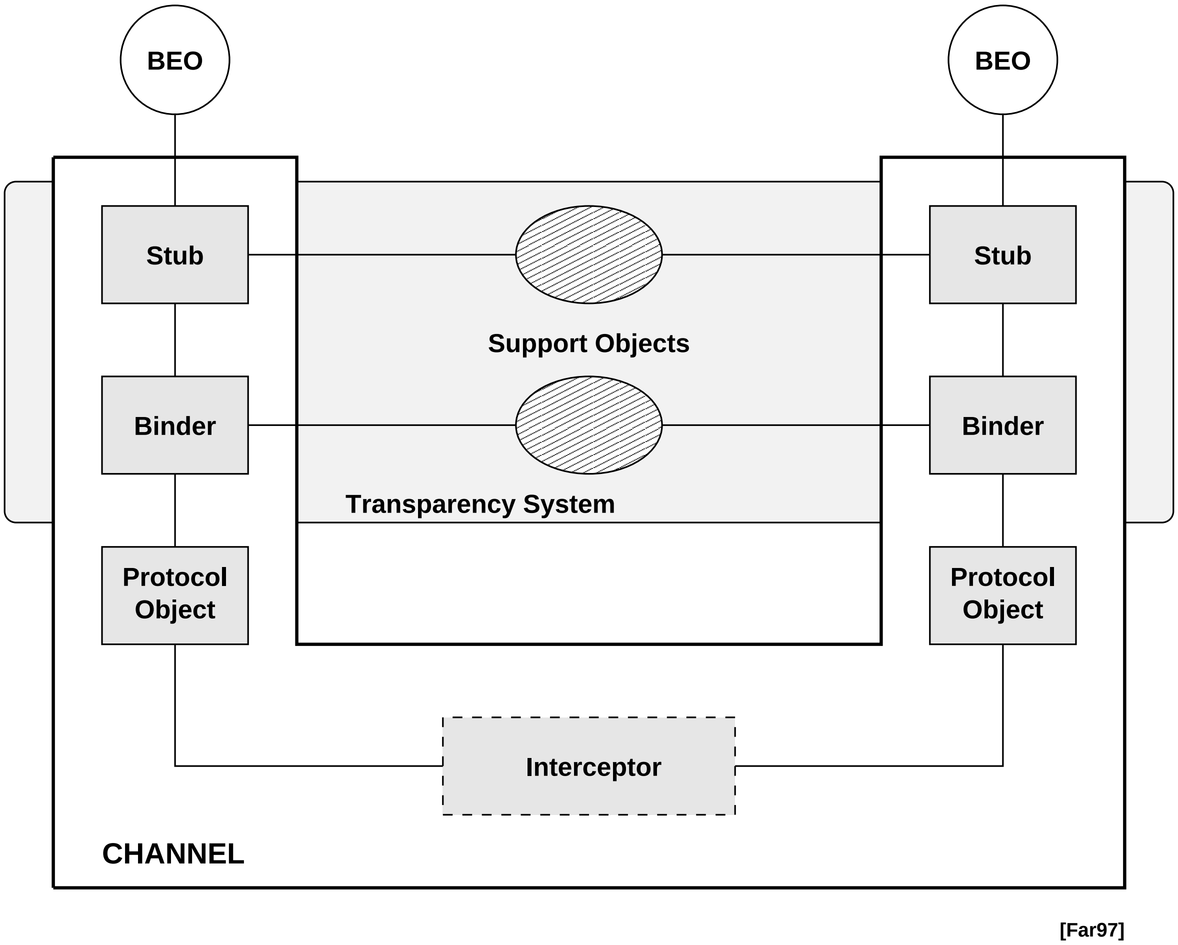


Abbildung 3.14: Generischer Aufbau eines Channel

Channel im RM-ODP setzt sich aus speziellen Engineering Objects zusammen, die diese Transparenzen realisieren sollen.

Stubs sind die Objekte in einem Kanal, die direkt mit den zu verbindenden Basic Engineering Objects interagieren. Sie sind abhängig vom Typ der unterstützten Schnittstelle. Bei einem Operation Interface übernehmen sie beispielsweise das Ein- und Auspacken der Parameter eines Prozeduraufrufs (*marshalling*). *Binders* sichern die Ende-zu-Ende-Integrität des Kanals. Neben dem Aufbau der Bindungen und der Fehlersicherung sorgen sie gemeinsam mit Unterstützungsobjekten, z.B. ein Verzeichnisdienst, außerhalb des Kanals für die Bereitstellung der angesprochenen Verteilungstransparenzen. Protokollobjekte realisieren das Transportsystem für den Kommunikationskanal mit einer ausreichenden Dienstgüte und Verläßlichkeit. Falls sich die zu verbindenden Objekte in unterschiedlichen Domänen technischer oder organisatorischer Art befinden, ist möglicherweise zusätzlich ein sog. *Interceptor* notwendig, der Format- und Protokollkonversionen durchführt oder Sicherheits- und Zugangskontrollen realisiert. Kanäle sind nicht auf Punkt-zu-Punkt-Verbindungen wie in der Abbildung beschränkt; das RM-ODP definiert auch Multicast-Verbindungen.

3.5.4 Open Distributed Management Architecture

Basierend auf dem „Metastandard“ RM-ODP werden für bestimmte Anwendungsbereiche spezielle ODP-Standards entwickelt. Für den Bereich Netz- und Systemmanagement entwickelt die ISO die *Open Distributed Management Architecture* (ODMA [ISO95a]). Dieser Standard beschreibt eine Architektur zur Spezifikation und Entwurf von verteilten Managementanwendungen wie auch von Applikationen für das Management von verteilten Anwendungen. ODMA erweitert die Konzepte und Managementfunktionen der OSI *Systems Management Architecture* (SMA) auf offene, verteilte Systeme.

Im Computational Viewpoint der ODMA sind Manager und Agent Computational Objects mit Managementschnittstellen. Es handelt sich hierbei stets um Operational Interfaces. Ein Manager besitzt eine Client-Schnittstelle, über die er Operationen wie `get(attr)` oder `set(attr)` auf einer Server-Schnittstelle eines Agenten ausführen kann. Umgekehrt kann der Agent asynchrone Ereignisse über seine Client-Schnittstelle an den Manager senden. Dieser empfängt die Meldungen an seiner Server-Schnittstelle. Ereignisse werden also wie Managementoperationen, ausgehend vom Agenten, modelliert. Bevor diese Interaktionen stattfinden können, müssen Manager und Agent Bindungen für die Schnittstellen etablieren. Abbildung 3.15 faßt die Beziehungen zwischen Manager und Agent zusammen.

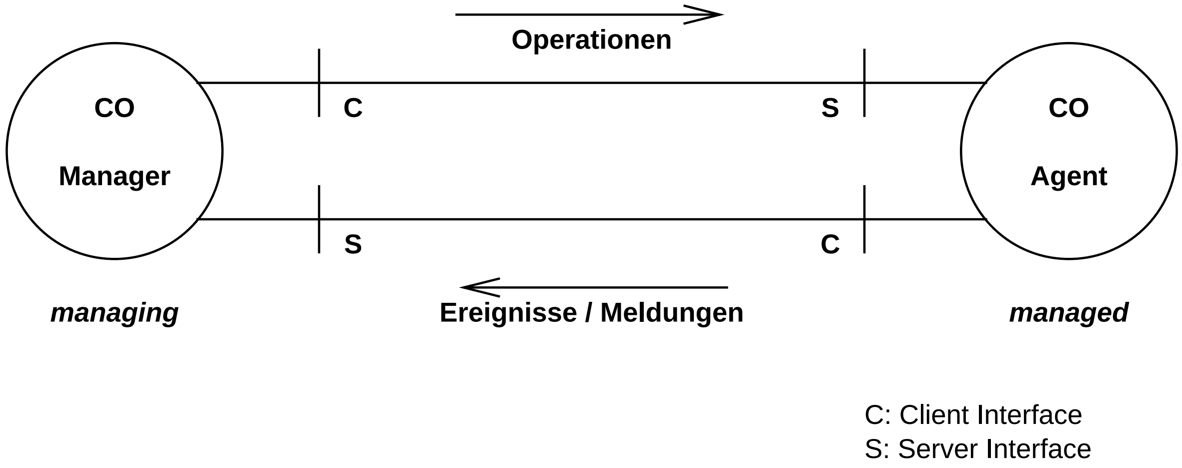


Abbildung 3.15: ODMA-Managementschnittstellen

Die Managementobjektklassen des in dieser Arbeit zu spezifierenden Modells definieren Agentenobjekte, die Managementschnittstellen im Sinne des obigen Modells bereitstellen. Die IDL-Beschreibungen zu den Klassen stellen die Computational Language dar. Der CORBA-konforme ORB stellt im Sinne des Engineering Viewpoint die Kommunikationsinfrastruktur für die Manager- und Agentenobjekte zur Verfügung. Das folgende Kapitel enthält einen Überblick über CORBA.

3.6 Die Common Object Request Broker Architecture (CORBA)

Common Object Request Broker Architecture (CORBA) ist ein offener Standard für verteilte Objekte. Er entstand aus der 1989 gegründeten, herstellerunabhängigen *Object Management Group* (OMG), die heute das weltweit größte Software-Konsortium ist. Der Trend zu objektorientierten und verteilten Anwendungen, der auf der zunehmenden Vernetzung von Computern und gleichzeitiger Etablierung objektorientierter Programmiersprachen basiert, erforderte ein Kommunikationsparadigma, welches die Interoperabilität von Software-Komponenten (Objekten) innerhalb einer heterogenen, vernetzten Systemlandschaft erlaubt. CORBA stellt eine Infrastruktur bereit, mit deren Hilfe verteilte Objekte unabhängig von der zugrundeliegenden Netztechnologie, unabhängig vom Betriebssystem und unabhängig von der Programmiersprache, in der sie realisiert sind, Interaktionen in Form von Methodenaufrufen eingehen können. Diese drei Ebenen der Heterogenität werden vor dem Software-Entwickler verschattet.

Es ist unmöglich im Rahmen dieser Arbeit eine ausführliche Einführung in CORBA zu bieten. Diese Aufgabe ist der Sekundärliteratur zu CORBA (z.B. [OHE96, Sie96]) überlassen. Technische Details sind in der CORBA-Spezifikation [Obj95] zu finden. Gute Tutorials ([Sch96b, Sch96c, Sch96a]) bietet Douglas C. Schmidt kostenlos über das Web an. Zur Programmierung von CORBA-Objekten in Java (siehe auch Kapitel 6) können [OH97] und [VD97, Vog97] empfohlen werden. Der folgende kurze Überblick über die wichtigsten Aspekte von CORBA ist an [Sta97] angelehnt.

CORBA ist eigentlich nur der Standard für die wichtigste Komponente der *Object Management Architecture* (Abbildung 3.16), nämlich des *Object Request Broker* (ORB). Der Broker stellt einen Vermittlungsmechanismus zum Aufruf von Methoden auf entfernten Objekten bereit, indem er den Aufruf eines Clients einschließlich Parameter an ein Server-Objekt wei-

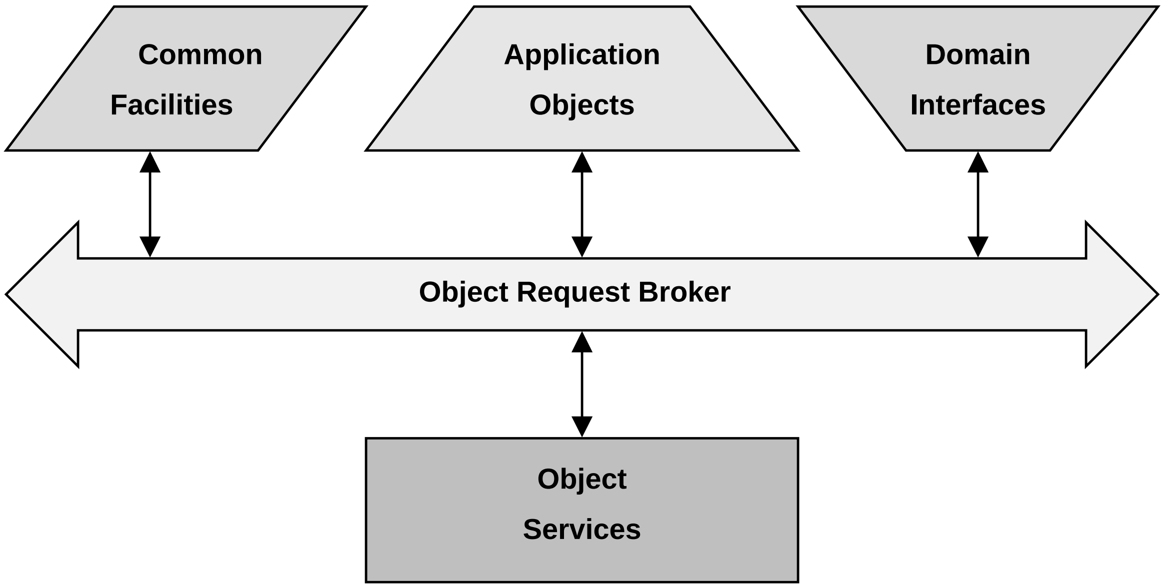


Abbildung 3.16: Die Object Management Architecture (OMA)

terleitet und umgekehrt Ergebnisse bzw. Fehler zurückgibt. Ebenso übernimmt er die Lokalisierung der Server-Objekte. Auf dem ORB bauen die übrigen Komponenten der Referenzarchitektur OMA auf. Die *Object Services* sind standardisierte, fundamentale Dienste, die für die Entwicklung von verteilten, objektorientierten Anwendungen erforderlich sind. Diese decken immer wieder benötigte Funktionen wie das Auffinden von Objekten (Naming-Service), das Kreieren, Kopieren, Verschieben oder Löschen von Objekten (Lifecycle-Service), der Versand von asynchronen Ereignismeldungen (Event-Service) oder die persistente Speicherung von Objekten (Persistent-Object-Service) ab. Auf einer höheren Ebene standardisieren die *Common Facilities* infrastrukturelle Dienste wie Komponententechnologien (z.B. *JavaBeans*) für Verbunddokumente oder Administrationsfunktionen. Die *Common Facilities* sind dabei unabhängig von einem bestimmten Anwendungsbereich (Domäne). Für spezielle Domänen wie Gesundheitswesen, Finanzdienstleistungen, Transport oder Telekommunikation bilden die *Domain Interfaces* standardisierte[4] Rahmenwerke. Nicht standardisiert sind natürlich die von den Software-Entwicklern erstellten Anwendungen in Form von *Application Objects*. In dieser Arbeit dienen die Anwendungsobjekte zur Realisierung eines Agenten und zugehöriger Managementanwendung.

Zur Verschattung der Heterogenität zieht die Broker-Architektur mehrere Abstraktionsschichten zwischen der Plattform, bestehend aus Netztechnologie und Betriebssystem, und der Anwendung ein. Der ORB vermittelt Nachrichten zum Transport von Methodenaufrufen eines Clients an ein Server-Objekt sowie umgekehrt zur Übermittlung von Ergebnissen oder Ausnahmen. Der Client muß nicht wissen, wo sich das Server-Objekt physikalisch befindet. Er benötigt lediglich eine Referenz auf den Server, anhand derer der ORB das Objekt über ein Verzeichnis lokalisiert. Hierzu ist es erforderlich, daß die Server-Objekte ihre Verfügbarkeit beim ORB anmelden. Eine weitere Abstraktionsschicht stellen Proxy-Objekte dar. Der Aufruf einer Methode auf einem entfernten Objekt soll sich nicht von einem auf einem lokalen Objekt unterscheiden. Hierzu dienen Stellvertreterobjekte (Proxies), die auf dem ORB aufsetzen. Auf Client-Seite verfügt ein Proxy über die gleiche Schnittstelle wie das entfernte Server-Objekt. Der Client ruft eine Methode auf dem Proxy auf. Dies führt – für den Client unsichtbar – zum Generieren einer plattformunabhängigen Nachricht, die der ORB weitertransportiert. Das Proxy wartet auf das Ergebnis, welches es dem Aufrufer im lokalen Format zurückgibt. Auf Server-Seite gibt es analog dazu ein Server-Proxy, welches gegenüber dem Server als lokaler Client auftritt. In einem großen Netz können mehrere ORBs (unterschiedlicher Hersteller und auf verschiedenen Betriebssystemen) an der Nachrichtenvermittlung beteiligt sein. Zur reibungslosen Kooperation der ORBs untereinander bedarf es eines gemeinsamen, standardisierten Kommunikationsprotokolls.

CORBA realisiert die beschriebene Architektur. Die Struktur des ORB ist in Abbildung 3.17 dargestellt. Die Proxy-Objekte, sog. *Stubs* auf Client-Seite und *Skeletons* auf Server-Seite, setzen auf dem Kern des ORB auf. CORBA definiert mit der *Interface Definition Language* (IDL) eine eigene Sprache, um die Unabhängigkeit von einer bestimmten Programmiersprache für Client und Server zu ermöglichen. IDL ist eine an C++ angelehnte Sprache zur Definition von Schnittstellen für Dienste, die ausschließlich Konstrukte zur Beschreibung von Attributen, Methoden und Datentypen eines Objekts enthält. Ein Server-Objekt beschreibt die Zugriffs-

[4] Die Standardisierung innerhalb der OMA wird nach und nach vollzogen. Während zum ORB und zu den meisten *Services* bereits Spezifikationen vorliegen, wird an den Standards für die *Common Facilities* und *Domain Interfaces* zur Zeit noch gearbeitet.

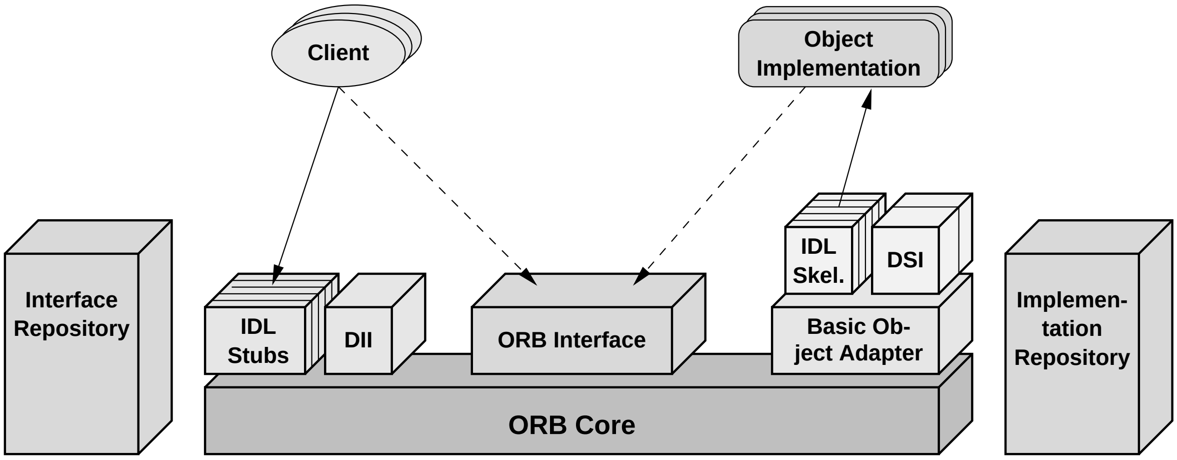


Abbildung 3.17: Die Struktur des CORBA 2.0 Object Request Broker

möglichkeiten auf seine Dienstschnittstelle für einen Client in Form einer IDL-Spezifikation. Die OMG legt über standardisierte *language mappings* die Abbildung von IDL auf konkrete Programmiersprachen wie C++, Java oder Smalltalk fest. Anhand dieser Abbildungsvorschrift generiert ein IDL-Compiler das Server-Skeleton in der jeweiligen Implementierungssprache. Der Benutzer eines Objekts erzeugt ebenfalls mit Hilfe eines IDL-Compilers den Client-Stub, der ein Zugriff auf das entfernte Server-Objekt wie auf eine lokale Klasse ermöglicht.

Vom IDL-Compiler generierte *Stubs* und *Skeletons* bilden eine statische Schnittstelle zur Nutzung von Diensten. Damit eine Anwendung auch dynamisch zur Laufzeit Dienste von Server-Objekten in Anspruch nehmen kann, deren IDL-Beschreibung zur Zeit der Übersetzung nicht vorlag, definiert CORBA das *Dynamic Invocation Interface* (DII). Zu allen im System registrierten Serverobjekten enthält das *Interface Repository* (IR) die IDL-Beschreibungen ihrer Schnittstellen. Mit den Typinformationen aus dieser Datenbank kann ein Client dynamisch über das DII einen Methodenaufruf an einen Server absetzen. Der Server kann DII-Aufrufe nicht von Stub-Aufrufen unterscheiden. Auf Server-Seite ist das *Dynamic Skeleton Interface* das Äquivalent zum DII. Über diese Schnittstelle können Aufrufe an dynamisch erzeugte Objektimplementierungen übergeben werden. Diese Funktionalität wird u.a. für Gateways zu anderen Objektsystemen oder für generische Brücken zwischen ORBs benötigt.

Gewöhnlich sind Aufrufe über das statische Interface synchron. Der Client blockiert, bis er vom ORB über den Stub ein Ergebnis oder eine Ausnahme erhält. Werden Operationen in IDL mit dem Schlüsselwort *oneway* versehen, benutzt CORBA hierfür eine asynchrone und unzuverlässige (*best effort*) Aufrufsemantik, bei der keine Ergebnisse oder Ausführungsbestätigungen zurückgegeben werden. Über das DII können Methoden zusätzlich auch asynchron *mit* Ergebnisrückgabe ausgeführt werden (*deferred synchronous*). Der Client blockiert hierbei nicht. Über einen Polling-Mechanismus wird überprüft, ob ein Ergebnis vorliegt.

Clients identifizieren das Objekt, auf dem sie eine Methode aufrufen möchten, gegenüber dem ORB durch eine eindeutige Objektreferenz (IOR), ohne über den physikalischen Ort des Objekts Kenntnis besitzen zu müssen. Der ORB unterhält mit dem *Implementation Repository* ein Verzeichnis, welches u.a. die Abbildung von IOR auf physikalische Objektposition erlaubt. Damit dieser Mechanismus funktioniert, müssen sich Server-Objekte über den *Basic Object*

Adapter (BOA) beim ORB registrieren. Der BOA generiert für ein neues Objekt die IOR, die der ORB in seinem Verzeichnis ablegt. Auch das Aktivieren von registrierten, aber nicht gestarteten Servern bei Eintreffen einer Client-Anfrage beim ORB kann der BOA übernehmen. Clients ermitteln die Objektreferenzen zu benötigten Diensten über Verzeichnisdienste (Naming-Service), Trader, aus Dateien, die IORs als Zeichenketten (*stringified object references*) enthalten, oder durch proprietäre Mechanismen. Hinter dem *ORB interface* verbergen sich verschiedene nützliche Funktionen sowohl für Clients als auch für Server.

Seit der Version 2.0 wird die Interoperabilität zwischen ORBs verschiedener Hersteller durch das *General Inter-ORB Protocol* (GIOP) gewährleistet, dessen Abbildung auf vorhandene Protokollwelten wie TCP/IP oder OSI standardisiert wurde. Jeder ORB muß zumindestens die Kooperation auf Basis von TCP/IP über das *Internet Inter-ORB-Protocol* (IIOP) unterstützen. Brücken dienen zur Konvertierung zwischen proprietären ORB-Protokollen und IIOP bzw. GIOP-Implementierungen für andere Basisprotokolle. Damit können auch Anschlüsse von CORBA an andere verteilte Objektumgebungen wie Java RMI von *JavaSoft* oder *Microsoft* DCOM geschaffen werden.

3.7 Bestehende Managementlösungen

3.7.1 Modulare Managementagenten auf SNMP-Basis

Bestehende Lösungen für das System- und Anwendungsmanagement im LAN-Bereich basieren meistens auf der Managementarchitektur des *Internet Activities Board* (IAB) und der *Internet Engineering Task Force* (IETF). Das Internet-Management (siehe z.B. [HA93, HNW95, Sta96]) fand seine weite Verbreitung aufgrund seiner Standardisierung und der relativ einfachen Implementierbarkeit der Spezifikationen. Ursprünglich ausschließlich für das Netzmanagement entwickelt, werden seine Konzepte seit etwa 1993 auch verstärkt für das Management von Endsystemressourcen und seit neuerem auch für das Anwendungsmanagement eingesetzt. Für ein integriertes Management ist es erforderlich, daß MIB-Module und deren Implementierungen (Agenten), die von dem jeweiligen Hersteller der Ressource stammen, innerhalb eines Endsystems kooperieren können. Hierzu wurde der SNMP-Hauptagent eines Systems um die offene Schnittstelle DPI (*Distributed Protocol Interface*) erweitert. Dieser Ansatz ist in Abbildung 3.18 dargestellt und in [HKN96] näher beschrieben.

Auf der Agentenseite können sich MIB-Module einzelner Ressourcen über DPI für einen bestimmten Teil des Internet-Registrierungsbaums anmelden und damit ihre Zuständigkeit für diesen Teil signalisieren. Das Protokollmodul des Hauptagenten übergibt Aufträge (*SNMP-Requests*) an den jeweils zuständigen Subagenten, der ein MIB-Modul realisiert, und nimmt umgekehrt Ergebnisse bzw. asynchrone Ereignismeldungen entgegen. Auf Managerseite können verschiedene Anwendungen über eine gemeinsame Plattform per SNMP auf den modularen Agenten zugreifen.

Die *Desktop Management Task Force* (DMTF) hat zur Einbindung von vernetzten PCs in das integrierte Management eine ähnliche Lösung entwickelt. Das *Desktop Management Interface* (DMI) ermöglicht ebenfalls den Zugriff auf unterschiedliche MIB-Module innerhalb eines zu managenden Endsystems. DMTF-MIBs bauen jedoch auf einem eigenen Informationsmodell

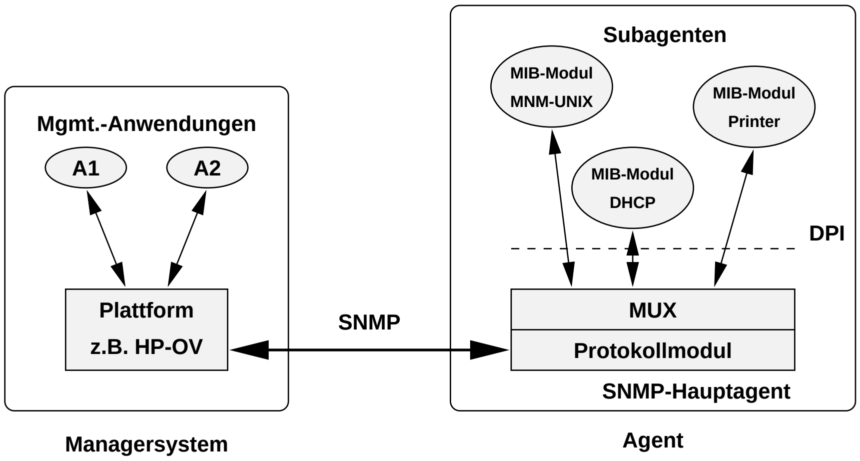


Abbildung 3.18: Managementarchitektur mit modularen Agenten

auf, deren Beschreibungssprache MIF (*Management Information Format*) eine Untermenge des Internet-Informationsmodells darstellt.

Diese Lösungen können allenfalls zu einer Integration innerhalb des Internet-Managements führen. Außerdem herrscht nicht einmal hier Einheit bei Herstellern und Standardisierungsgremien, wie die beiden unterschiedlichen Spezifikationen für Intra-Agentenschnittstellen zeigen. Es existieren ferner weitere, proprietäre Ansätze. Das später vorgestellte Produkt *Systems Monitor* von IBM setzt beispielsweise für die beschriebene Funktionalität das Protokoll SMUX ein. Kein Hersteller von Systemkomponenten oder Anwendungen wird jedoch MIB-Module und Subagenten für mehrere unterschiedliche Managementarchitekturen liefern.

Auch werden die in Kapitel 2.2 genannten Anforderungen an ein Informationsmodell für das integrierte Management von den bestehenden Lösungen auf SNMP-Basis nicht erfüllt. Neben den Schwächen des datentypbasierten Modellierungsansatzes und dem immer noch fehlenden Funktionsmodell ist vor allem die Zukunftssicherheit nicht gewährleistet, da es fraglich ist, ob sich SNMP als Basis für das Management verteilter Systeme durchsetzen kann.

Nachdem in diesem Abschnitt Techniken zur Kooperation von Agenten für verschiedene MIB-Module vorgestellt wurden, gibt der folgende Abschnitt einen Überblick, welche MIBs überhaupt bereits für das System- und Anwendungsmanagement existieren.

3.7.2 MIBs für das System- und Anwendungsmanagement

Es gibt unzählige proprietäre MIBs, die Hersteller zu ihren individuellen Produkten mitliefern. Zum Teil sind diese die Basis für sehr spezielle Managementwerkzeuge. Oft ist es sogar nur möglich, mit Hilfe eines MIB-Browsers einer Plattform die spezifische Managementinformation der Ressource auszulesen. Das Hauptproblem bei proprietären Hersteller-MIBs liegt darin, daß die bottom-up von der Ressource zur Verfügung gestellten Informationen meist nicht die Bedürfnisse eines Betreibers an das Management treffen. Ein weiteres Problem ist, daß die selbe Managementinformation in den unterschiedlichen Hersteller-MIBs immer wieder unter jeweils anderem Namen neu definiert wird. Dies führt zu erheblicher Redundanz und Unübersichtlichkeit im Internet-Registrierungsbaum.

Für das System- und Anwendungsmanagement wurden erst wenige generische MIBs von unabhängigen Organisationen entwickelt. Immerhin gibt es aber innerhalb der IETF diverse Arbeitsgruppen, die sich mit der Thematik befassen und die Vorschläge (*Internet-Drafts*) weiterentwickeln. Problematisch ist aber, daß erst wenige Implementierungen zu den generischen MIBs existieren und somit deren praktische Tauglichkeit für das Management heutiger verteilter Systeme schlecht beurteilt werden kann. Es folgt ein Überblick über ausgewählte bestehende MIBs:

Host Resources MIB [GW93]: Diese SNMP-MIB definiert Objekte für das Management von Endsystemen. Die Attribute sind so allgemein gehalten, daß sie auf eine Vielzahl von Endsystemen unabhängig von Architektur, Betriebssystem und installierten Anwendungen angewendet werden können. Sie besteht aus sechs Gruppen. Die ersten drei Gruppen behandeln allgemeine Betriebssystemressourcen (System), Speicherressourcen (Storage) und Managementinformation zu logischen Geräten (Device). Die übrigen drei Gruppen bieten Informationen zu lokal installierter Software und zu Anwendungen, die sich gerade in Ausführung befinden. Kritisch anzumerken ist, daß die MIB verteilte Systeme kaum und verteilte Anwendungen gar nicht berücksichtigt. Desweiteren ist nur ein Monitoring und kein aktives, steuerndes Management möglich.

MNM-UNIX-MIB [KH96]: In seiner Diplomarbeit [Kri94] hat Uwe Krieger einen Vorschlag für eine SNMP-MIB für das Management von UNIX-Endsystemen gemacht, die Konzepte der *Host Resources MIB* [GW93] im Hinblick auf aktives Management erweitert. Diese MIB wurde vom «Münchner Netzmanagement Team» (MNM-Team) zur MNM-UNIX-MIB weiterentwickelt. Diese MIB wurde mittels modularer Agenten, wie im letzten Abschnitt beschrieben, implementiert und erlaubt das steuernde Management der folgenden Ressourcen heterogener UNIX-Workstations:

- Devices: Prozessoren, Hauptspeicher, Festplatten, Drucker, lokale Dateisysteme
- Prozesse
- Benutzer und Gruppen

Die Informationen der MIB werden in Kapitel 4.3 in das Objektmodell dieser Arbeit integriert.

Network Services Monitoring MIB [KF94]: Diese MIB definiert Managementinformation, die sich auf alle Arten von Netzdiensten bzw. verteilten Systemdiensten anwenden lassen soll. Beispiele für solche Dienste sind Agenten für den Versand von elektronischer Post (MTA) oder Verzeichnisdienste (DNS). Sie soll ein effektives Monitoring der Kommunikationsressourcen auf der Anwendungsschicht erlauben, z.B. für Lastmessungen oder zum Aufspüren von überlasteten oder unterbrochenen Verbindungen. Hierfür führt sie zwei SNMP-Tabellen ein. Die erste Tabelle enthält einen Eintrag für jeden Netzdienst, der dem Monitoring unterworfen ist. Die zweite Tabelle besitzt Attribute zu allen zu den Netzdiensten gehörenden, offenen Kommunikationsverbindungen.

System Application MIB [SK97]: Diese MIB des IAB hat momentan den Status eines Internet-Drafts und ist noch nicht endgültig standardisiert. Sie definiert Objekte für das Fehler-, Konfigurations- und Leistungsmanagement von Anwendungen. Ihr Einsatzgebiet geht damit über das traditionelle Systemmanagement hinaus. Sie betrachtet eine

Anwendung als Managementobjekt, welches sich aus ausführbaren und anderen Dateien zusammensetzt und auf *einem* Rechner ausgeführt wird. Die Managementinformation der MIB soll auf möglichst alle Arten von Anwendungen anwendbar sein und vollständig durch Methoden des Betriebssystems zugänglich gemacht werden. Eine Implementierung dieser MIB erfordert daher keine Instrumentierung der Anwendungen, also kein Ändern oder Hinzufügen von Anwendungs-Code. Die generischen Attribute dieser MIB sollen später durch die *Application MIB* (siehe folgenden Punkt) und durch MIB-Module für spezifische Anwendungen verfeinert und erweitert werden. Dies erfordert dann aber eine Instrumentierung der Anwendungen. Das bedeutet, daß schon bei der Entwicklung Schnittstellen für das Management in die Anwendungen integriert werden müssen, wie sie z.B. auch das RM-ODP vorsieht. Die *System Application MIB* besteht aus zwei großen Gruppen mit Objekten für installierte Software und gerade ausgeführte Anwendungen. Damit erweitert sie die entsprechenden Gruppen der *Host Resources MIB*. Die wichtigsten Einschränkungen der *System Application MIB* sind die zwei folgenden Punkte:

- Beschränkung des Managements auf lokal installierte und ausgeführte Anwendungen
- Keine Steuerungsmöglichkeiten für aktives Management

Hier setzt die *Application Management MIB* an, die im folgenden besprochen wird.

Application Management MIB [SKPK97]: Diese MIB, die ebenfalls erst als Internet-Draft vorliegt, baut auf den vorher besprochenen *Host Resources MIB*, *Network Services Monitoring MIB* und *System Application MIB* auf. Sie führt weitergehende, wichtige Konzepte für das Anwendungsmanagement ein, deren Realisierung aber die Instrumentierung der Anwendungen voraussetzt. Hier ein kurzer Überblick über die Erweiterungen der Managementinformation und -funktionalität:

- Unterstützung für generische Durchsatzmessungen innerhalb von Anwendungen
- Auftragsmanagement (*units of work*)
- Möglichkeiten für generische Antwortzeitmessungen
- Eingeschränkte Unterstützung von verteilten Anwendungen
- Generisches Ressourcenmanagement bezüglich offener Dateien, IO-Kenndaten auf Anwendungsebene und Nutzung des Netzes
- Generisches Logging
- Aktive Elemente für das Konfigurationsmanagement und Möglichkeiten zum Starten und Stoppen von Anwendungen

Hierzu werden SNMP-Tabellen zu Diensten und deren Ressourcen wie offene Dateien, offene Kommunikationsverbindungen, Transaktionen und Prozessen eingeführt.

3.7.3 IBM Systems Monitor

Als Abschluß des Überblicks über den State-of-the-Art soll schließlich noch ein kommerizielles Produkt für das Systemmanagement vorgestellt werden. Es handelt sich um den *IBM Systems*

Monitor[5], Version 2, der in [IBM94] detailliert beschrieben wird. Dieses Produkt auf Basis von SNMP stellt Managementinformation zu Endsystemressourcen wie Festplatten, Prozessoren, Druckern, etc. und Betriebssystemressourcen wie Dateisystemen, Prozessen, Virtueller Speicher, etc. bereit. Den Aufbau der Managementlösung zeigt Abbildung 3.19. Die Bestandteile des Produkts sind grau schattiert dargestellt.

Auf der untersten Ebene der zu überwachenden Endsysteme arbeiten SNMP-Agenten, deren Standard-MIB MIB-2 über die Subagentenschnittstelle SMUX erweitert werden kann. Der *Systems Information Agent* (SIA) unterstützt die proprietäre SIA-MIB für Endsysteme unter den Betriebssystemen AIX 3.2.5 und OS/2. Workstations anderer Hersteller können über einen Agenten, der die *System Monitor MIB* unterstützt, in die Managementlösung eingebunden werden. Dieser Agent wurde für diverse Hardware-Plattformen und deren UNIX-Derivate portiert. Auf die Funktionen der genannten MIBs wird weiter unten eingegangen. Die SNMP-Agenten werden vom sog. *Mid-level Manager* (MLM) überwacht und gesteuert, der den zweiten wichtigen Bestandteil des Produkts ausmacht. Dieser entlastet die zentrale Managementstation (hier: *NetView for AIX*), indem er die Überwachung der Ressourcen (durch Polling) eines bestimmten Subnetzes übernimmt. Zu seinen Aufgaben gehören die Überwachung des Status der Endsysteme, das Sammeln und Auswerten von Managementdaten, die Schwellwertüber-

[5] Das Produkt ist inzwischen nicht mehr auf dem Markt. Die Funktionalität wurde in die Managementlösung *Tivoli TME 10* integriert.

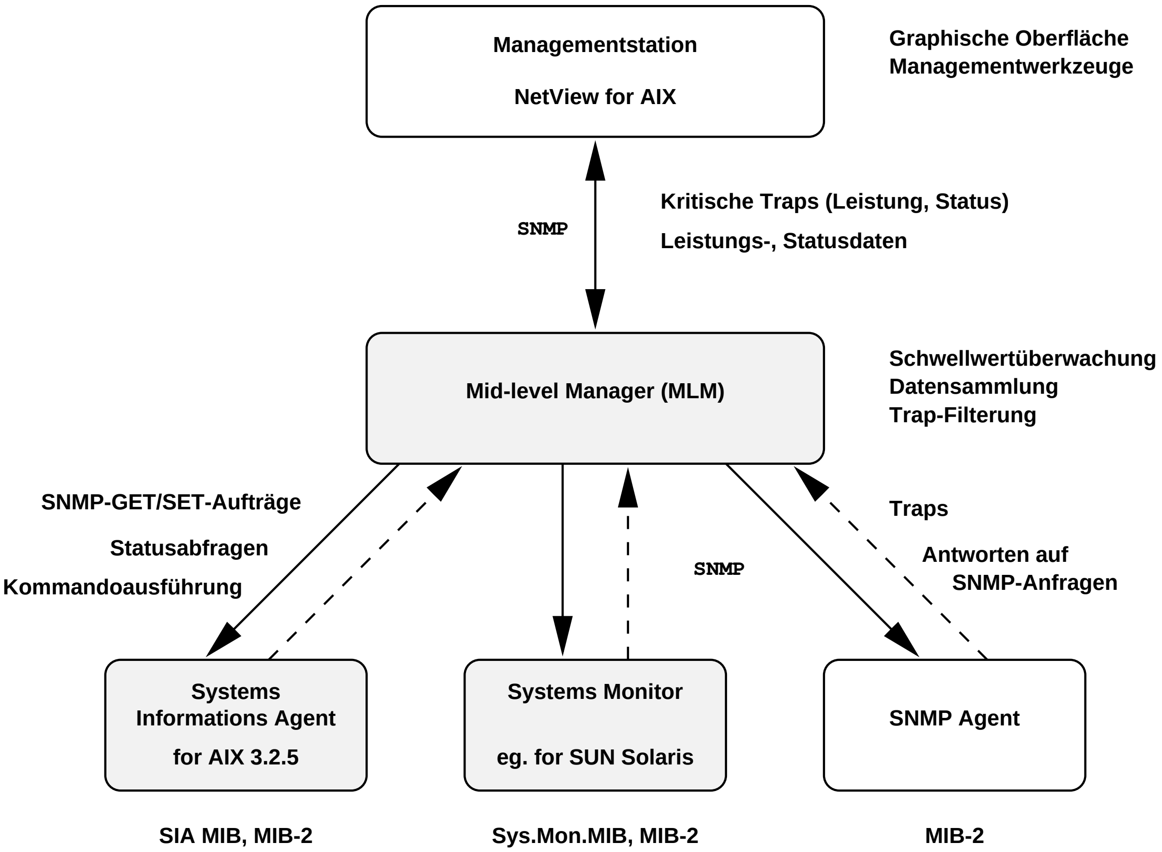


Abbildung 3.19: Aufbau des IBM Systems Monitor

wachung und die Filterung von asynchronen Ereignismeldungen (*traps*). Der MLM meldet Statusänderungen „seiner“ Ressourcen an den *Top-level Manager*, leitet kritische Traps weiter und stellt ihm aufbereitete Daten zur Verfügung.

Die SIA-MIB definiert über 600 Einzelobjekte für das Systemmanagement. Die Informationen der MIB-Variablen lassen sich in die Bereiche allgemeine Systembeschreibung, Systemkonfiguration, Geräte, Paging, Dateisysteme, Subsysteme, Prozesse, Benutzer und Systemauslastung einteilen. Die bereitgestellte Information ist zum Großteil auch über normale UNIX-Kommandos zugänglich. Über die MIB ist somit vor allem das Überwachen einer Workstation bezüglich Leistung, Fehler, Status, laufender Prozesse, etc. möglich. Erwähnenswert ist ferner die Möglichkeit, benutzerdefinierte Dateien auf Existenz, Zugriff, Inhalt und Modifikationen zu überwachen (*file monitor table*). Außerdem kann die MIB vom Benutzer um eigene Variablen erweitert werden. Zu diesen Variablen können beliebige UNIX-Kommandos spezifiziert werden, die die entsprechende Information ermitteln (*command table*).

Die Informationsmenge der SIA-MIB geht weit über die der *Host Resources MIB* hinaus. Außerdem ist sie bei weitem detaillierter. Dies wertet der Hersteller als Vorteil, da dadurch eine genauere Überwachung der Endsysteme möglich ist. Allerdings geht dabei der Anspruch der *Host Resources MIB*, möglichst generisch zu sein, vollkommen verloren. Die Anwendung dieser MIB in heterogener Umgebung ist nicht möglich. Eine Untermenge der Information der SIA-MIB enthält die *System Monitor MIB*. Der dazugehörige Agent wurde von IBM für verschiedene andere, weit verbreitete UNIX-Derivate portiert.

Die beiden beschriebenen MIBs werden den Anforderungen des integrierten Managements nicht gerecht. Da sie im Internet-Registrierungsbaum im privaten Herstellerzweig von IBM eingeordnet werden, ist ihre Anwendung in heterogener Umgebung von der Portierung der Agenten durch die Firma IBM abhängig. Ferner erlauben die MIBs nur die Überwachung der Systemressourcen. Das aktive Management ist auf das Setzen von SNMP-Variablen beschränkt. Das Anwendungsmanagement wird kaum berücksichtigt. Es ist lediglich möglich, die Existenz von Prozessen zu überwachen. Der Anwendungsstatus läßt sich nur für sog. Subsysteme des Betriebssystems ermitteln.

Der Ansatz dieser Arbeit erscheint für die Belange des integrierten Managements geeigneter. Die generischen Basisklassen des Objektmodells sollen die breite Anwendbarkeit auf beliebige Systeme sichern. CORBA als offener Standard wird von vielen Herstellern mitgetragen. Während beim *Systems Monitor* ein proprietärer (Sub-)Agent die Managementinformation für die gesamte Workstation liefert, können CORBA-Agentenobjekte unterschiedlichster Herkunft gemeinsam eine integrierte Managementlösung für ein individuelles System bilden.

Kapitel 4

Ein Objektmodell für das Management von UNIX-Systemen

Das Hauptziel dieser Diplomarbeit ist, ein Objektmodell für integriertes System- und Anwendungsmanagement zu spezifizieren. Dieses Kapitel beschreibt, wie dieses Ziel durch ein Top-Down-Vorgehen erreicht wird. Dazu wird im Abschnitt 4.1 kurz dargestellt, was das Modell leisten soll und warum eine Top-Down-Vorgehensweise gewählt wird. Der zweite Abschnitt beschreibt die Basisklassen für das Management, die aus den Konzepten des Referenzmodells für offene verteilte Verarbeitung (RM-ODP) gewonnen werden. In Abschnitt 4.3 wird gezeigt, wie der vorhandene Prototyp eines Objektmodells für das Systemmanagement von UNIX-Workstations in das in dieser Arbeit erarbeitete Modell integriert werden kann. Für das Management von Systemdiensten werden anschließend in Abschnitt 4.4 generische Klassen aus den ODP-Basisklassen abgeleitet. Die Leistungsfähigkeit des Objektmodells wird im Abschnitt 4.5 durch den Vergleich der Managementinformation und -funktionalität des Modells mit der einiger standardisierter bzw. vorgeschlagener MIBs für das System- und Anwendungsmanagement überprüft. Eine Zusammenfassung der Ergebnisse findet sich in Abschnitt 4.6.

4.1 Top-Down-Modellierung

Eine der Anforderungen des integrierten Managements an die Modellierung ist, heterogene Systeme durch *ein* Informationsmodell beschreiben zu können. Hierzu eignet sich die objektorientierte Modellierung, wie in Kapitel 2.1.5 beschrieben, am besten. Es sollen also Objektklassen gefunden werden, die von herstellerspezifischen, proprietären Details der Ressourcen abstrahieren, aber dennoch ein effizientes Management der Ressource erlauben.

Bei der Top-Down-Vorgehensweise wird versucht, zuerst generische, also möglichst allgemeine, nicht systemspezifische Basisklassen einzuführen. Hierzu werden im nächsten Abschnitt Konzepte des RM-ODP herangezogen, weil dieses Modell versucht, einen Architekturrahmen für die Entwicklung und Spezifkation von Software in einer offenen, verteilten Systemumgebung zu geben. Da das RM-ODP natürlich aufgrund dieser Intention völlig systemunabhängig ist, aber dennoch die erforderlichen Objekte einer verteilten Systemlandschaft genau beschreibt, werden einige Konzepte und Objekte des RM-ODP unter dem Managementgesichtspunkt benutzt, um generische Basisklassen für das System- und Anwendungsmanagement zu finden. Hierbei werden in dieser Arbeit Ansätze aus der bisher unveröffentlichten Arbeit [Neu97] von Bernhard Neumair aufgegriffen und vertieft.

Sind Basisklassen gefunden, gilt es, für diese Attribute und Methoden zu definieren. Diese stellen die Managementinformation bzw. -funktionalität der MOCs dar. Attribute und Methoden ergeben sich aus der Anforderungsanalyse der Funktionsbereiche Konfiguration, Fehler, Leistung, Sicherheit und Abrechnung. Gerade im Bereich des Systemmanagements gibt es auch bereits einige standardisierte MIBs, die als „Lieferanten“ für Attribute und Methoden dienen können.

Es ist nicht leicht, bei der Definition von Attributen und Methoden einen guten Kompromiß zu finden. Wird zuwenig Information oder Funktionalität definiert, ist ein effizientes Management der Ressource mit dieser Objektklasse nicht möglich. Eine wichtige Fragestellung ist daher, welche Managementaspekte generell für diese Klasse von Ressourcen gelten. Bei einer sehr detaillierten Modellierung besteht andererseits die Gefahr, daß nicht mehr alle Ressourcen dieser Klasse durch die Basisklasse abgedeckt werden. Außerdem muß natürlich darauf geachtet werden, daß die spätere Implementierung des Modells möglich ist. Die spezifizierte Information muß daher einem Agenten für die modellierte Ressource zugänglich sein.

Die Basisklassen können für die Anwendung auf konkrete Systeme oder Ressourcen anschließend in Unterklassen weiter verfeinert werden. Hierzu ist eine geeignete Spezialisierungshierarchie zu finden, bei der strikte Vererbung anzuwenden ist. Auf diese Weise können auch bereits vorhandene spezielle Modelle in das Objektmodell integriert werden. Allgemeingültige Attribute und Methoden können gegebenenfalls im Modell „nach oben geschoben“ werden, d.h. in die Basisklassen aufgenommen werden.

4.2 Generische Basisklassen zum RM-ODP

Das Referenzmodell für offene verteilte Rechensysteme (*RM-ODP*) beschreibt unter Zuhilfenahme verschiedener Viewpoints die Architektur verteilter Systeme (vgl. 3.5). Für das System- und Anwendungsmanagement relevante Konzepte finden sich vor allem im Computational Viewpoint und im Engineering Viewpoint. Aus den Konzepten dieser beiden Viewpoint Languages können für die meisten der zu managenden Ressourcen geeignete generische Managementobjektklassen gewonnen werden. Mit der Identifikation und Beschreibung solcher MOCs befassen sich die folgenden Abschnitte.

4.2.1 Computational Viewpoint

Im weiteren werden die auf dem Computational Viewpoint basierenden Objektklassen herausgearbeitet und ihre Bedeutung für das Management erläutert. Den entsprechenden Ausschnitt aus dem Objektmodell zeigt Abbildung 4.1.

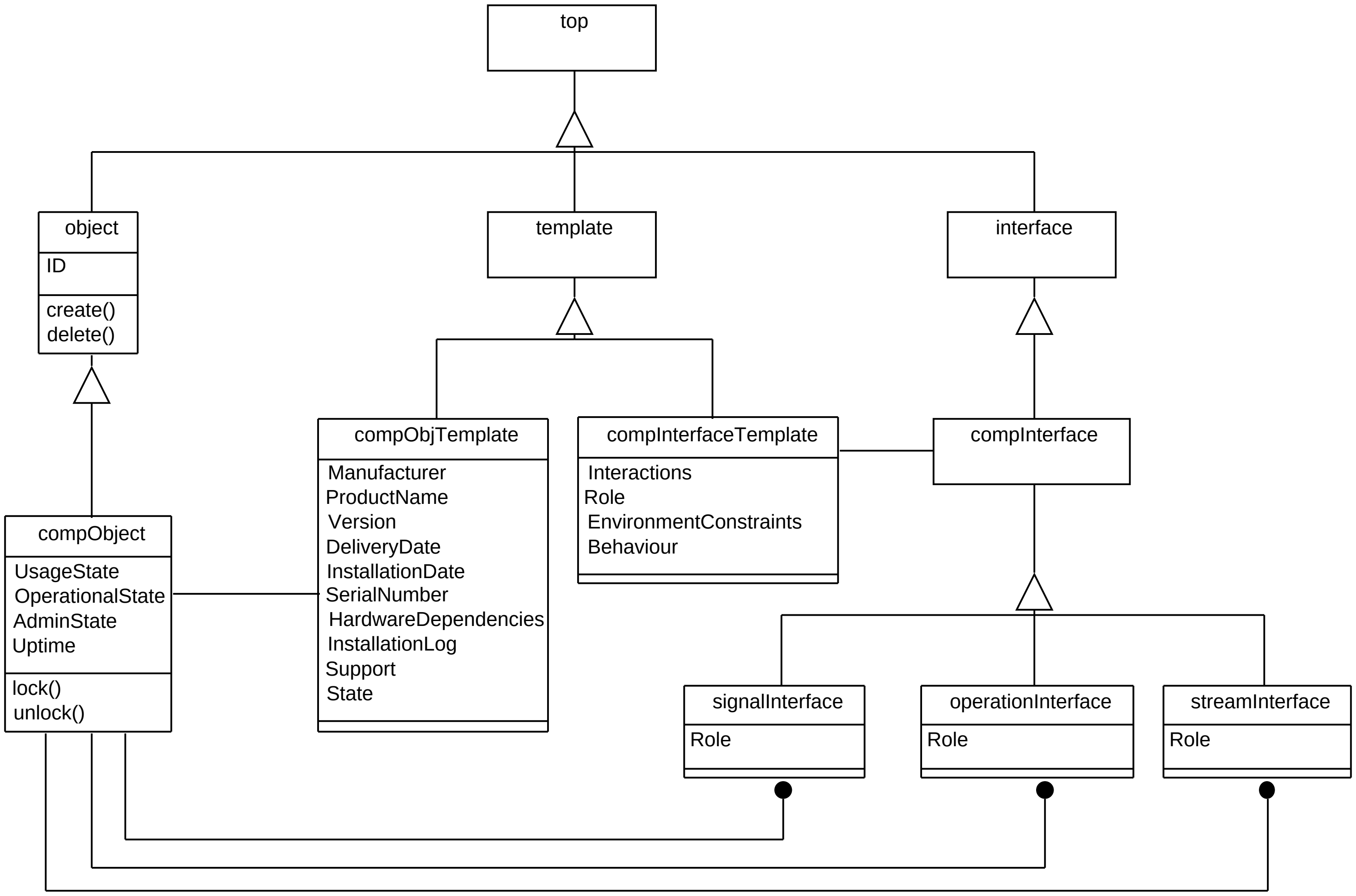


Abbildung 4.1: Managementobjektklassen zum Computational Viewpoint

Computational Object

Das Konzept des *Computational Object* wurde in Kapitel 3.5.2 erläutert. Ein Computational Object ist eine Komponente einer verteilten Anwendung, die an ihren fest definierten und typisierten Schnittstellen Dienste anbieten oder nutzen kann. Damit eine Interaktion stattfinden kann, müssen die beteiligten COs die entsprechenden Schnittstellen aneinander binden.

Für das Management betrachten wir ein Computational Object im objektorientierten Sinn als eine Software-Komponente, die zur Laufzeit ein Operation Interface zur Verfügung stellt, über welches Managementoperationen ausgeführt werden können. Natürlich fallen unter diese Definition auch Agenten einer (verteilten) Managementanwendung, die Netz- und Systemressourcen überwachen und steuern.

Es wird daher die Managementobjektklasse **compObject** eingeführt, deren Instanzen die managementrelevanten Abstraktionen benannter Software-Komponenten darstellen. Ein Beispiel für eine Instanz der Klasse **compObject** ist ein gestarteter NFS-Server auf einem System. Vorerst erhält die Klasse nur das Attribut `Uptime`, welches der Startzeitpunkt der SW-Komponente ist.

Die Klasse **compObject** wird von einer noch allgemeineren Klasse **object** abgeleitet. In einem objektorientierten System besitzt jedes Objekt einen eindeutigen Identifikator (`ID`). Objekte können neu erschaffen oder gelöscht werden. Auf welche Weise diese Vorgänge, die durch die Methoden `create()` und `delete()` repräsentiert werden, in einem konkreten System realisiert sind (z.B. Konstruktoren, *Object Factories*), wird für das Modell nicht näher betrachtet.

Computational Interface

Wie bereits erläutert, sind die Schnittstellen eines Computational Objects die Punkte, an denen – je nach Rolle – ein Dienst bereitgestellt oder genutzt wird. RM-ODP definiert unterschiedliche Arten von *Computational Interfaces*, abhängig davon, ob es sich bei den Interaktionen um Signale, Operationen oder Datenströme handelt. Vergleiche hierzu Kapitel 3.5.2.

Schnittstellen werden im Modell durch die allgemeine Klasse **interface** vorgesehen. Hiervon abgeleitet wird die Klasse **compInterface** für ein Computational Interface, die wiederum in die drei Klassen **signalInterface**, **operationInterface** und **streamInterface** spezialisiert wird. Letztere werden durch Aggregationsbeziehungen der Klasse **compObject** zugeordnet. Die Rolle, die das Objekt bezüglich der Interaktionen an einer Schnittstelle einnimmt, wird durch das Attribut `Role` modelliert. Für eine Instanz von **operationInterface** kann es die Werte «Client» oder «Server» annehmen.

Anhand dieser Klassen kann eine Managementanwendung überwachen, welche Dienste eine SW-Komponente anbietet. Sie gestatten daher einen sog. *service view* auf eine laufende Anwendung. Natürlich ist es erforderlich, daß ein Agent bzw. die SW-Komponente selbst Instanzen dieser Klassen kreiert. Jede SW-Komponente sollte für den Zugriff des Managements zusätzlich ein Management Operation Interface bereitstellen, über welches die Komponente überwacht und gesteuert werden kann.

Computational Interface Template

Im RM-ODP gehört zu jedem Computational Interface ein *Computational Interface Template*, welches die Eigenschaften der Schnittstelle genau spezifiziert. Wie in Kapitel 3.5.2 beschrieben, enthält das Template hierfür die Signatur der möglichen Interaktionen, die Rolle, das Verhalten und die Bedingungen, die die Schnittstelle an die Umgebung stellt. Im RM-ODP umfaßt das Template alle Informationen, die für das Instantiieren der Schnittstelle zur Laufzeit erforderlich sind.

Für das Modell wird die Klasse **compInterfaceTemplate** mit entsprechenden Attributen eingeführt. Diese geht eine 1:1-Assoziation zur Klasse **compInterface** ein. Für das Management sind Informationen wichtig, welche Interaktionen an einer Schnittstelle möglich sind. Für das Fehler- und Leistungsmanagement sind zusätzlich die Anforderungen an die Dienstgüte relevant.

Die Attribute, die die Managementinformation bereitstellen, werden anhand eines Operation Interface erklärt. Das Attribut `Interactions` gibt Auskunft über die Operationen, die ein Server an dieser Schnittstelle ausführen kann. Für die Kommandoschnittstelle eines FTP-Servers wären dies z.B. die akzeptierten Befehle. Das Attribut `Role` legt fest, ob die Schnittstelle für die Menge der Operationen die Rolle «Client» oder «Server» einnimmt. Schnittstellen mit der Rolle «Client» bezeichnen Dienste, die das CO in Anspruch nimmt, und die deshalb für den ordnungsgemäßen Betrieb der Komponente von anderen COs im System bereitgestellt werden müssen.

`Behaviour` beschreibt das beobachtbare Verhalten des Objektes in Bezug auf die Operationen der Schnittstelle. Die Beschreibung kann auch Angaben zu internen Aktionen des Objekts und zur erlaubten Aufrufreihenfolge von Operationen enthalten. Das Attribut `EnvironmentConstraints` enthält Dienstgüteanforderungen des betreffenden Objekts und der unterstützenden Umgebung, damit der an der Schnittstelle angebotene Dienst seinerseits einen bestimmten Grad an Dienstgüte bereitstellen kann.

Interaction Info

Systemdienste stellen einen Dienst an einer Schnittstelle zur Verfügung. Ein bekanntes Beispiel für einen solchen Dienst ist ein FTP-Server. Dieser besitzt eine Kommandoschnittstelle, an der sich Benutzer anmelden und authentifizieren können, um danach Dateien aus einem hierarchischen Verzeichnisbaum herunterladen zu können. Der FTP-Server instantiiert also ein Operation Interface, welches Interaktionen zwischen dem Client (Benutzer) und dem Server über Operationen wie z.B. `open`, `close`, `user`, `cd`, `get`, `put` usw. zuläßt. Durch das Kommando `get` wird zusätzlich ein Stream Interface instantiiert, über welches eine Datei als Bytestrom (*flow*) übertragen wird.

Für das Management eines FTP-Dienstes sind nun beispielsweise folgende Fragestellungen interessant. Wieviele Benutzer haben den Dienst seit einem bestimmten Zeitpunkt in Anspruch genommen? Wieviele unberechtigte Anmeldeversuche wurden zurückgewiesen? Welche Datenmenge hat ein bestimmter Benutzer übertragen? Welche durchschnittliche Datenrate wurde bei Dateiübertragungen erzielt? Wie oft kam es zu Abbrüchen von Übertragungen? Diese Fragestellungen berühren mehrere funktionale Dimensionen des Managements.

Hieraus ist zu erkennen, daß unter anderem die Bereiche Leistungs- (*performance*), Abrechnungs- (*accounting*), Fehler- (*fault*) und Sicherheitsmanagement (*security management*) Informationen über Interaktionen benötigen. Dieses Bedürfnis wird im Objektmodell durch die generische Managementobjektklasse **interactionInfo** berücksichtigt. Diese Klasse definiert Managementinformation, die auf alle Typen von Interaktionen anwendbar ist. Das Attribut `Name` identifiziert die betreffende Interaktion. Im Zähler `Count` wird die Häufigkeit einer bestimmten Interaktion festgehalten, während `Last` den Zeitpunkt des letzten Auftretens enthält. `Bytes` ist ein Zähler für die Anzahl der übertragenen Bytes einer Interaktion. Das Zurücksetzen der Attribute wird durch die Methode `reset()` realisiert.

Über den Typ der Interaktion wird die Vaterklasse **interactionInfo** weiter spezialisiert. Es entstehen die Sohnklassen **signalInfo**, **announcementInfo**, **interrogationInfo**, **terminationInfo** und **flowInfo**. Diese spezialisierten MOCs werden durch Assoziationen mit den entsprechenden Typen von Computational Interfaces in Relation gesetzt. Zur Laufzeit können einer Instanz eines Interface je nach Anzahl der erlaubten Interaktionen natürlich ein oder mehrere Instanzen eines speziellen Interaction-Infos zugeordnet sein. Die OMT-Modellierung der Klassen ist der Abbildung 4.2 zu entnehmen.

Bei einem kontinuierlichen Datenstrom, wie z.B. einer Audioübertragung, sind Echtzeitbedingungen einzuhalten, um einen gewissen Grad an Dienstgüte zu garantieren. Dementsprechend werden in der MOC **flowInfo** u.a. Attribute wie Übertragungsverzögerung (`Latency`), Varianz der Übertragungsverzögerung (`Jitter`), Durchsatz (`Throughput`) und das Verhältnis zwischen Spitzen- und durchschnittlicher Datenübertragungsrate (`Burstiness`) vorgesehen. Auch der Zeitpunkt des Beginns einer Übertragung (`StartTime`) ist von Interesse.

Die Klasse **interrogationInfo** enthält zusätzliche Managementinformation für RPC-ähnliche Interaktionen. Hier sind für das Fehler- und Leistungsmanagement Attribute wie Antwortzeit (`LastDelay`) und Absendezeitpunkt einer Anfrage (`StartTime`) wichtig. Zu einer Anfrage kann es mehrere Typen von Antworten (*terminations*) geben. Natürlich sind auch sämtliche Fehler, die bei einer Anfrage auftreten können, mögliche Antworten. Die Klasse **terminationInfo** ist daher durch die Anforderungen des Fehler- und Sicherheitsmanagements gerechtfertigt. Analoges gilt für die Klasse **signalInfo**.

Alle oben skizzierten Fragen beim Management eines FTP-Dienstes lassen sich nun durch Auswertung der Managementinformation der eingeführten Klassen beantworten. Das Attribut `Count` in einer Instanz von **interrogationInfo** für die Operation `open` liefert die Anzahl der Dienstnutzer. Instanzen von **terminationInfo** für die Antworten «Access denied» und «Connection broken» geben Auskunft über die Anzahl abgewiesener Anmeldeversuche bzw. unterbrochener Datenübertragungen. Der Mittelwert über das Attribut `Throughput` der Instanzen von **flowInfo** ist die durchnittliche Übertragungsrate.

Sicherlich fehlen Attribute, um auch spezielle Interaktionen wie z.B. Transaktionen in einer verteilten Datenbank managen zu können. Dies kann aber durch weitere Spezialisierung der eingeführten generischen Managementobjektklassen leicht erreicht werden. Durch die Beschränkung wird erreicht, daß die MOCs für viele verschiedene Systemdienste verwendbar sind. Außerdem wurde gezeigt, daß trotz des hohen Abstraktionsgrades viele Aspekte aus den genannten Funktionsbereichen von Managementanwendungen durch die bereitgestellte Information behandelt werden können.

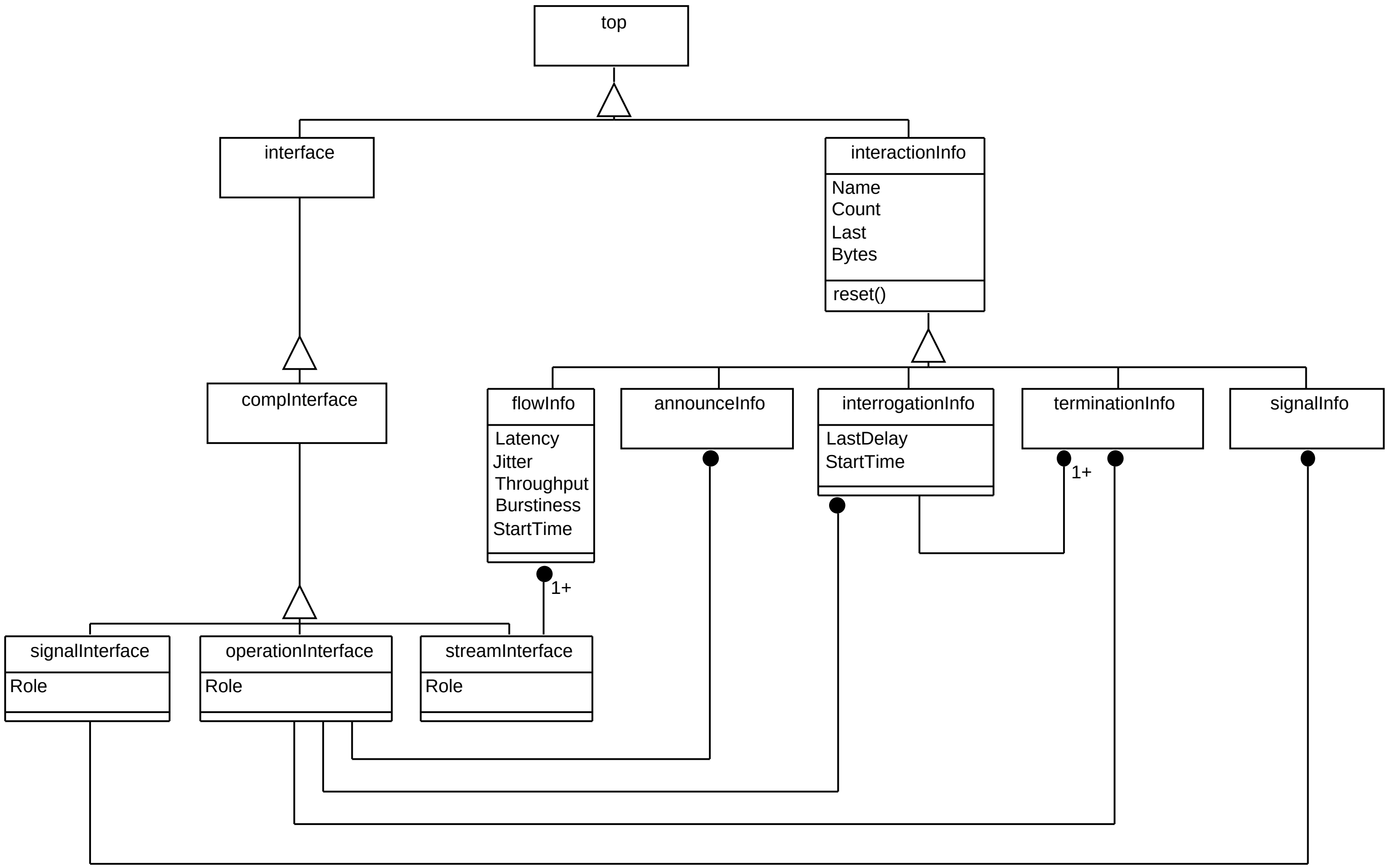


Abbildung 4.2: Managementobjektklassen zu Interaktionen

Computational Object Template

Ein *Computational Object Template* umfaßt laut RM-ODP alle für die Instantiierung des COs erforderlichen Informationen. Unter dem Managementgesichtspunkt betrachten wir das Computational Object Template als „ein Stück“ Software in einem verteilten Rechensystem. Die Instantiierung des Templates entspricht dem Starten der Software-Komponente und damit beispielsweise dem Bereitstellen eines Systemdienstes.

In das Modell wird die Klasse **compObjTemplate** eingefügt, die wichtige Informationen für das Software-Management zur Verfügung stellt. Ein Überblick über die Aufgaben des Software-Managements und die Ansätze der DMTF findet sich in Kapitel 2.1.3. Ein Schwerpunkt ist sicherlich die Installation von Software-Komponenten. Dies wird bei der Modellierung daher besonders berücksichtigt.

Die Attribute der MOC **compObjTemplate** (siehe Abbildung 4.3) basieren auf der *Software Standard Groups Definition* der DMTF. Es ist sinnvoll, detaillierte Informationen für das SW-Management in der generischen Klasse **compObjTemplate** anzusiedeln, da diese Managementinformation unabhängig vom Typ der SW-Komponente ist. Damit eine Managementapplikation Aufgaben des SW-Managements in verteilter heterogener Umgebung überhaupt erfüllen kann, ist eine möglichst hohe Abstraktion von der spezifischen Komponente erforderlich. Dies rechtfertigt die Modellierung. Aus verschiedenen Gründen wurde die Managementinformation aber auf die zusätzlichen MOCs **Package** und **InstallElement** sowie Beziehungsklassen für die Abbildung der Abhängigkeiten aufgeteilt.

Im Computational Viewpoint des RM-ODP werden verteilte Anwendungen und Systemdienste als funktionale Zerlegung in Objekte dargestellt, die über wohldefinierte Schnittstellen interagieren. Hierbei soll u.a. Orts- und Verteilungstransparenz herrschen. Genausowenig ist Bestandteil des Computational Model, welche Prozesse und Dateien ein Computational Object in verteilter Umgebung realisieren. Diese Aspekte werden vom Engineering Viewpoint berücksichtigt. Daher würden für das Management relevante Attribute einer SW-Komponente, wie z.B. Installationsort oder die Liste der zu der Komponente gehörenden Dateien, gegen die Architektur des RM-ODP verstoßen, wenn sie in die MOC **compObjTemplate** aufgenommen werden würden. Da aber auch der Engineering Viewpoint das Konzept des SW-Pakets nicht beinhaltet, wird die Klasse **Package** mit diesen Attributen ins Modell aufgenommen.

Diese Modellierung spiegelt auch die Realität besser wider. Ein SW-Paket, also eine Instanz der Klasse **Package**, kann aus mehreren SW-Komponenten, also Instanzen der Klasse **compObjTemplate**, bestehen. Beispielsweise kann ein Paket für den Systemdienst NFS gleichzeitig die Komponenten NFS-Server und NFS-Client enthalten. Im folgenden wird kurz auf die Attribute und Methoden der betrachteten Klassen eingegangen.

In der Klasse **compObjTemplate** sind die Attribute `Manufacturer`, `ProductName`, `Version`, `DeliveryDate`, `InstallationDate`, `SerialNumber` selbsterklärend. `HardwareDependencies` beschreibt Anforderungen der SW-Komponente an die Hardware wie z.B. Hauptspeicherbedarf oder benötigte Bildschirmauflösung. `InstallationLog` ist der Name eines Logs, in dem Meldungen über Ereignisse, die während der Installation auftreten, abgelegt werden. `Support` gibt an, auf welche Weise Unterstützung für die Software erlangt werden kann. In `State` könnten Informationen darüber enthalten sein, ob die Installation erfolgreich war und ob die Komponente gestartet werden kann. SW-Komponenten können andere Komponenten erfordern oder

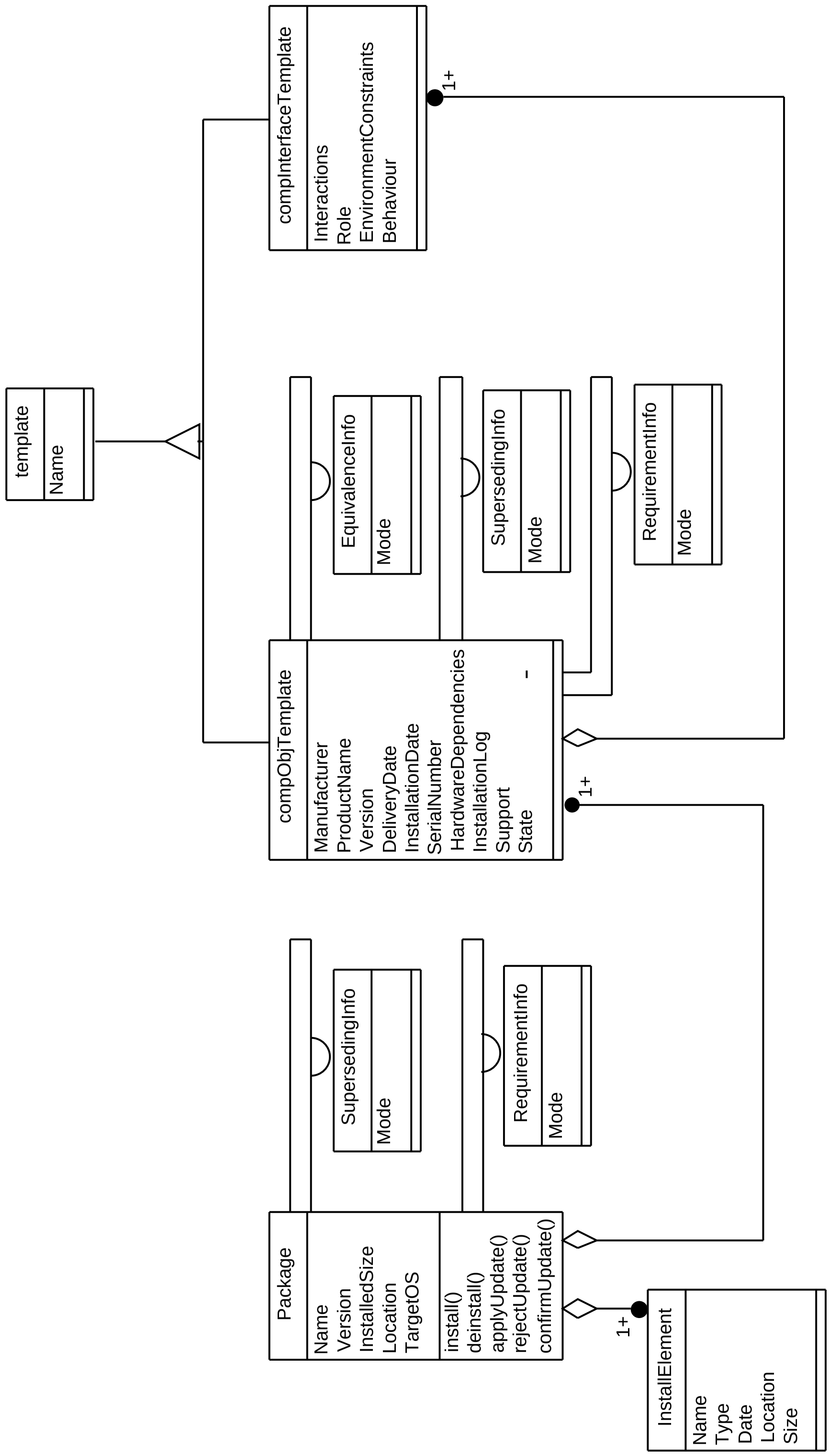


Abbildung 4.3: Managementobjektklassen für das Software-Management

ersetzen oder funktional äquivalent zu anderen sein. Dies wird durch die Beziehungen mit den Beziehungsklassen **RequirementInfo**, **SupersedingInfo** und **EquivalenceInfo** modelliert. Dabei beschreibt das Attribut `Mode` Einschränkungen für die Beziehung. Beispielsweise kann eine Komponente *ab* einer oder *bis* zu einer bestimmten Version erforderlich sein.

Die Attribute der Klasse **Package** sind ebenfalls nahezu selbsterklärend. `Location` beschreibt die Orte, an denen Teile der SW-Komponenten installiert sind. SW-Pakete können installiert (`install()`) und deinstalliert (`deinstall()`) werden. Beim Einspielen einer neuen Version eines Pakets kann der Upgrade unter Sicherung des alten Stands durchgeführt werden (`applyUpdate()`). Anschließend können die Änderungen zurückgenommen (`rejectUpdate()`) oder endgültig übernommen werden (`confirmUpdate()`). Auch Pakete können andere erfordern bzw. ersetzen. Dies wird analog zur Klasse **compObjTemplate** modelliert. Ein SW-Paket besteht in der Praxis aus einer Anzahl von Dateien (vgl. die Gruppe `FileList` in [DMT95]). Im Modell wird das mit der Aggregationsbeziehung zur Klasse **InstallElement** berücksichtigt.

Die Modellierung wird ebenso durch die bestehenden Möglichkeiten zur Manipulation von SW-Paketen verschiedener UNIX-Varianten gerechtfertigt. Unter AIX wird die beschriebene Managementinformation und -funktionalität durch das Administrationskommando `installp` vollständig abgedeckt und vom AIX-eigenen Systemmanagementwerkzeug *SMIT* bereitgestellt.

Möglicherweise sind die Attribute und Methoden der beiden Managementobjektklassen **Package** und **compObjTemplate** nicht ausreichend, um alle Szenarien des SW-Managements mittels einer Managementanwendung behandeln zu können. Außerdem fehlt die genaue Deklaration der zum Teil nicht elementaren Attributtypen. Einerseits würde es den Rahmen dieser Arbeit sprengen, Detailfragen des Anwendungsmanagements zu betrachten, da dies eine genaue Analyse der Anforderungen von Applikationen für das Anwendungsmanagement sowie die Verifikation, ob die geforderte Information und Funktionalität überhaupt von gängigen Systemen bzw. Software-Paketen geliefert wird, erfordern würde. Andererseits darf nicht erwartet werden, daß eine Anwendung, die auf den generischen Basisklassen arbeitet, die gleiche Funktionalität wie ein proprietäres Spezialwerkzeug bietet. Unbestreitbar lassen sich aber wichtige Basisklassen für das SW-Management aus dem RM-ODP ableiten.

Im Computational Model des RM-ODP werden verteilte Anwendungen und Systemdienste als Ansammlung von Computational Objects gesehen, die sich praktisch im „luftleeren Raum“ befinden. Ohne sich ihrer Verteilung bewußt zu sein, kooperieren die Objekte durch Interaktionen ihrer Schnittstellen. Das Modell versteckt alle Details einer darunterliegenden verteilten Plattform, auf der die Objekte zu Prozessen werden und die Interaktionen über Kommunikationskanäle, d.h. Verbindungen in einem Rechnernetz, abgewickelt werden müssen. Eine Sicht auf diese Konzepte bietet der Engineering Viewpoint.

4.2.2 Engineering Viewpoint

Die Konzepte des RM-ODP für den Engineering Viewpoints wurden in Kapitel 3.5.3 vorgestellt. Insbesondere beschreibt das Referenzmodell Infrastrukturobjekte einer offenen, verteilten Systemplattform, welche z.B. Kommunikationsmechanismen und Verteilungstransparenzen realisieren. Zu diesen Infrastrukturobjekten, welche Systemressourcen in einem verteilten System darstellen, werden im nächsten Schritt generische Managementobjektklassen eingeführt, die als Abstraktionen der Ressourcen für das Management dienen sollen. Da sich der Engineering Viewpoint mit den Laufzeiteigenschaften einer verteilten Systemplattform beschäftigt, liegt es nahe, hieraus Klassen für das Konfigurations-, Status- (*Monitoring*) und Fehlermanagement gewinnen zu können.

Templates für Capsule und Cluster

Zu einer Software-Komponente gehören eine oder mehrere ausführbare Dateien (*executables*). Dies wird im Objektmodell durch eine entsprechende Assoziation von der Klasse **compObj-Template** zu der neuen Klasse **capsuleTemplate** gekennzeichnet (siehe Abbildung 4.4). Laut RM-ODP muß das Capsule Template die zur Instantiierung eines Capsules benötigte Information, die aber nicht näher spezifiziert wird, enthalten. Die Instantiierung eines Capsules ist gleichbedeutend mit dem Starten eines Prozesses. Das korrekte Starten von Anwendungen oder Systemdiensten gehört zu den Aufgaben des Konfigurationsmanagements. Für die Klasse **capsuleTemplate** werden daher Attribute eingeführt, die dem Konfigurationsmanagement Informationen bereitstellen. Das Attribut `InstantProcedure` gibt Auskunft über Ort (z.B. Pfadangabe) und Name der ausführbaren Datei. Ein Executable besitzt einen Typ (`Type`), z.B. Binary oder Shellskript, der auch Auskunft darüber gibt, auf welchem Betriebssystem bzw. Prozessor es ausführbar ist. Über mögliche und sinnvolle Belegung von Aufrufparametern informiert das Attribut `RuntimeParameter`. `Size` gibt die Größe des Executables an. Weiterhin wird ein Attribut `RequiredObjects` modelliert, das angibt, welche Objekte zur Ausführung der Datei vorhanden sein müssen. Bei den Objekten kann es sich um Initalisierungs- oder Konfigurationsdateien, um benötigte Bibliotheken oder Systemschnittstellen etc. handeln. Die Informationen sind auch für das Fehlermanagement relevant. Arbeitet ein Systemdienst nicht ordnungsgemäß, kann der Administrator überprüfen, ob die zugehörigen Prozesse (Capsules) korrekt gestartet wurden und ob alle erforderlichen Objekte in der Umgebung vorhanden sind.

Zur Modellierung dieser Klasse mag kritisch angemerkt werden, daß die (wenigen) Attribute sehr vielfältige, nicht genau erfaßte Managementinformationen enthalten. Man ist versucht, die Information auf mehrere speziellere Attribute aufzuteilen. Hierbei besteht aber die Gefahr, daß man die Allgemeinheit der generischen Klasse aufgibt, weil man Spezifika eines bestimmten Systems oder einer Maschinenarchitektur einführt. Beispielsweise kann ein Capsule Template für ein Executable auf einem Host unter MVS nicht gleich aussehen wie ein Template für eine Batch-Datei unter MS-DOS. Für unterschiedliche Plattformen muß das Capsule Template daher noch weiter spezialisiert werden. Außerdem ist es in einem erweiterten Modell denkbar, das Attribut `RequiredObjects` durch Assoziationen («*requires*») zu anderen MOCs des Engineering Viewpoint zu ersetzen. Hier ein Beispiel für eine Instanz der generischen Klasse **capsuleTemplate** für den Hintergrundprozeß lpd des BSD-Druckdienstes:

Name: lpd

InstantProcedure: /usr/sbin/lpd

Type: ELF 32-bit LSB executable, Intel 80386, version 1, dynamically linked, stripped

RequiredObjects: /etc/printcap

RuntimeParameters: none

Size: 36984

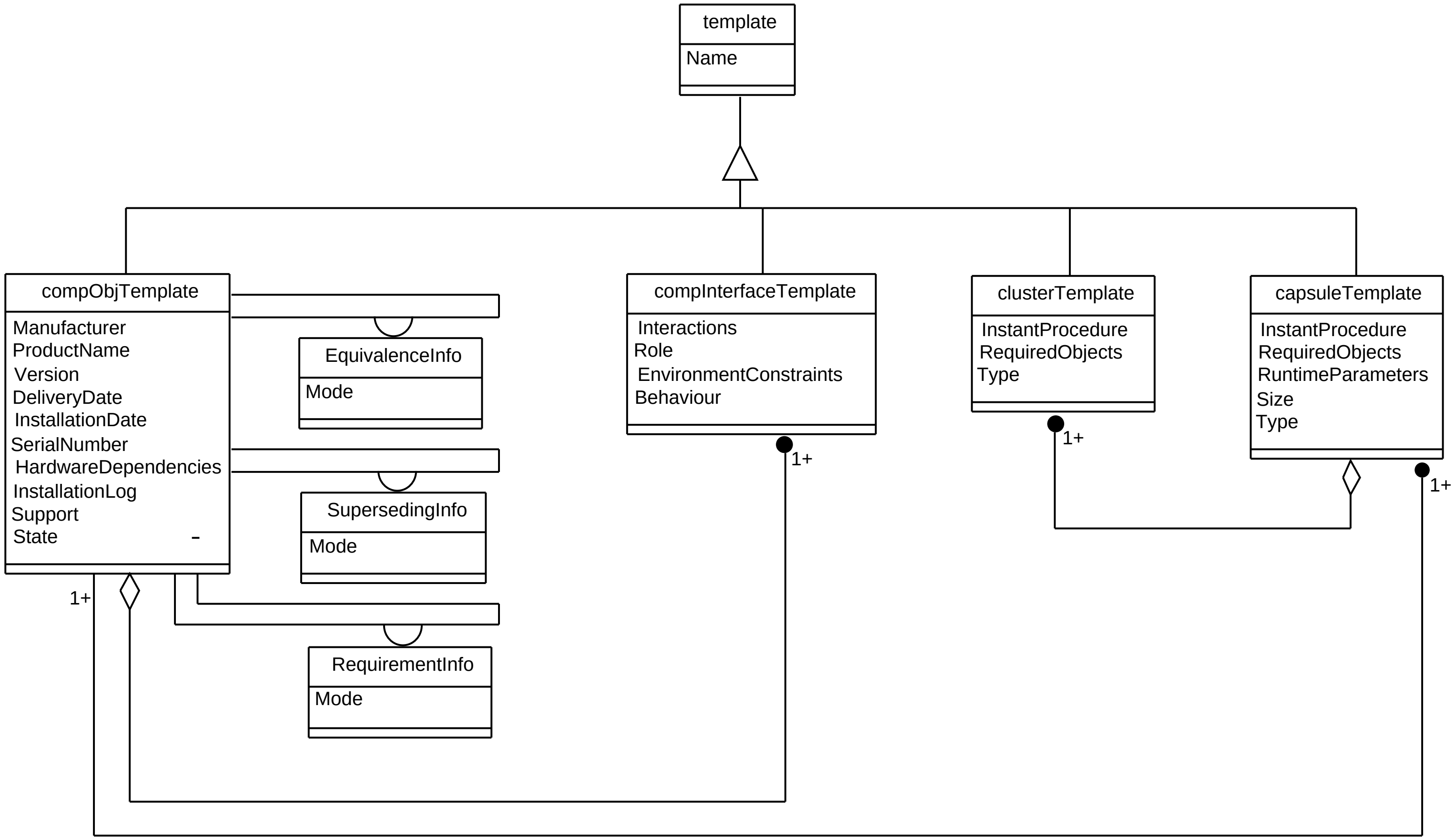


Abbildung 4.4: Managementobjektklassen zu Templates des Engineering Viewpoint

Weiterhin erhält die Klasse **capsuleTemplate** eine Aggregationsbeziehung zur Klasse **clusterTemplate**. Dies ist die logische Schlußfolgerung aus der Tatsache, daß im RM-ODP ein Capsule aus ein oder mehreren Clusters besteht. Managementinformation für die Klasse **clusterTemplate** ist analog zu **capsuleTemplate** definiert, wobei die Attribute `Size` und `RuntimeParameters` entfallen. Im obigen Beispiel wäre dem Capsule Template lpd eine Instanz eines Cluster Template für eine Bibliothek zugeordnet:

Name: libc.so

InstantProcedure: /lib/libc.so.5

Type: ELF 32-bit LSB shared object, Intel 80386, version 1, not stripped

RequiredObjects: ELF dynamic linker: ld-linux.so

An dieser Stelle ist ein kleiner Bruch mit dem RM-ODP auszumachen. Die Strukturregeln für Cluster (*cluster rules*) in [ISO95d] legen fest, daß ein Cluster nur in genau einem Capsule enthalten sein darf. Das Konzept der dynamischen, gemeinsam genutzten Bibliotheken (*dynamic shared libraries*), welches in vielen Systemen aus Performance-Gründen realisiert ist, verstößt aber gegen diese Regel. Da in dieser Arbeit aber versucht wird, die Konzepte des RM-ODP für das Management von real existierenden Systemen anzuwenden, welche selbst nicht nach dem RM-ODP entworfen und konstruiert wurden, wird eine Abweichung vom Standard hier und möglicherweise auch an anderer Stelle in Kauf genommen.

Im folgenden werden generische Managementobjektklassen zum Engineering Viewpoint definiert, die Abstraktionen von Systemressourcen zur Laufzeit darstellen. Aus diesen lassen sich später MOCs ableiten, die Managementinformation aus dem traditionellen Bereich des Systemmanagements bereitstellen. Damit können Managementanwendungen auf Systemressourcen wie Prozesse, Geräte und Kommunikationsverbindungen einwirken. Die Notwendigkeit hierfür ergibt sich u.a. aus dem erforderlichen Monitoring von Ressourcen für das Leistungs- und Fehlermanagement.

Capsule, Cluster und Basic Engineering Object

Wie bereits erwähnt, versteht das RM-ODP unter einem Capsule einen Prozeß im virtuellen Speicher eines Rechners. Die MOC **Capsule** ist somit die Managementabstraktion eines sich in Ausführung befindenden Prozesses unter einem beliebigen Betriebssystem. Die in der Klasse enthaltene Managementinformation ist so universell modelliert, damit **Capsule** als Vaterklasse für speziellere MOCs wie **UNIX_Process**, **OS/2_Process** oder **WinNT_Task** dienen kann. Hierunter fallen ein systemspezifischer Identifikator (`ID`), Angaben über Laufzeit (`ElapsedTime`) und Prozessorzeit (`CPUTime`) des Prozesses, der momentane Zustand (`State`), Menge des belegten RAM (`Memory`) und der Besitzer (`Owner`). Allgemeine Eingriffsmöglichkeiten auf Prozesse sollten das Stoppen und Wiedereinsetzen, das Senden von Signalen und das Beenden zulassen. Diese Eingriffe sind durch die Methoden `stop()`, `resume()`, `signal()` und `terminate()` vorgesehen.

Gemäß RM-ODP besteht ein Capsule aus ein oder mehreren Clusters und diese wiederum aus ein oder mehreren Basic Engineering Objects. Diese Enthaltenseinshierarchie spiegelt auch

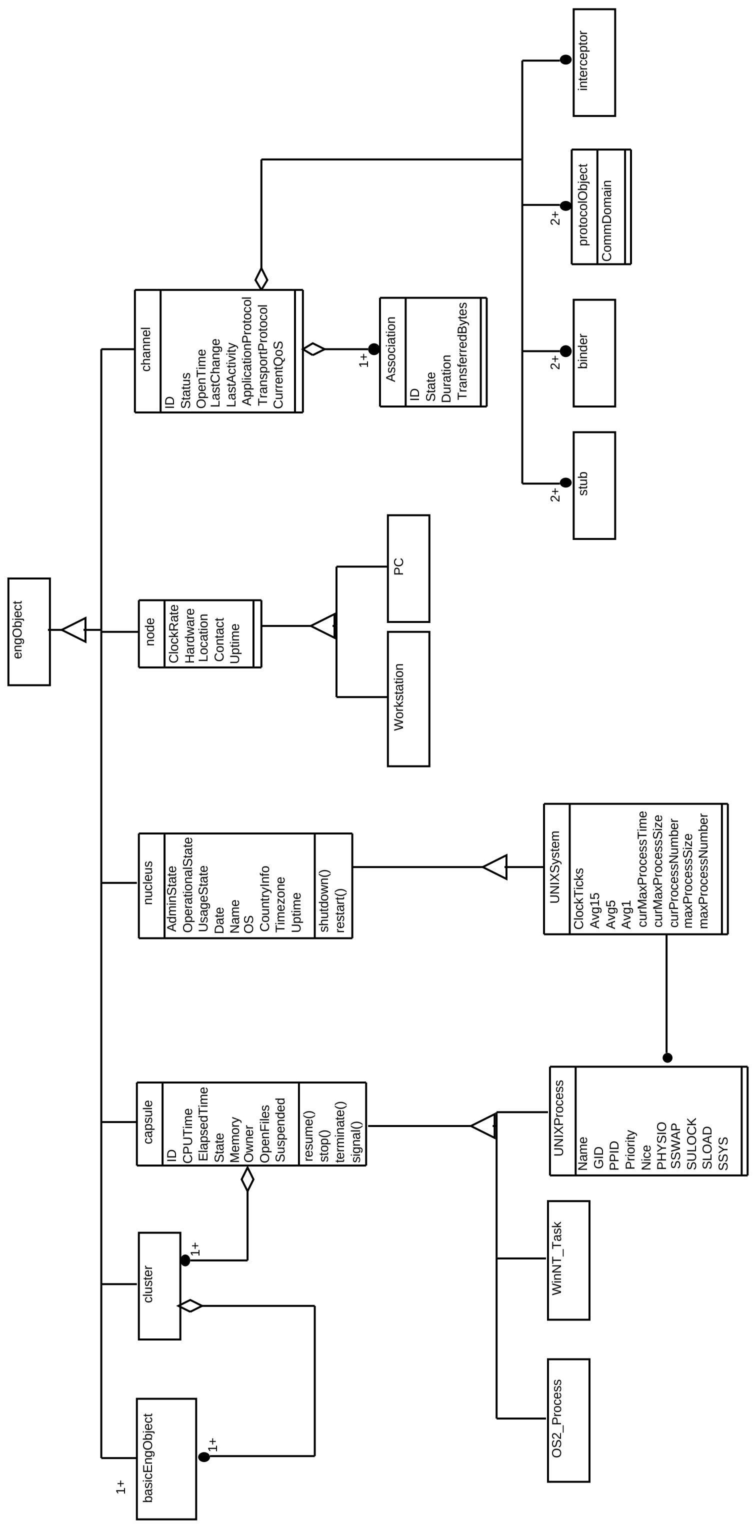


Abbildung 4.5: Managementobjektklassen zu Engineering Objects

das Objektmodell der MOCs durch Aggregationsbeziehungen wider. Die Abbildung 4.5 zeigt den entsprechenden Ausschnitt aus dem Objektmodell.

Nucleus

Die Managementobjektklasse **Nucleus** beinhaltet Information über das Betriebssystem als Ganzes eines Rechners. Sie dient als Vaterklasse für MOCs für bestimmte Betriebssysteme wie beispielsweise **OS/2_System**, **UNIX_System** oder **WinNT_System**. Ein Rechner wird innerhalb eines Netzes anhand seines logischen (Host-)Namens identifiziert (`Name`).

Den Status eines Systems kann man über die Attribute `AdminState`, `OperationalState` und `UsageState` auslesen (vgl. [ISO93, IBM93]). Da das Betriebsssystem alle Rechen-, Speicher- und Kommunikationsressourcen eines Rechners verwaltet und somit grundsätzlich verfügbar sein muß, ist der Operational State in allen Fällen «enabled». Der Usage State ist zu jeder Zeit «active». Im Normalfall muß der Admin State «unlocked» sein. Während eines Neustarts – `restart()` – oder eines Systemabschlußes – `shutdown()` – wechselt der Operational State zu «disabled» und der Admin State zu «shuttingDown» .

Weiterhin gibt die Klasse Auskunft über die Systemzeit (`Date`), über Name und Version des Betriebssystems (`OS`), über Ländereinstellungen (`CountryInfo`) und Zeitzone (`Timezone`) und über die Zeitspanne seit dem letzten Neustart (`Uptime`).

Node

Die Klasse **Node** ist die Hardware-Sicht auf einen Rechner, während **Nucleus** mehr das Betriebssystem erfaßt. Ein Node besteht physikalisch aus mehreren Hardware-Komponenten (*devices*) wie Prozessoren, Speicher, Festplatten, etc. Die eingeführten Attribute sind an sich selbsterklärend und lassen sich grob in die Gruppen «MaschineInfo» für Maschinenspezifika wie `Manufacturer`, `Model` und `Architecture` und «LocationInfo» aufteilen. `Location` könnte beispielsweise den Aufstellungsort nach Land, Stadt, Gebäude, Stockwerk, Raum, etc. aufschlüsseln. Den zuständigen Administrator enthält das Attribut `Contact`.

Channel

Die Interaktionen zwischen Computational Objects einer Anwendung werden auf der verteilten Plattform über Kanäle (*Channels*) zwischen den zugehörigen Basic Engineering Objects abgewickelt. Die Managementobjektklasse **channel** ist daher die Abstraktion einer Kommunikationsverbindung innerhalb einer verteilten Anwendung oder eines verteilten Systemdienstes.

Die interne Struktur eines Channels nach dem RM-ODP wird durch die Aggregationsbeziehungen zu den MOCs **stub**, **binder**, **protocolObject** und **interceptor** abgebildet. Im einfachsten Fall einer Punkt-zu-Punkt-Verbindung besteht ein Channel aus genau zwei Stubs, zwei Binders und zwei Protocol Objects, bei Multicast-Verbindungen aus entsprechend mehreren. Befinden sich die kommunizierenden Objekte in unterschiedlichen administrativen, organisatorischen oder technischen Domänen, kommen noch ein oder mehrere Interceptors hinzu.

Stubs, Binders, Protocol Objects, etc. werden im RM-ODP eingeführt, um eine Kommunikationsarchitektur zu schaffen, die verschiedene Transparenzen, wie z.B. Ortstransparenz, für die verteilten Objekte ermöglicht. CORBA ist beispielsweise eine Realisierung einer solchen Architektur. Da der Standard aber unabhängig von konkreten Systemen ist, wird auch nicht spezifiziert, *wie* diese Transparenzen realisiert werden bzw. *welche* Funktionen die Objekte implementieren. Darüber hinaus handelt es sich bei einem Channel um ein sehr allgemeines Konzept, mit dem sowohl Interaktionen wie RPCs als auch multimediale Datenströme beispielsweise innerhalb eines Konferenzsystems beschrieben werden können. Daher ist es schwierig, den beschriebenen Objektklassen konkrete Managementinformation zuzuordnen, ohne die Allgemeinheit zu beschränken. Zur Modellierung der Klassen für Binder, Protocol Object und Interceptor wären außerdem Aspekte des Netzmanagements zu betrachten, was über den Rahmen dieser Arbeit hinausgeht.

Das Attribut `CommDomain` in der Klasse **protocolObject** gibt die Kommunikationsdomäne – z.B. «OSI» oder «Internet» – an, in welcher das Protocol Object etabliert ist. Innerhalb einer Kommunikationsdomäne können Protocol Objects ohne dazwischenliegendes Gateway (Interceptor) miteinander kommunizieren.

Die Attribute, die im folgenden für die Klasse **channel** eingeführt werden, beschreiben Managementinformation, die auf Kommunikationsverbindungen beliebiger Art angewendet werden kann. Jeder Channel besitzt einen eindeutigen Identifikator (`ID`), den Zeitpunkt, an dem er kreiert wurde (`OpenTime`) und einen Zustand (`Status`). Mögliche Zustände für einen Kanal könnten «up», «down», «congested» oder «unknown» sein. Bei der Einrichtung eines Kanals wird eine bestimmte Dienstgüte gefordert, die durch die Environment Constraints des zugehörigen Computational Interface Templates festgelegt ist. In der Praxis ist die Dienstgüte bei Verbindungsaufbau aber Gegenstand von Verhandlungen. Das bedeutet, daß auch Kanäle eingerichtet werden, die nicht die geforderte Dienstgüte erfüllen. Die tatsächliche realisierte Dienstgüte eines Kanals kann über das Attribut `CurrentQoS` abgefragt werden. Den Zeitpunkt der letzten Konfigurationsänderung eines Kanals liefert das Attribut `LastChange`, der Zeitpunkt der letzten Aktivität wird in `LastActivity` festgehalten. Die Attribute `ApplicationProtocol` und `TransportProtocol` geben Auskunft über die eingesetzten Protokolle der Anwendungs- bzw. Transportschicht.

An einen Channel können mehrere Basic Engineering Objects gebunden sein. Zur Identifizierung der Kommunikationspartner dienen dann sog. *Binding Endpoint Identifier*, die innerhalb eines Prozesses (Capsule) Gültigkeit haben und nicht mit den Engineering Object References verwechselt werden dürfen. Es ist also auf Anwendungsebene ein Multiplexen von Verbindungen (*associations*) über einen Kanal möglich. Dies wird durch die Aggregationsbeziehung zur Klasse **Association** modelliert. Mögliche Attribute für diese Klasse sind `ID`, `Duration`, `State` und `TransferredBytes`, deren Sinn sich aus dem Namen ergibt.

Die generischen Klassen **channel** und **Association** sollen ein Monitoring von Kommunikationsverbindungen innerhalb verteilter Anwendungen zum Zwecke des Fehler-, Leistungs- und Abrechnungsmanagements ermöglichen. Die Modellierung dieser beiden Klassen greift daher Vorschläge der *Network Services Monitoring MIB* [KF94] auf (siehe Kapitel 3.7.2).

4.2.3 Beziehungen zwischen den Klassen des Computational und Engineering Viewpoint

RM-ODP definiert die Abhängigkeiten zwischen den Konzepten verschiedener Sichten durch Strukturregeln der sog. *Viewpoint Correspondences*. An dieser Stelle wird kurz auf die Beziehungen zwischen Computational und Engineering Viewpoint eingegangen, welche auch die Managementobjektklassen des Modells wiedergeben sollen.

Ein Computational Object wird durch mehrere Basic Engineering Objects realisiert. Hierfür gibt es im Modell eine 1:n-Assoziation zwischen der Klasse **compObject** und **basicEngObject**. Da die Basic Engineering Objects wiederum in Capsules enthalten sind, kann das Management über die Relationen zwischen den MOCs die Prozesse zu einer Software-Komponente identifizieren.

Weiterhin legt das Referenzmodell fest, daß ein Computational Interface durch genau ein Engineering Interface realisiert wird. Zur Adressierung eines Engineering Interface bei der Einrichtung eines Kanals dient die eindeutige Engineering Interface Reference. Innerhalb der TCP/IP-Welt kann z.B. ein Tupel bestehend aus IP-Adresse und Port (*socket* bezüglich eines Transportprotokolls (TCP oder UDP) als Engineering Interface Reference betrachtet werden. In einer CORBA-Umgebung dienen hierzu die IORs der CORBA-Objekte. Das Objektmodell sieht folglich eine von **interface** vererbte Klasse **engInterface** vor, die das Attribut `Reference` besitzt und eine 1:1-Beziehung zu **compInterface** eingeht.

RM-ODP erlaubt, daß sich mehrere Basic Engineering Objects an einen Kanal und damit an ein Engineering Interface binden. Zur Unterscheidung von Kommunikationsverbindungen einzelner Basic Engineering Objects, die über den gleichen Kanal abgewickelt werden, dient der sog. *Binding Endpoint Identifier*. Dieser ist nur innerhalb des Basic Engineering Objects gültig. Daher wird das Attribut `BindingEndpointID` der Klasse **basicEngObject** zugeordnet. In der TCP/IP-Welt ist eine Referenz auf einen Socket (File-Deskriptor) ein Beispiel für einen Binding Endpoint Identifier.

4.3 Integration eines bestehenden Modells

Im folgenden Abschnitt wird beschrieben, wie ein vorhandenes Objektmodell für das integrierte Management von UNIX-Workstations durch die Wahl einer geeigneten Spezialisierungshierarchie in das in diesem Kapitel vorgestellte, auf ODP-Konzepten basierende, generische Objektmodell eingegliedert werden kann. Bisher wurden in einer Top-Down-Vorgehensweise aus den Viewpoints des RM-ODP unter Beachtung der Anforderungen aus den funktionalen Dimensionen des Managements generische Klassen herausgearbeitet, die als Basisklassen für das Management von Endsystemressourcen und verteilten Systemdiensten dienen sollen. Hierbei wurden noch keine bestimmten Systeme betrachtet. Im nächsten Schritt soll ein spezielles Objektmodell, welches sich durch entsprechende Implementierungen als geeignet für das Management von Systemressourcen eines heterogenen UNIX-Workstation-Clusters erwiesen hat, durch Spezialisierung an die Basisklassen angebunden werden. Dabei handelt es sich um ein Bottom-Up-Vorgehen, da anhand der gegebenen Managementinformation einer speziellen Ressource überlegt wird, welcher Anteil davon generell auf Ressourcen dieser Klasse angewendet werden kann. Gleichzeitig wird durch die vorhandene Implementierung sichergestellt, daß die modellierte Information auch tatsächlich durch die Ressource bereitgestellt wird. Wenn dieser Schritt gelingt, ist dadurch einerseits die Einführung der generischen Klassen gerechtfertigt und andererseits die angestrebte Anwendbarkeit der gewonnenen Managementinformation auch auf andere Systeme gegeben.

Holger Sirtl hat die Managementinformation der MNM-UNIX-MIB (siehe Abschnitt 3.7.2) in seiner Arbeit „Objektorientierte Modellierung von Workstations für ein integriertes Systemmanagement" [Sir96] in ein OMT-Objektmodell überführt. Hierbei wurde nach folgendem Algorithmus vorgegangen. Für jede Gruppe der SNMP-MIB wurde eine Objektklasse im OMT-Modell kreiert. Die einfachen Datentypen einer SNMP-Gruppe wurden zu Attributen der betreffenden Objektklasse. Zum Beispiel wurde die Gruppe «System» mit den Objekttypen `sysName` und `sysLocation` zur Klasse **System** mit den Attributen `Name` und `Location`. SNMP-Tabellen wurden ebenfalls in Objektklassen überführt. Jede Zeile einer SNMP-Tabelle entspricht somit einer Instanz der Objektklasse. Die Variablen einer SNMP-Tabelle wurden wiederum zu Attributen der Objektklasse. Beispielsweise besitzt die SNMP-Tabelle, die die Prozeßtabelle der Workstation darstellt, u.a. die Felder `processPID` und `processSize`. Im OMT-Modell wird dagegen die Prozeßtabelle als Menge von Objektinstanzen einer Objektklasse **Process** mit den Attributen `ID` und `Size` modelliert. Natürlich generiert dieser Algorithmus kein echtes objektorientiertes Modell. Es fehlen Konzepte wie Aggregation, Spezialisierung (Vererbung) oder Methoden. Deshalb wurde das Modell manuell weiterbearbeitet. Vererbungshierachien wurden gebildet, indem gleiche Attribute in den aus den Gruppen erzeugten Objektklassen in eine Oberklasse geschoben wurden. Somit erben z.B. alle MOCs für spezielle Geräte die Statusattribute `UsageState`, `AdminState` und `AvailState` aus der gemeinsamen Oberklasse **Device**. Attribute, die aus sog. „Pushbutton-Variablen" entstanden sind, das heißt, die durch Belegen des Attributs mit einem bestimmten Wert eine Aktion in der Ressource auslösen, wurden durch Methoden ersetzt. Die Methoden `disable()` und `enable()` der Klasse **Device** ersetzen somit die Zuweisung des Wertes «0» bzw. «1» an die bei vielen Geräten vorhandenen Variable `action`.

Das hieraus entstandende Modell zeigt die Abbildung A.4 im Anhang A. Im Mittelpunkt steht die zentrale Klasse **System** mit sternförmigen Aggregationsbeziehungen zu diversen Spezialisierungen der Klasse **Device**. Diese Modellierung bildet die Realität in der Hinsicht korrekt ab, daß eine Workstation aus verschiedenen Geräten wie Prozessor, Arbeitsspeicher, Festplatten etc. aufgebaut ist. Die Assoziation zwischen **System** und **Process** modelliert die von einer Workstation ausgeführten Prozesse. Aktive Benutzer einer Workstation werden durch die Beziehungsklasse **ActiveUser** zwischen **System** und der Klasse **Account** dargestellt. Die bekannte UNIX-Dateisystemhierarchie wird durch Instanzen der Klassen **Filesystem** – als Oberklasse für verschiedene Typen von Filesystemen – und **MountPoint** aufgebaut.

Das vorhandene Modell kann ohne Schwierigkeiten an die generischen ODP-Basisklassen angebunden werden. Nach dem Engineering Viewpoint verwaltet das Nucleus-Objekt eines Node – also das Betriebssystem eines Rechners – alle Rechen-, Speicher- und Kommunikationsressourcen. Die Komponenten einer Workstation, die die Ressourcen zur Verfügung stellen, werden im ODP-Modell vollständig verschattet. Natürlich sind die Objektklassen, die ein aktives Management einzelner Systemkomponenten erlauben, für das Systemmanagement wichtig. Aus diesem Grund wird die Vererbungshierarchie ausgehend von der Klasse **Device** zunächst unverändert in das neue Modell übernommen.

Die Klasse **System** des alten Modells enthält sowohl Managementinformation über die Workstation selbst als auch über das auf ihr laufende Betriebssystem. Informationen über die Maschine, wie z.B. die Typbezeichnung (`Hardware`), der Aufstellungsort (`Location`) und die Kontaktperson werden im neuen Modell durch die Klasse **node** bereitgestellt, die gegebenenfalls für eine UNIX-Workstation durch eine Klasse **Workstation** verfeinert werden kann. Diese Attribute werden also aus der Klasse **System** entfernt. Die Managementinformation, die das Betriebssystem betrifft, verbleibt in der Klasse **System**. Diese Klasse wird in **UNIXSystem** umbenannt und kann nun von der generischen Klasse **nucleus** abgeleitet werden. Die Attribute, die bereits in der Klasse **nucleus** enthalten sind und folglich generell für alle Klassen von Betriebssystemen gelten, werden natürlich aus **UNIXSystem** entfernt, da sie von **nucleus** ererbt werden. Hierunter fallen zum Beispiel das Systemdatum `Date`, die Betriebssystemversion `OS` und der logische Rechnername `Name`. Hier wird nochmals die Vorgehensweise zur Gewinnung allgemeiner, generischer Managementinformation und -funktionalität deutlich. Für alle Attribute und Methoden einer Managementobjektklasse, die eine spezielle Ressource beschreibt, wird die Frage gestellt, ob sie generell auf alle Klassen dieser Ressource anwendbar sind. In diesem Fall können sie in eine generische Oberklasse übernommen werden.

Es könnte die Frage entstehen, warum die MOCs, die Geräte einer Workstation repräsentieren, eine *part-of*-Beziehung mit der Klasse **UNIXSystem** eingehen und nicht mit der Klasse **Workstation**. Dies liegt daran, daß die Klassen die logische Sicht des Betriebssystem auf das Gerät liefern und nicht die Sicht auf die Hardware-Komponenten selbst. Bei der Klasse **Processor** gibt `Name` den logischen Device-Namen, z.B. «/dev/p0», und `SystemTime` den Anteil der Betriebssystemprozesse an der genutzten Prozessorzeit an. Für eine Klasse **RISC_Processor** als Bestandteil (*part of*) der Klasse **Workstation** wären jetzt Hardware-Attribute wie Oberflächentemperatur (`SurfaceTemperature`), Lüftergeschwindigkeit (`VentilatorSpeed`) oder Sockelbezeichnung (`SocketDesc`) denkbar. Von solchen MOCs wird im Objektmodell aber abgesehen, da diese Art von Managementinformation i.a. nicht mit Mitteln des Betriebssystems erlangt werden kann. Eine Implementierung eines Agenten hierfür würde zusätzliche spezielle Meß- und Steuer-Hardware, wie z.B. einen Temperaturfühler in obigem Beispiel, erfordern.

Analog zur Klasse **System** kann die Klasse **Process** im neuen Modell von der generischen ODP-Klasse **capsule** abgeleitet werden. Hierzu wird sie in **UNIXProcess** umbenannt, da sich von **capsule** auch Klassen wie **OS/2_Process** oder **WinNT_Task** ableiten lassen. In jedem Betriebssystem lassen sich einem Prozeß ein eindeutiger Identifikator, ein Benutzer, genutzte Prozessorzeit, ein Zustand, etc. zuordnen. Auch können Prozesse i.a. gestoppt oder beendet werden. Diese Attribute und Methoden vererbt **capsule** im Modell an **UNIXProcess**, die nur noch UNIX-spezifische Attribute wie `GID`, `Priority` etc. enthält. Die angesprochenen Vererbungshierarchien zeigt die Abbildung 4.5 auf Seite 72. Das angepaßte Modell ist in den Abbildungen 4.6 und 4.7 dargestellt.

Weitere Klassen des übernommenen Modells lassen sich nicht von ODP-Basisklassen ableiten. Dies liegt daran, daß der Fokus beim bestehenden Modell auf das Management von lokalen Systemressourcen einer einzelnen Workstation gesetzt war. Dies ist auch der klassische Bereich des Systemmanagements, der die Verwaltung einzelner, nicht vernetzter Rechner umfaßt. Bei modernen Systemlandschaften handelt es sich mittlerweile aber fast ausnahmslos um Rechner, die über ein leistungsfähiges Netz miteinander verbunden sind und deren Ressourcen als einziges großes verteiltes System allen Benutzern zur Verfügung stehen. Da ein verteiltes System ein Kommunikationssystem voraussetzt, ist eine Abgrenzung des Netzmanagements vom

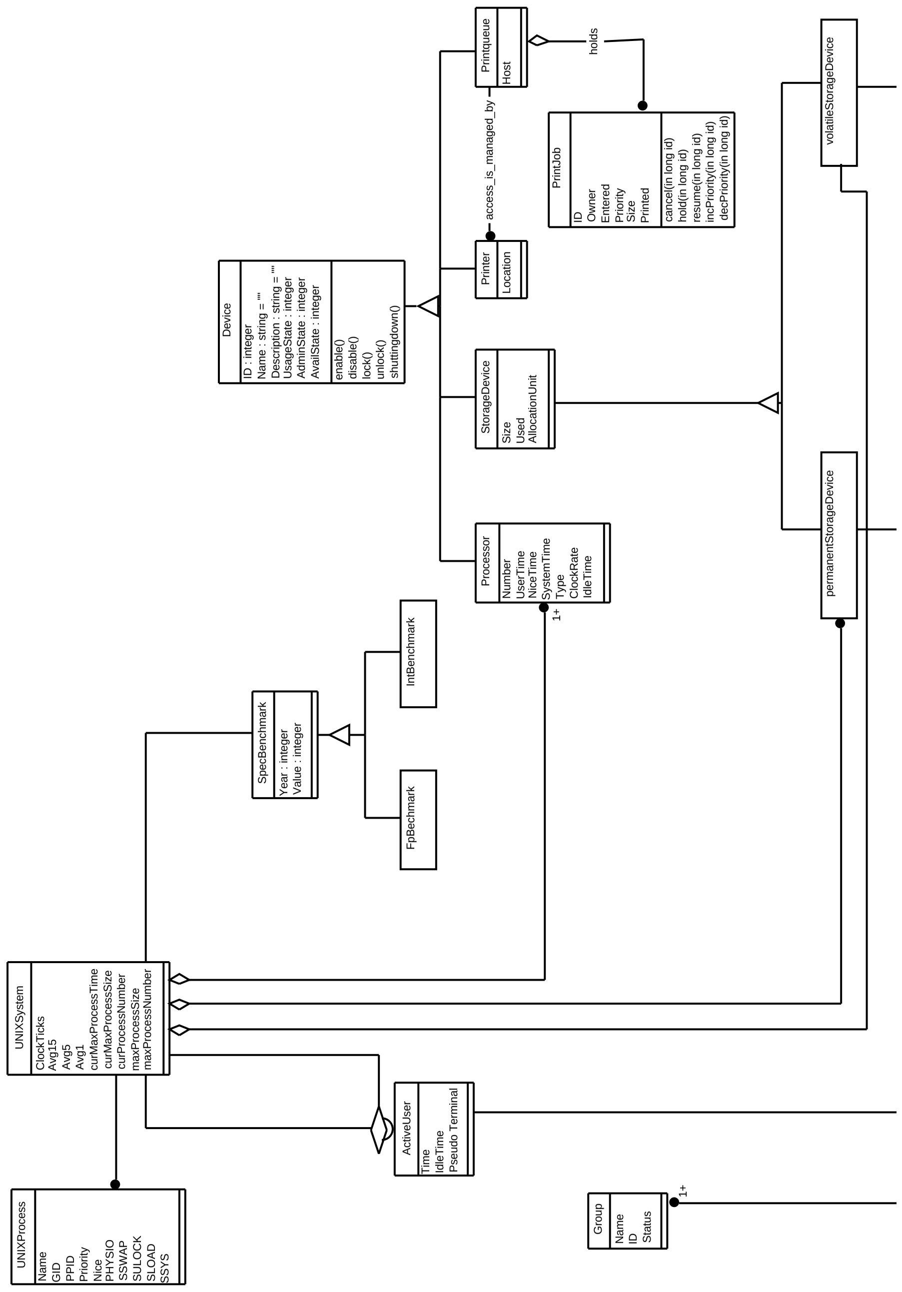


Abbildung 4.6: Objektmodell für das Systemmanagement von UNIX-Workstations – Teil 1

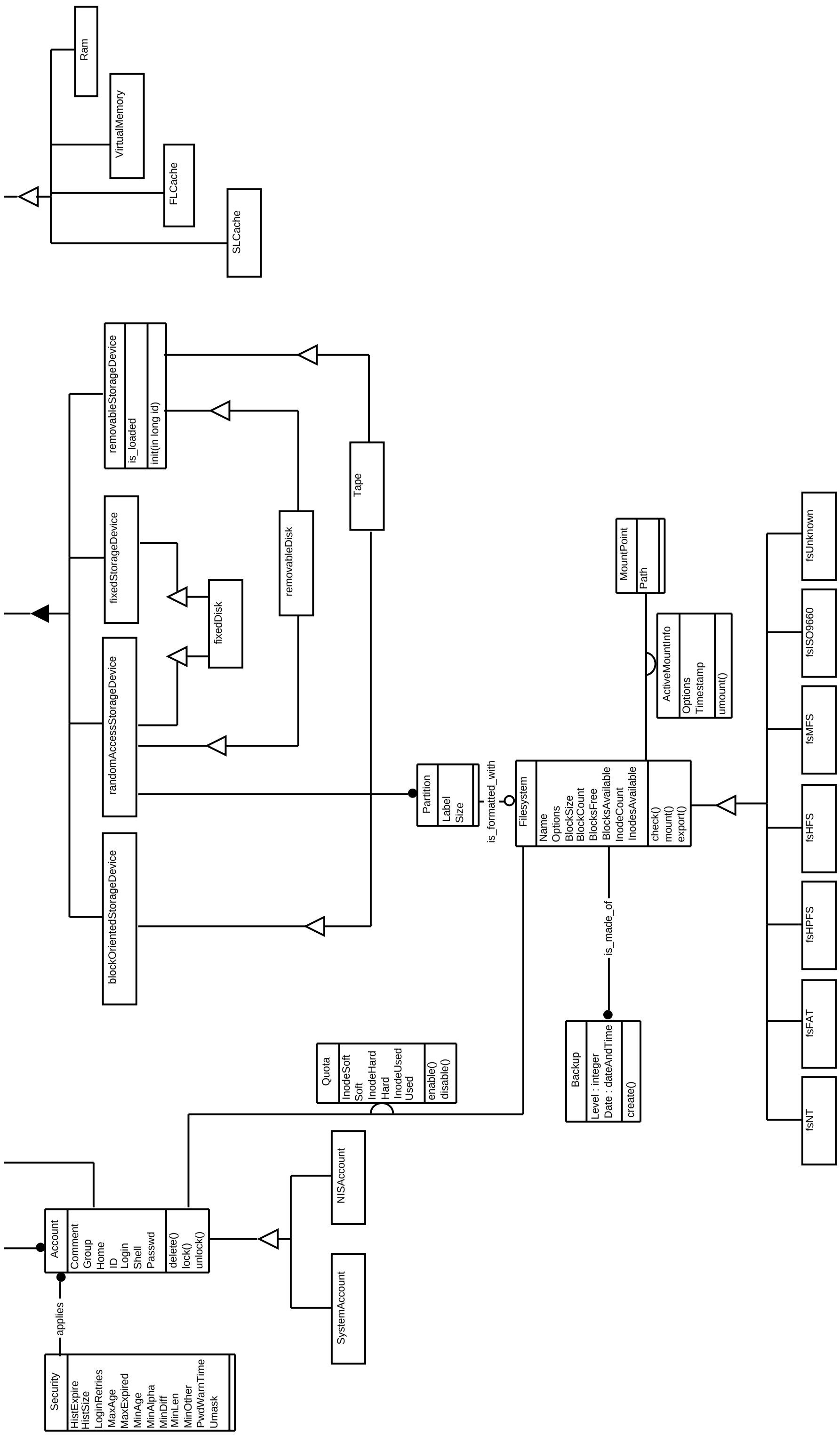


Abbildung 4.7: Objektmodell für das Systemmanagement von UNIX-Workstations – Teil 2

Systemmanagement nicht mehr möglich bzw. sinnvoll. Einen immer größer werdenden Stellenwert im Systemmanagement nimmt aber die Verwaltung und Administration der verteilten Systemdienste ein. Hierzu sind auch Konzepte des Anwendungsmanagement erforderlich.

Im folgenden Kapitel wird das Objektmodell durch generische Klassen für das Management von verteilten Systemdiensten erweitert und gleichzeitig gezeigt, daß sich die eingeführten ODP-Basisklassen auf eine Vielzahl von Anwendungen und Diensten anwenden lassen.

4.4 Generische Klassen für verteilte Systemdienste

Die korrekte Funktion der Systemdienste ist zentrale Voraussetzung für Anwendungen in einem verteilten System. Hieraus ergibt sich auch sofort die Notwendigkeit für das Management von Systemdiensten, welches zum Systemmanagement gezählt werden kann, da die Systemdienste die Funktionen des Betriebssystems erweitern. Wie bereits erwähnt, können nicht nur die Dienstnutzer, sondern auch die Server, die einen Systemdienst anbieten, auf mehrere Rechner in einem Netz verteilt sein. Dadurch kann ein Systemdienst als verteilte Anwendung aufgefaßt werden, dessen Management mehr von den Konzepten des Anwendungsmanagements geprägt sein wird. Die Grenzen zwischen System- und Anwendungsmanagement verschwimmen also zunehmend. Dies rechtfertigt zudem auch die Basisklassen des RM-ODP im Objektmodell für das Systemmanagement.

Im folgenden werden Managementobjektklassen für die Objekte «Client» und «Server» eines Systemdienstes modelliert. Diese Klassen sollen möglichst generisch sein, d.h. ein effizientes Management möglichst vieler verschiedener Systemdienste erlauben. Die definierten Klassen sollen also unabhängig von einem konkreten Dienst sein. Dies soll aber eine Spezialisierung oder Erweiterung der generischen Klasse für ein Werkzeug zum Management eines konkreten Dienstes nicht ausschließen.

Markus Gutschmidt zerlegt in seiner Dissertation „Ein Objektmodell für ein integriertes Management von Systemdiensten mit Client/Server-Struktur“ [Gut95] einen Server in die funktionalen Komponenten «Funktionseinheit», «Bedienstation» und «Bedieneinheit», «Warteschlange» und «Auftrag». Die Zerlegung und Begriffsbildung geht auf die Verkehrstheorie zurück. Zu diesen Komponenten definiert er entsprechende Managementobjektklassen mit den folgenden Beziehungen. Ein Server ist eine Funktionseinheit, die aus mindestens einer Bedienstation besteht. Eine Bedienstation kann aus mehreren Bedieneinheiten aufgebaut sein. Dies entspricht einem Server, der parallel Aufträge über mehrere Prozesse oder Threads bearbeiten kann. Weiterhin kann ein Server keine, genau eine oder mehrere Warteschlangen für Aufträge besitzen und keinen, genau einen oder mehrere Aufträge enthalten.

In dieser Arbeit wird versucht, einen Systemdienst und seine zugehörigen *managed objects* mit den Konzepten und Sichten des RM-ODP zu modellieren. Daher handelt es sich bei den Modulen «Client» und «Server» eines verteilten Dienstes um Software-Komponenten und damit zur Laufzeit um Computational Objects. Folglich werden die MOCs **Client** und **Server** von der Klasse **compObject** abgeleitet. Ein Server stellt seinen Dienst an einer oder mehreren Schnittstellen (Computational Interfaces) zur Verfügung. Systemseitig realisiert wird ein Dienst durch Prozeduren (Basic Engineering Objects) innerhalb eines oder mehrerer Prozesse (Capsules).

Markus Gutschmidt ordnet den Komponenten Bedienstation/Bedieneinheit, Warteschlange und Auftrag eigene Managementinformation und -funktionalität zu. Hier wird die Information und Funktionalität in der Klasse **Server** bzw. **Client** zusammengefaßt. Diese Vereinfachung wird dadurch gerechtfertigt, daß die meisten der derzeit etablierten und verbreiteten Systemdienste keine Managementschnittstellen für Einzelkomponenten ihrer Client- und Servermodule beinhalten. In den später untersuchten Diensten NFS und NIS ist beispielsweise ein Management der Warteschlangen oder einzelner Aufträge gar nicht möglich. Weiterhin soll es sich bei den Klassen **Server** und **Client** um generische Basisklassen handeln. Konkrete Server und Clients, die ein Management ihrer Komponenten erlauben, können durch Unterklassen von **Server** und **Client** mit Aggregationsbeziehungen zu neuen Managementobjektklassen für ihre Komponenten modelliert werden.

Das hier beschriebene Modell eines Servers wird in Abbildung 4.8 verdeutlicht. Die obere Hälfte zeigt den Computational Viewpoint. Das Computational Object «Server» besitzt eine Dienst- und eine Managementschnittstelle. Ein Client nutzt den Dienst durch Binden der Server-Schnittstelle. Eine Managementanwendung überwacht über das Computational Object «Manager» den Server. In der unteren Hälfte wird der Engineering Viewpoint gegenübergestellt. Client und Server sind jeweils durch einen Prozeß (Capsule) realisiert und kommunizieren über einen Kanal. Der Agent, der dem Manager Informationen über den Server bereitstellt, ist in diesem Fall ebenfalls ein eigener Prozeß, der auf dem gleichen System wie der Server läuft und somit über eine lokale Schnittstelle (*local binding*) mit dem Server kommuniziert. Der Agent kann weitere Schnittstellen zu anderen Managed Objects besitzen. Unter der Annahme, daß die Managementanwendung auf einem anderen Rechner läuft, tauschen Agent und Manager Informationen wiederum über einen Kanal aus.

Bei der Modellierung der Klassen **Server** und **Client** stehen die Betriebsaspekte Steuerung und Überwachung eines Dienstes im Vordergrund. Die Managementinformation und Funktionalität wird, wie in Kapitel 2.1.4 erläutert, durch Analyse der Anforderungen aus den folgenden Funktionsbereichen definiert:

- Leistungsmanagement
- Fehlermanagement
- Sicherheitsmanagement
- Statusmanagement
- Auftragsmanagement
- Konfigurationsmanagement

4.4.1 Server

Es folgt zunächst die Beschreibung der generischen Klasse **Server**, da sich herausstellen wird, daß die Klasse **Client** eine Untermenge der Attribute und Methoden von **Server** enthält. Die Attribute und Methoden für die Klasse **Server** werden im folgenden durch Analyse der Anforderungen aus den oben genannten Managementfunktionsbereichen bestimmt. Das im weiteren beschriebene Modell findet sich in Abbildung 4.9.

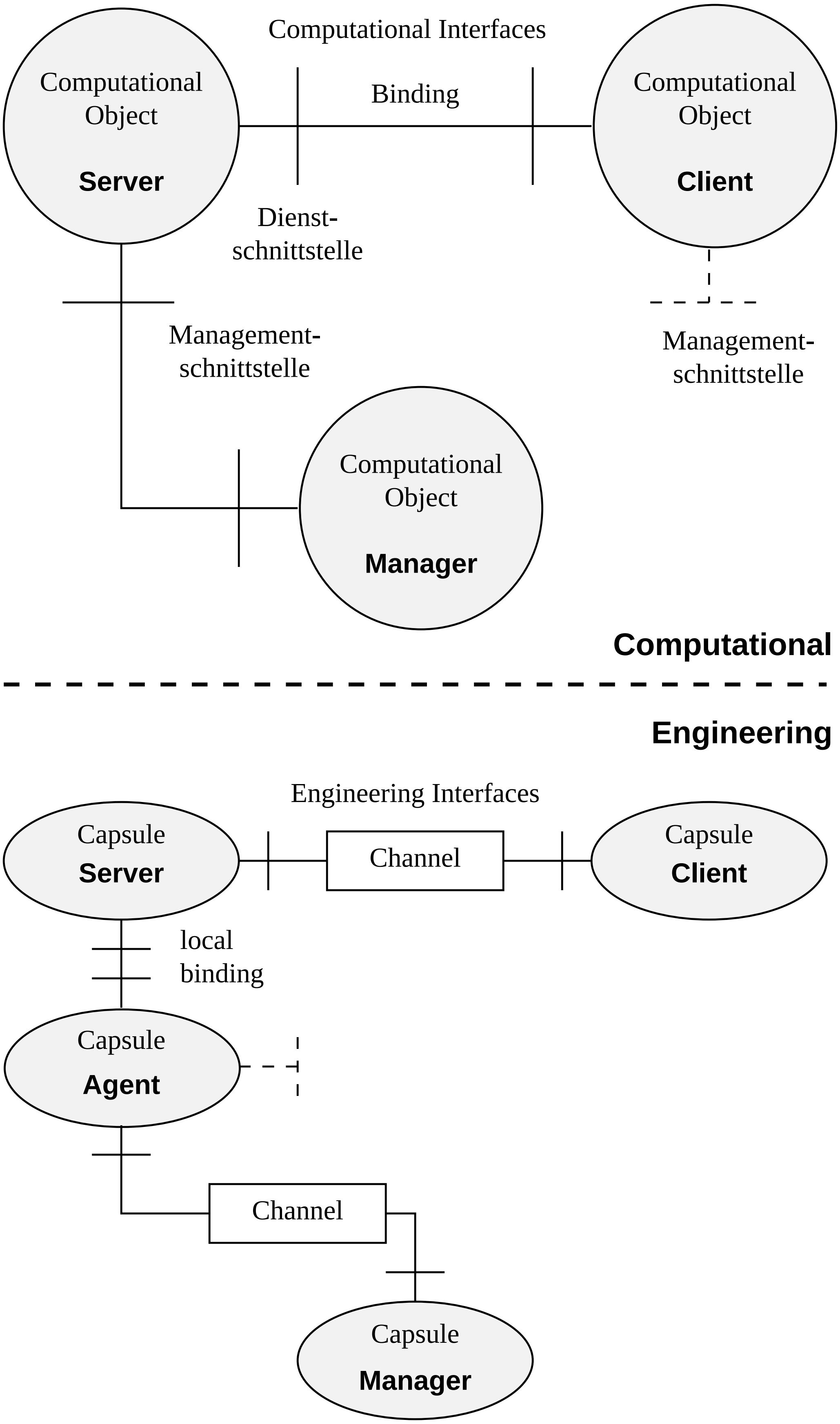


Abbildung 4.8: RM-ODP-Modell eines Servers mit Managementschnittstelle

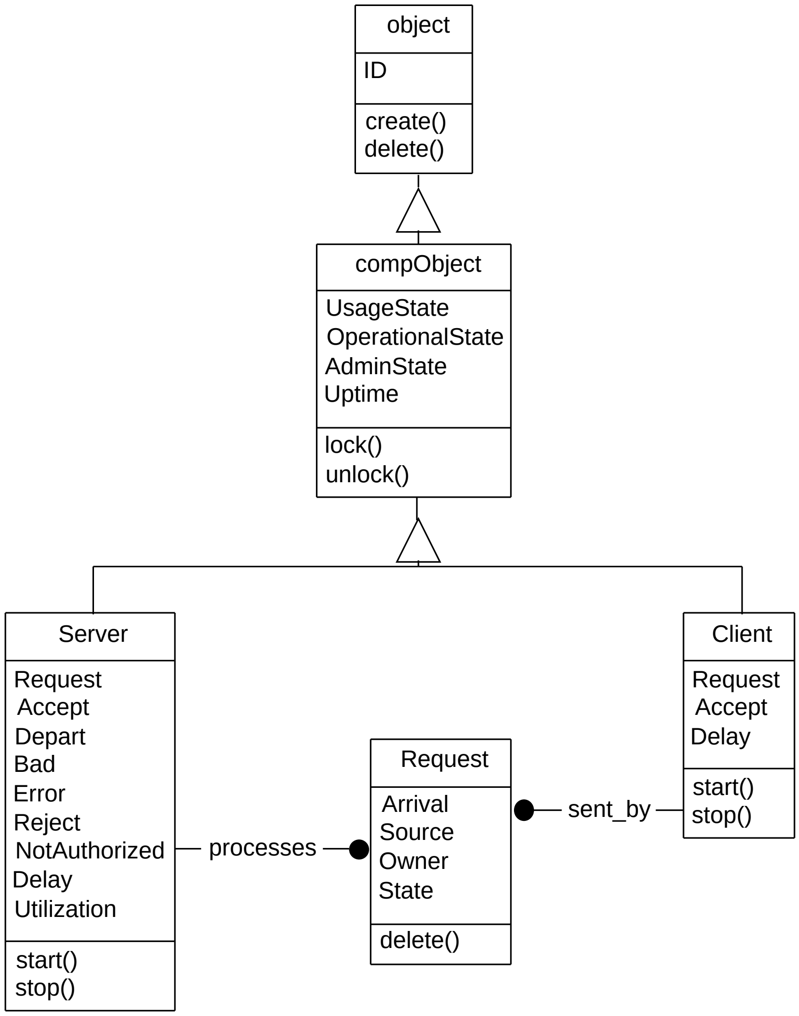


Abbildung 4.9: Generische Klassen für das Management von Systemdiensten

Leistungsmanagement

Aufgabe des Leistungsmanagements ist das Sammeln und Auswerten von Meßdaten für die Dienstgüte (QoS) wie z.B. Durchsatz, Auslastung und Antwortzeit. Diese Größen müssen zur Laufzeit für einen ordentlichen Betrieb des Dienstes überwacht werden, aber auch für eine langfristige Planung über einen längeren Zeitraum statistisch ausgewertet werden.

Die für die Auswertung der gesammelten Daten benötigte Funktionalität, wie z.B. Schwellwertüberwachung, Mittelwertbildung etc., sollte einer Anwendung von der Managementplattform zur Verfügung gestellt werden, damit sie bei der Modellierung der Ressourcen nicht redundant definiert wird. Aus diesem Grund enthält die Klasse **Server** keine entsprechenden Methoden.

Die Definition der folgenden Attribute geht auf [Gut95] bzw. [Neu93] zurück. Alle Zähler beziehen sich dabei auf ein Intervall $[t_0, t[$, wobei t den aktuellen Zeitpunkt bezeichnet und t_0 den Zeitpunkt des Starten des Servers bzw. den letzten Rücksetzzeitpunkt der Zähler. Es gilt $t > t_0$.

- Lastüberwachung

  Request: Zähler für die Gesamtanzahl der im Intervall $[t_0, t[$ beim Server eingetroffenen Aufträge.

  Reject: Zähler für die Aufträge, die der Server im Intervall $[t_0, t[$ zwecks Mangel an Ressourcen (z.B. Überlauf einer Warteschlange) nicht bearbeiten konnte.

  Accept: Zähler für die im Intervall $[t_0, t[$ eingetroffenen und akzeptierten Aufträge. Es gilt: $Accept = Request - Reject$. Dies schließt nicht aus, daß ein Auftrag vom Server zunächst akzeptiert wird, aber nach einer Prüfung, z.B. wegen fehlender Berechtigung des Clients, verworfen wird.

- Füllung[1] und Durchsatzüberwachung

  Depart: Zähler für die Aufträge, dessen Bearbeitung vom Server im Intervall $[t_0, t[$ beendet wurde. Dies ist die Summe der korrekt bearbeiteten und der nach Prüfung abgelehnten Aufträge.

  Utilization: Pegel für die Füllung, also für die Anzahl der sich zum Zeitpunkt t im Server befindenden Aufträge. Es gilt: $Utilization = Accept - Depart$. Obwohl es sich um ein abgeleitetes Attribut handelt, sollte es vom Server direkt bereitgestellt werden, um falsche Ergebnisse durch Zeitdifferenzen beim Ablesen von Accept und Depart zu vermeiden.

- Antwortzeit

  Delay: Bearbeitungszeit des letzten im Intervall $[t_0, t[$ beendeten Auftrags. (Typ: Zeitdauer)

Es ist nicht immer der Fall, daß reale Server alle definierten Attribute für das Leistungsmanagement bereitstellen (können). Ist beispielsweise die erforderliche Zeit zum Auslesen eines Attributs größer als die Bearbeitungszeit eines Auftrags, kann `Delay` nicht sinnvoll bestimmt werden. Bei vielen etablierten Systemdiensten wurde aber bei der Entwicklung der Software versäumt, die geforderte Managementinformation verfügbar zu machen. Dies könnte oft durch geeignete Instrumentierung der Server ohne großen Aufwand nachgeholt werden.

Gutschmidt beschreibt in seiner Arbeit zwei weitere Attribute. Der Zähler «Correct» gibt die Zahl der korrekt bearbeiteten und beendeten Aufträge im Intervall $[t_0, t[$ an. Das Attribut «Error» zählt die fehlerhaft bearbeiteten Aufträge im gleichen Intervall. Zusätzlich gilt die Beziehung $Depart = Correct + Error$. Die Modellierung der Klasse **Server** weicht hiervon etwas ab, wie die nächsten beiden Abschnitte erläutern.

Zum Leistungsmanagement gehört auch die Aufgabe, die durch Überwachung und Auswertung der oben definierten Leistungskennzahlen identifizierten Engpässe durch geeignetes Tuning des Servers zu beseitigen. Hierfür ist es aber kaum möglich, generische Funktionalität zu definieren. Meistens ist ein Tuning nur durch Verändern von Konfigurationsparametern und erneuter Initialisierung des Servers möglich.

[1] Unter Füllung versteht man die Anzahl der Aufträge, die gerade vom Server bearbeitet werden oder im Server auf Bearbeitung warten.

Fehlermanagement

Bei Servern für konkrete Systemdienste können eine Vielzahl von Fehlern mit unterschiedlichsten Ursachen auftreten. Für die generische Klasse **Server** können die Fehler aber in zwei große Gruppen eingeteilt werden . Das Attribut `Bad` zählt die Anzahl der im Intervall $[t_0, t[$ eingegangenen fehlerbehafteten Aufträge, die der Server als solche erkennt und vor der Bearbeitung verwirft. Beispiel hierfür sind Aufträge, die durch einen Übertragungsfehler verfälscht wurden. Ein zweites Attribut `Error` zählt die Aufträge, bei denen während der Bearbeitung ein interner Fehler im Server auftritt, der dazu führt, daß der Auftrag nicht korrekt abgeschlossen wird. Beispielsweise könnte in einem Server ein Deadlock oder ein *stack overflow error* auftreten.

Zusätzlich könnte gefordert werden, daß ein Server bei Auftreten eines Fehlers eine asynchrone Meldung (*notification*) möglichst mit Angabe der Ursache an die Managementanwendung sendet.

Sicherheitsmanagement

Eine Definition von Managementfunktionalität für die Zugangskontrolle zu einem Systemdienst ist für eine generische Klasse wiederum nahezu unmöglich. Zu unterschiedlich sind die möglichen Ausprägungen einer solchen Kontrolle. Diese kann z.B. durch Authentifizierung mittels Kennung und Paßwort, durch Führung von Berechtigungslisten (*access control lists*) oder sogar durch einen eigenen Sicherheitsdienst (z.B. Kerberos) erfolgen.

Für die Klasse **Server** wird daher nur ein Attribut `NotAuthorized` eingeführt, welches die Aufträge zählt, die vom Server im Intervall $[t_0, t[$ aufgrund fehlender Berechtigung verworfen wurden.

Wiederum könnte gefordert werden, daß der Server bei Ablehnung eines Auftrags aus Sicherheitsgründen eine asynchrone Meldung mit Informationen über den abgelehnten Auftrag an die Managementapplikation sendet.

Nun kann die Anzahl der im Intervall $[t_0, t[$ erfolgreich bearbeiteten Aufträge über die Formel $Depart - Bad - Error - NotAuthorized$ berechnet werden. Dieses Attribut wird nicht in die Klasse aufgenommen, da es sich aus anderen Attributen ableiten läßt. Da sich die Zähler `Bad` und `NotAuthorized` im Normalfall nur langsam erhöhen, ist die Gefahr von falschen Ergebnissen aufgrund von Zeitdifferenzen beim Ablesen relativ gering.

Statusmanagement

Statusattribute sind nötig, um feststellen zu können, ob ein Dienst verfügbar ist und wie die aktuelle Nutzung des Dienstes ist. Darüber hinaus muß für die Administration Funktionalität vorhanden sein, die es erlaubt, den Status des Servers zu beeinflussen.

Die Definition der Statusattribute basiert auf der *State Management Function* der ISO [ISO93]. Dort wird der Status einer Ressource über die folgenden drei Attribute ausgedrückt. Die Auswahl der zulässigen Werte ist für die Anwendung auf einen Server angepaßt.

Betriebsbereitschaft (*Operational State*): Dieses Attribut gibt an, ob ein Server einen Systemdienst bereitstellt. Hierzu muß der Server installiert und in Betrieb sein. Die Betriebsbereitschaft nimmt den Wert «enabled» ein, falls der Server in der Lage ist, Aufträge entgegenzunehmen, und «disabled», wenn dies nicht der Fall ist. Außerdem ist der Wert «unknown» möglich, wenn die Betriebsbereitschaft zu einem bestimmten Zeitpunkt nicht bestimmt werden kann. Auf dieses Attribut ist nur lesender Zugriff von Seiten des Managers möglich.

Nutzung (*Usage State*): Dieses Attribut gibt an, ob ein Dienst zum gegenwärtigen Zeitpunkt genutzt wird, das heißt, ob ein Server gerade einen oder mehrere Aufträge bearbeitet. Ein Server ist «idle», wenn er nicht benutzt wird, «active», wenn er einen oder mehrere Aufträge bearbeitet, aber noch freie Kapazitäten zur gleichzeitigen Bearbeitung weiterer Aufträge besitzt, und «busy», falls er völlig ausgelastet ist. Außerdem ist wieder der Wert «unknown» für die Nutzung möglich. Auch dieses Attribut kann natürlich nur gelesen werden.

Administration (*Administrative State*): Der Administrationszustand gibt an, ob ein Server seinen Dienst erbringen darf («unlocked») oder nicht («locked»). Solange ein Server Aufträge bearbeitet, aber keine neuen mehr akzeptieren darf, befindet er sich im Zustand «shutting down». Der ISO-Standard sieht vor, eine Steuerung einer Ressource durch schreibenden Zugriff auf dieses Attribut zu ermöglichen. Bei objektorientierter Modellierung sollte diese „Push-Button-Semantik“ aber vermieden werden. Zum Sperren bzw. Entsperren einer Ressource werden daher Methoden `lock()` und `unlock()` eingeführt. Da allerdings nur wenige der verbreiteten Server tatsächlich gesperrt werden können, wird für den Fall, daß diese Funktionalität auf einen Server nicht anwendbar ist, der zusätzliche Wert «not applicable» vorgesehen. Wiederum ist auch der Wert «unknown» möglich.

Server, die über ihren Status noch detaillierter Auskunft geben können, sollten durch eine Unterklasse von **Server** mit zusätzlichen Attributen modelliert werden. Der ISO-Standard definiert die beschriebenen Attribute allgemein für das Statusmanagement aller Arten von Ressourcen. Ist ein Attribut nicht sinnvoll anwendbar auf ein Objekt, wäre es denkbar, dieses wegzulassen oder seinen Wertebereich einzuschränken. Die Statusattribute und die Methoden `lock()` und `unlock()` werden deshalb der Oberklasse **compObject** von **Server** zugeordnet, da sie prinzipiell für alle laufenden Anwendungen Gültigkeit besitzen. Strikte Vererbung läßt allerdings kein Weglassen eines Attributs in einer vererbten Klasse zu. Hingegen verstößt das Beschränken des Wertebereichs eines Attributs in einer Unterklasse nicht gegen Regeln der Objektorientierung. Aber auch die Beschränkung führt zu Problemen, falls der Manager auf eine spezielle Ressource über eine Instanz einer allgemeineren Oberklasse zugreift. Diese Problematik wird in [ISO91] genauer diskutiert. Auch einige objektorientierte Sprachen, wie z.B. IDL, lassen eine Einschränkung nicht zu, wenn das Attribut vom Typ «Aufzählung» ist. Soll der Wertebereich in der Unterklasse erweitert werden, muß das Attribut in diesem Fall als «String» modelliert werden.

Bei einer Zustandsänderung des Attributs `OperationalState` sollte vom Server eine asynchrone Meldung mit Nennung des neuen Betriebszustands erzeugt werden. Die im UNIX-Umfeld verbreiteten Server schreiben zum Teil beim Starten und beim ordnungsgemäßen Stoppen einen Eintrag in ein Log. Damit ein Agent eine asynchrone Meldung für die Managementanwendung erzeugen kann, muß er das Log regelmäßig abfragen (*polling*). Wünschenswert wäre daher eine Instrumentierung des Servers zur Erzeugung „echter" asynchroner Meldungen. Hierbei ergibt sich aber das Problem der Abhängigkeit von einer bestimmten Managementarchitektur: CORBA *events*, SNMP *traps* oder OSI *notifications*. Da ein Server, der abrupt terminiert, kaum in der Lage sein wird, eine Meldung über Änderung seines Betriebszustands abzusetzen, ist es für die Implementierung der Klasse erforderlich, daß der Agent die Betriebsbereitschaft überwacht und die Meldung für den Manager erzeugt.

Auftragsmanagement

Aufgabe des Auftragsmanagements ist die Beobachtung und Steuerung von Einzelaufträgen. Das Objektmodell sieht eine Klasse **Request** für Einzelaufträge vor mit einer Assoziation «*processes*» zur Klasse **Server** und einer Assoziation «*is sent by*» zur Klasse **Client**. Implementierungen von Agenten für Server, die das Auftragsmanagement nicht unterstützen, werden diese Klasse nicht instantiieren.

Ein Auftrag muß innerhalb eines Servers eindeutig identifiziert werden können. Hierfür ist aber kein eigenes Attribut nötig, da jede Objektinstanz einen eindeutigen Objektidentifikator (OID) besitzt. Der Zeitpunkt, an dem der Auftrag beim Server eingetroffen ist, wird im Attribut `Arrival` festgehalten. Weiterhin ist der Name des Benutzers (`Owner`), der den Auftrag abgesetzt hat, und der Rechner (`Source`), von dem er abgesendet wurde, vorgesehen. Den Bearbeitungszustand enthält das Attribut `State`. Mögliche Werte hierfür sind «waiting», «processing», oder «unknown». Von der Modellierung der Größe eines Auftrags wird abgesehen, da hierfür unterschiedliche Interpretationen (z.B. Druckdatenmenge, Größe der Auftragsbeschreibung, etc.) je nach Auftragsart möglich sind. Dieses Attribut kann für konkrete Server in eine Unterklasse aufgenommen werden.

An Funktionalität ist die Methode `delete()` zum Löschen eines Auftrags notwendig. Bei einigen Servern kann die Abarbeitungsreihenfolge durch das Management beeinflußt werden. Die unterschiedlichen Bedienstrategien (sequentiell, prioritätsgesteuert, *urgent flag*, etc.) konkreter Server machen es aber unmöglich, hierfür Funktionaliät in der generischen Klasse zu definieren.

Konfigurationsmanagement

Das Konfigurationsmanagement für Server hat die Aufgabe, einen Server so einzurichten, daß er den Dienst möglichst optimal erbringt. Hierzu besitzen konkrete Server Parameter, die als Attribute des *managed object* modelliert werden können. Allerdings gibt es keine allgemeingültigen Konfigurationsattribute für die Oberklasse **Server**. Wünschenswert wäre die Definition von generischer Funktionalität zur Änderung der Konfiguration eines Servers. Im UNIX-Umfeld sind Konfigurationsdateien für Systemdienste üblich. Zum Auslesen und Editieren dieser Dateien könnte Funktionalität definiert werden. Es gibt aber vor allem im PC-Bereich Tendenzen, diese im System verstreuten Konfigurationsdateien durch eine zentrale Registrierdatenbank für alle Arten von Ressourcen abzulösen, auf der eine Anwendung für das Dienstmanagement aufsetzen kann. Sollte sich hierfür ein offener Standard durchsetzen, könnte generische Funktionalität zur Änderung der Konfiguration von Ressourcen – also auch Servern – definiert werden.

Das Objektmodell sieht die Methoden `start()` und `stop()` zum Starten und ordnungsgemäßen Stoppen eines Servers vor.

Managementinformation zur Installation eines Dienstes bzw. einer Anwendung, die ebenfalls zum Konfigurationsmanagement gezählt werden kann, enthalten die Klassen **Package** und **compObjTemplate** (siehe 3.5.2).

Zusammenfassung

Dieser Abschnitt beschreibt die Managementinformation und -funktionalität einer generischen Klasse **Server**, die von konkreten Servern für Systemdienste abstrahiert. Die Klasse soll als minimale Basis für Managementanwendungen dienen und ein integriertes Management von Systemdiensten durch *eine* Anwendung ermöglichen. Trotz der Allgemeinheit wird für das Status- und Leistungsmanagement wichtige Information und Funktionalität geboten. Es wurde bewußt darauf verzichtet, die innere Struktur eines Servers durch mehrere Klassen für einzelne Komponenten zu modellieren. Durch die Einfachkeit der Klasse sollte eine Implementierung eines Agenten für einen konkreten Server möglichst ohne zusätzliche Instrumentierung der Software, die den Dienst realisiert, möglich sein.

4.4.2 Client

Den Großteil des Managements eines Systemdienstes macht sicherlich die Überwachung, Konfiguration und Steuerung des Servers aus. Ist aber eine zentrale Instanz auf einem System erforderlich, damit der Dienst genutzt werden kann, muß diese Instanz, also der Client, ebenfalls in das Management mit einbezogen werden. Dies ist beispielsweise dann der Fall, wenn alle Aufträge an einen Server über einen einzigen Client auf dem System abgewickelt werden. Oft ist der Client dann integraler Bestandteil des Betriebssystems und ermöglicht so die Dienstnutzung durch andere Anwendungen. In gewisser Weise fungieren diese Clients also als Server für andere Anwendungen. Ein Beispiel hierfür ist der später genauer betrachtete Dateidienst NFS.

Analog zur Klasse **Server** wird auch die generische Klasse **Client** anhand der Managementfunktionsbereiche modelliert. Da Information und Funktionalität eine Untermenge von **Server** darstellen, wird die Beschreibung knapp gehalten. Die Klasse **Client** ist ebenfalls eine Unterklasse von **compObject**. Der Client besitzt mindestens ein Operational Interface, welches er an die entsprechende Dienstschnittstelle des Servers binden kann. Daneben sollte er eine im folgenden beschriebene Managementschnittstelle instantiieren.

Leistungsmanagement und Fehlermanagement

Für das Leistungsmanagement werden beim Client nur die Attribute `Request`, `Accept` und `Delay` eingeführt. `Request` enthält die Anzahl Aufträge, die der Client im Intervall $[t_0, t[$ an den Server gesendet hat. Hierbei bezeichnet t_0 wiederum den Zeitpunkt, an dem der Client gestartet wurde, bzw., an dem die Zähler zuletzt zurückgesetzt wurden. Der Zeitpunkt des Auslesens ist t. Die Anzahl der Aufträge, die korrekt bearbeitet wurden, das heißt, für die der Client eine Antwort erhalten hat, wird durch das Attribut `Accept` gezählt. `Delay` ist die Antwortzeit des letzten Auftrags im Intervall $[t_0, t[$ aus der Sicht des Clients. Hierin sind neben der Bearbeitungszeit durch den Server auch die Übertragungszeiten für die Anforderung und die Antwort enthalten. Ein Vergleich der durchschnittlichen Antwortzeiten von Client und Server gibt Ausschluß über die Übertragungsverzögerung des Netzes. Die Attribute `Depart` und `Utilization` sind nicht sinnvoll auf den Client anzuwenden. Meist erhält der Client auch keine Rückmeldung, warum ein von ihm gesendeter Auftrag nicht bearbeitet worden ist. Dies ist z.B. dann der Fall, wenn der Server den Auftrag aufgrund eines Übertragungsfehlers nicht oder verfälscht erhält, oder wegen Mangel an Ressourcen vom Server verworfen wird. Die Anzahl der nicht bearbeiteten Aufträge kann aus der Differenz von `Request` und `Accept` natürlich berechnet werden. Für diesen Wert kann in der Managementanwendung ein Schwellwert definiert werden, um bei Überschreitung Maßnahmen zur Fehlersuche zu initiieren.

Statusmanagement und Konfigurationsmanagement

Von **compObject** erbt die Klasse **Client** die Statusattribute Betriebsbereitschaft, Nutzung und Administrationsstatus.

OperationalState: Analog zur Klasse **Server** sind die Werte «enabled», «disabled» und «unknown» für die Betriebsbereitschaft möglich.

UsageState: Für einen Client haben die Werte folgende Bedeutung. Bei streng serieller Dienstnutzung drückt «Idle» aus, daß der Client derzeit keinen Auftrag gesendet hat und demnach nicht wegen Wartens auf eine Antwort blockiert («busy») ist. Für den Spezialfall, daß ein Client mehrere Aufträge gleichzeitig stellen kann (z.B. durch einen eigenen *thread* pro Auftrag), ist der Wert «active» vorgesehen. «Active» bedeutet dann, daß der Client zwar auf Antwort eines Servers wartet, aber trotzdem weitere Aufträge stellen kann. Oft wird die Nutzung des Clients nicht bestimmbar und damit «unknown» sein.

AdminState: «Unlocked» heißt, daß der Client Aufträge zur Dienstnutzung stellen *darf*. Ein temporäres Sperren eines Clients durch ein Lock-Kommando ist eher unüblich. Der Wert «locked» wird aber angenommen, wenn der Client durch den Manager gestoppt wird. Erhält der Client im Zustand «busy» bzw. «active» ein Stopsignal, so nimmt er bezüglich der Administration den Status «shutting down» ein. Neue Aufträge dürfen dann nicht mehr gestellt werden. Nach Erhalt der Antwort für den letzten Auftrag geht der Wert in «locked» über. Auch der Administrationsstatus kann unbekannt («unknown») sein.

Bezüglich des Konfigurationsmanagements gelten für den Client die gleichen Aussagen wie für den Server. Daher werden an Funktionalität ebenfalls nur die Methoden `start()` und `stop()` zum Starten und Beenden eines Clients definiert.

Auftragsmanagement

Ein Auftragsmanagement auf Client-Seite erscheint wenig sinnvoll. Sowohl vor als auch nach dem Senden der Anforderung an den Server ist ein Auftrag im Client in der Regel nicht sichtbar, d.h. auch nicht beobachtbar oder steuerbar. Eine Ausnahme hiervon ist der Fall eines asynchron arbeitenden Clients, der vom Server eine Art Auftragsbestätigung in Form eines *handle* erhält. Trotzdem wird das Auftragsmanagement in der Regel nur den Server betreffen. Die Aufträge führen meist nur beim Server zu einer Ressourcenbindung, die vom Management überwacht oder beeinflußt werden muß. Da aber jeder Auftrag einem Client zugeordnet werden kann, wird im Objektmodell eine 1:n-Assoziation von der Klasse **Client** zur Klasse **Request** vorgesehen.

Anwendung auf konkrete Systemdienste

Die in diesem Kapitel definierten generischen Klassen **Client** und **Server** für das Management von verteilten Systemdiensten werden in Kapitel 5 auf die beiden konkreten Dienste *Network File System* (NFS) und *Network Information Service* (NIS) angewendet.

4.5 Bezug zu verschiedenen Systemmanagement-MIBs

Nach Abschluß der Modellierung wird im folgenden darauf eingegangen, inwieweit die Managementinformation und Funktionalität, die in verschiedenen, vorwiegend aus dem Bereich des IETF-Managements stammenden MIBs definiert ist, durch das vorgestellte Objektmodell abgedeckt wird. Diese wurden bereits in Kapitel 3.7.2 vorgestellt. Hiermit soll neben der Rechtfertigung der definierten Objektklassen auch die Leistungsfähigkeit des Modells demonstriert werden.

4.5.1 Host Resources MIB und MNM-UNIX-MIB

Die *Host Resources MIB* [GW93] enthält in den ersten drei Gruppen allgemeine Managementinformationen zum Betriebssystem, Speicher und Geräten eines Endsystems. Die Managementobjekte dieser Gruppen wurden in der MNM-UNIX-MIB verfeinert und sind zu einem großen Teil im Objektmodell integriert (siehe 4.3).

Weiterhin enthält die *Host Resources MIB* drei Software-Gruppen. Die Gruppe «hrSWInstalled» definiert eine SNMP-Tabelle, deren Einträge lokal auf diesem Rechner installierte Software-Komponenten darstellen. Dies können Komponenten des Betriebssystems, Gerätetreiber und Anwendungen sein. Die Attribute dieser Gruppe sind vollständig in den generischen Klassen **Package** und **compObjTemplate** enthalten. Die Gruppen «hrSWRun» und «hrSWRunPerf» definieren zwei Tabellen mit Managementinformation zu Software-Komponenten, die sich gerade in Ausführung befinden. Ein Eintrag in der ersten Tabelle beschreibt die Software-Komponente durch Attribute wie Name, Pfad, Aufrufparameter, während ein zugehöriger Eintrag in der zweiten Tabelle den von der SW-Komponente belegten Speicher und die genutzte Prozessorzeit enthält. Diese Informationen lassen sich ebenfalls aus verschiedenen ODP-Basisklassen des Objektmodells auslesen. Ein Beispiel mag dies verdeutlichen.

Betrachtet wird eine SW-Komponente, die einen Druckdienst zur Verfügung stellt. Unter UNIX ist dies gewöhnlich der *Line Printer Daemon* lpd. Die Einträge der SNMP-Tabellen würden beispielsweise so aussehen:

```
hrSWInstalledEntry:
    hrSWInstalledIndex:   5
    hrSWInstalledName:    "UNIX Line Printer Daemon, Vers. 4.5.0"
    hrSWInstalledID:      912345
    hrSWInstalledType:    operatingSystem (2)
    hrSWInstalledDate:    Thu Dec 18 17:24:58 MST 1997

hrSWRunEntry:
    hrSWRunIndex:         7
    hrSWRunName:          "UNIX Line Printer Daemon, Vers. 4.5.0"
    hrSWRunID:            912345
    hrSWRunPath:          /usr/sbin/lpd
    hrSWRunParameters:    ""
    hrSWRunType:          operatingSystem (2)
    hrSWRunStatus:        running (1)

hrSWRunPerfEntry:
    hrSWRunPerfCPU:       5347896 (centi seconds)
    hrSWRunPerfMem:       12 (kBytes)
```

Im Objektmodell wird eine sich in Ausführung befindliche SW-Komponente durch eine Instanz der Klasse **compObject** repräsentiert. Die zugehörige Instanz von **compObjTemplate** liefert zusammen mit dem **Package**-Objekt, welches das Template enthält, die Information der SNMP-Tabellenzeile `hrSWInstalledEntry`. Das Computational Object «Druckdienst» wird im System durch ein oder mehrere Basic Engineering Objects realisiert. Über die Assoziationskette zwischen den Instanzen von **compObject**, **basicEngObject**, **cluster** und **capsule** wird die Instanz der Klasse **capsule** identifiziert. Diese liefert die Information über den Status, Speicher und Prozessorzeit. Die Instanz der Klasse **capsuleTemplate**, die wiederum über die Assoziation ausgehend von dem Computational Object Template erreicht wird, liefert die noch fehlenden Angaben des Eintrags `hrSWRunEntry`, nämlich den Pfad und die Aufrufparameter.

Die Managementinformation der *Host Resources MIB* wird vom Objektmodell nahezu vollständig abgedeckt bzw. sogar erheblich erweitert. Darüber hinaus erlaubt die MIB lediglich ein Monitoring der in ihr definierten Managementobjekte. Mit den MOCs des Objektmodells kann auch aktiv, das heißt steuernd auf die Ressourcen eingewirkt werden.

4.5.2 Network Services Monitoring MIB

Die *Network Services Monitoring MIB* [KF94] definiert allgemeine Attribute, die ein effektives Monitoring von Anwendungen, die Netzdienste zur Verfügung stellen, ermöglichen sollen. Die MIB besitzt eine Tabelle für die betrachteten Dienste (`applEntry`) und eine Tabelle für die offenen Kommunikationsverbindungen dieser Dienste (`assocEntry`). Werden die Verbindungen vom Netzdienst entgegengenommen, heißen sie *inbound associations*, im anderen Fall, bei dem der Netzdienst eine Verbindung initiiert, *outbound association*.

Einige Variablen in `applEntry` sind nicht spezifisch für das Management von Netzdiensten. Diese Attribute, wie z.B. `applOperStatus` bzw. `applUptime`, werden von der MOC **compObject** abgedeckt. Kommunikationsverbindungen werden über das Objektmodell durch Instanzen der Klasse **channel** beschrieben. Während die *Network Services Monitoring MIB* eine Verbindung nur von der Seite des Netzdienstes betrachtet, liefert **channel** die Sicht auf eine Verbindung als Ganzes. Ein Kanal bindet zwei Instanzen von Engineering Interfaces aneinander. Über die Rolle (Client oder Server) des zugehörigen Computational Interface auf der Seite des Netzdienstes können initiierte (*outbound* = Client) oder akzeptierte (*inbound* = Server) Verbindungen unterschieden werden. Die Anzahl der offenen Verbindungen eines Dienstes (`appInboundAssociations` und `appOutboundAssociations`) entspricht der Anzahl von Engineering Interfaces, die an Kanäle gebunden sind. Ein Großteil der übrigen in der MIB definierten Managementinformation kann direkt aus Attributen der Klasse **channel** gelesen werden.

4.5.3 System Application MIB

Die *System Application MIB* [SK97] besteht aus zwei großen Gruppen mit Objekten für installierte Software und gerade ausgeführte Anwendungen.

Die `System Application Installed Group` beschreibt zwei Tabellen für installierte SW-Pakete und deren Elemente, also Dateien. Zur Bereitstellung der Managementinformation ist es erforderlich, daß die Anwendungen als Software-Pakete vorliegen und – wie es unter UNIX üblich ist – mit einem zentralen Installationswerkzeug (*package utility*) installiert werden. Bei PC-Systemen kann die Information zum Teil aus einer globalen Datenbank des Betriebssystems (*registry*) ausgelesen werden. Da die Klassen **Package** und **InstallElement** des Objektmodells neben der *Software Standard Group Definition* gerade auf dieser MIB basieren, wird von ihnen der gleiche Informationsumfang zur Verfügung gestellt. Eine Ausnahme bildet das Attribut `sysApplInstallElmtRole`, welches für ausführbare Dateien Abhängigkeiten und Einschränkungen über boolesche Werte (z.B. «required» oder «dependent») definiert und einem Agenten ermöglicht, zur Laufzeit der Anwendung Aussagen über ihren Status zu machen. Das Attribut `RequiredObjects` der Klasse **capsuleTemplate** bietet diese Information in allgemeinerer Form.

In der zweiten Gruppe `System Application Run Group` gibt es eine Tabelle, die einen Eintrag für jede gerade ausgeführte Anwendung besitzt. Dieser Eintrag setzt sich neben einem Index aus der Startzeit und dem Status der Anwendung zusammen. In der MIB ist der Status der Anwendung identisch mit dem Status des zugehörigen Hauptprozesses und kann die Werte «running», «runnable», «waiting», «exiting» und «other» annehmen. Die Klasse **compObject** kann hier durch die drei vorhandenen Statusattribute genauer differentieren. Allerdings ist hierfür eine Instrumentierung der Anwendung nötig, da diese Statusangaben nicht aus der Prozeßliste extrahiert werden können. Eine zweite Tabelle enthält die Prozesse zu den Anwendungen aus der ersten Tabelle. Die für Prozesse typische Managementinformation wie Prozessorzeit, belegter Arbeitsspeicher und Aufrufparameter kann wiederum den Klassen **capsule** bzw. **capsuleTemplate** entnommen werden.

Beiden Tabellen dieser Gruppe ist eine weitere Tabelle zugeordnet, die Einträge beendeter oder abgebrochener Anwendungen bzw. Prozesse enthält. Befinden sich keine Prozesse der Anwendung mehr in der Prozeßliste, gilt die Anwendung als korrekt (*complete*) beendet. Im anderen Fall muß von einem Fehler ausgegangen werden und der Exit-Status ist *failed*. Die Tabellen dienen als eine Art Log (*history*) speziell für Anwendungen. Diese Funktionalität wird von dem Objektmodell nicht geboten. Es ist auch nicht sinnvoll, ein Log für Anwendungen ins Modell aufzunehmen. Da Logging und History-Funktionen elementare Managementfunktionen in vielen Bereichen darstellen, wäre es besser, hierfür generische Klassen zu entwerfen, die diese Funktionalität einer Managementanwendung generell zur Verfügung stellen. Ein Anwendungslog könnte dann eine Spezialisierung einer generischen Klasse **Log** sein.

In Abbildung 4.10 sind die generischen Klassen des Objektmodells den Tabellen der MIB gegenübergestellt. Die Information der Tabellen der `sysApplInstalledGroup` wird von den Klassen **Package**, **compObjectTemplate** und **InstallElement** abgedeckt. Analog wird die Information der `sysApplRunGroup` bis auf die History-Daten von den Klassen **compObject**, **capsuleTemplate**, **capsule** und **UNIXProcess** bereitgestellt. Insgesamt übertrifft der Informationsgehalt der Basisklassen den der MIB, wobei zusätzlich noch Operationen für ein aktives Management vorgesehen sind.

Die Managementinformation zu den laufenden Anwendungen kann im wesentlichen durch Polling der Prozeßliste gesammelt werden. Um eine feine Granularität der Information zu gewährleisten, müßte ein Agent dieses Polling in kurzen Abständen durchführen, was aber dazu führen kann, daß der Agent einen großen Teil der Rechenressourcen des Systems für sich verschlingt. Überhaupt ist es schwierig, allein aus der Analyse der laufenden Prozesse den Start und das Ende von Anwendungen zu identifizieren. Noch schwieriger gestaltet sich das Management im verteilten Fall, bei dem sich eine Anwendung aus Prozessen auf verschiedenen Systemen zusammensetzt und auf funktionierende Kommunikationsverbindungen angewiesen ist. Hier kann effizientes Management nur durch Instrumentierung der Anwendungen mit expliziten Managementschnittstellen erreicht werden, wie sie die folgende MIB fordert.

4.5.4 Application Management MIB

Die Managementinformation der *Application Management MIB* [SKPK97] ist wiederum in mehreren SNMP-Tabellen organisiert. Es wurde versucht, objektorientierte Techniken wie (multiple) Vererbung über gemeinsame Indizes in den Tabellen zu simulieren. Hieraus ist ersichtlich, daß eine mächtige objektorientierte Modellierungstechnik wie OMT zu einem anschaulicheren und die Realität besser abbildenden Informationsmodell führt.

Wie die Übersicht in Kapitel 3.7.2 zeigt, liegt ein Schwerpunkt der MIB auf der Einführung von generischen Konzepten für das Anwendungsmanagement. Im folgenden wird untersucht, inwiefern die von der MIB definierte generische Managementinformation und -funktionalität durch die ODP-Basisklassen des Objektmodells abgedeckt wird.

Die *Application Management MIB* führt eine dienstorientierte Sicht (*service level view*) ein. Es gibt daher eine Tabelle, die einer Anwendung ein oder mehrere benannte Dienste zuordnet. Dies entspricht genau der Sicht des Objektmodells, bei dem sich eine Anwendung aus ein oder mehreren Computational Objects zusammensetzt, die an ihren Schnittstellen Dienste anbieten. Weiterhin gibt es eine Tabelle, die die Prozesse beschreibt, die einen Dienst realisieren.

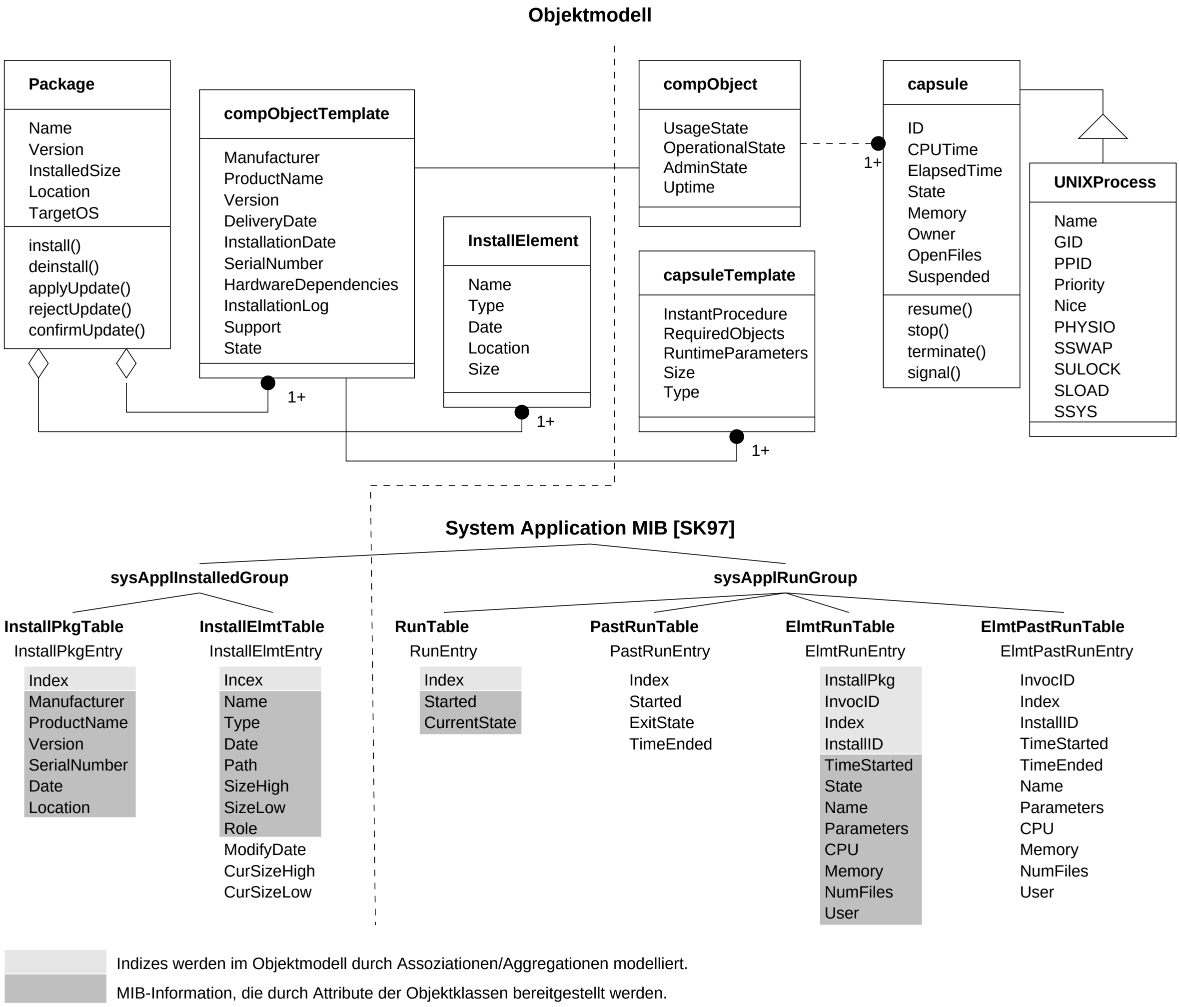


Abbildung 4.10: Gegenüberstellung von Basisklassen und System Application MIB

Im Modell entspricht dies Instanzen von **capsule**, die über die Enthaltenseinshierarchie Basic Engineering Object, Cluster, Capsule einem Dienst (Computational Object) zugeordnet werden können.

Für Dienste wie NFS-, FTP- oder WWW-Server ergeben sich wichtige Leistungsdaten aus der Statistik der Dateizugriffe. Relevante Fragestellungen sind hier „Auf welche Dateien wurde wie oft (lesend oder schreibend) zugegriffen?“ oder „Wie groß war die übertragene Datenmenge (in Bytes) im Mittel?“, etc. Hierzu definiert die MIB eine Tabelle `applOpenFileTable` mit Einträgen zu den offenen Dateien einer Anwendung. Die folgende Tabelle listet die wichtigsten Attribute eines Eintrags und deren Bedeutung auf:

`applOpenFileOpenTime`	Öffnungszeitpunkt der Datei
`applOpenFileName`	Dateiname
`applOpenFileReadRequests`	Anzahl Lesezugriffe
`applOpenFileReadFailures`	Anzahl Lesefehler
`applOpenFileBytesRead`	Anzahl gelesener Zeichen

`applOpenFileLastReadTime`	Letzter Lesezugriff
`applOpenFileWriteRequests`	Anzahl Schreibzugriffe
`[...]`	analog für Schreibzugriffe
`applOpenFileSize`	Dateigröße

Da das RM-ODP keine Konzepte für Dateien enthält, wird der Dateizugriff im Objektmodell über eine neue Klasse **fileInterface** modelliert, die eine Unterklasse von **operationInterface** ist. Für diese Klasse wird die feste Rolle «Client» angenommen. Sie modelliert eine Dateischnittstelle für die Anwendung zum Betriebssystem. Das entsprechende serverseitige Interface und die zugehörigen Engineering Interfaces sind Teil des Betriebssystems und werden nicht weiter modelliert.

Die Klasse **fileInterface** erhält die zusätzlichen Attribute `FileName` und `Size` für den Dateinamen und die Dateigröße. Sie abstrahiert von einer Schnittstelle mit den Methoden `read()`, `write()` etc. zum Zugriff auf Dateien. Jeder dieser Methoden sind entsprechende Unterklassen von **interactionInfo** zugeordnet. Um beispielsweise die Anzahl der Lesezugriffe auf eine geöffnete Datei festzustellen, ist das Attribut `Count` der zur Methode `read()` gehörenden Instanz von **interrogationInfo** auszulesen. Den letzten Zugriffszeitpunkt würde analog das Attribut `Last` liefern. Die Anzahl der Lesefehler wäre in einer entsprechenden Instanz von **terminationInfo** festgehalten. Das Attribut `Bytes` von **interactionInfo** gibt Auskunft über die Anzahl der gelesenen oder geschriebenen Zeichen. Der Zeitpunkt an dem die Datei geöffnet wurde, ist identisch mit dem Zeitpunkt an dem die Anwendung die Klasse **fileInterface** für diese Datei instantiiert hat.

Diese Information steht nur solange zur Verfügung, wie eine Anwendung eine Datei geöffnet hat. Wird die Datei geschlossen, wird der Eintrag in der SNMP-Tabelle bzw. die Instanz von **fileInterface** mit allen zugehörigen Interaction Infos gelöscht. Sollen die Daten auch in einer langfristigen Statistik über den Dateizugriff ausgewertet werden, bedarf es – analog zum bereits erwähnten Log – generischer Managementobjektklassen, die Funktionalität zur Führung von Statistiken verkörpern. Diese könnten für obigen Anwendungsfall wiederum geeignet spezialisiert werden. Diese Statistikklassen müßten neben Funktionalität zur Sammlung und Auswertung von Datenreihen, wie z.B. Mittelwertbildung, auch die Möglichkeit zur dauerhaften Speicherung der Daten in einer Datenbank bieten, damit Analysen über einen längeren Zeitraum (Tage, Wochen, Monate) durchgeführt werden können.

An dieser Stelle soll noch einmal angemerkt werden, daß genau wie bei der *Application Management MIB* natürlich eine Instrumentierung der Anwendungen erforderlich ist, um diese Art von Managementinformation auslesen zu können. Ein Agent innerhalb der Anwendung bzw. die Anwendung selbst muß zur Laufzeit die im Objektmodell spezifizierten Klassen wie **compObject**, **fileInterface** oder **interactionInfo** implementieren, die für eine Managementapplikation die sog. *Managed Object Boundary* darstellt.

Analog zur Tabelle der offenen Dateien definiert die *Application Management MIB* eine Tabelle `applOpenConnectionTable` mit Einträgen der offenen Kommunikationsverbindungen einer Anwendung. Ohne weitere Vertiefung listet Tabelle 4.1 zu den wichtigsten Variablen eines Eintrags die im Objektmodell korrespondierenden Klassen und Attribute. Alle Variablen eines Eintrags von `applOpenConnectionEntry` besitzen das Präfix `applOpenConnection` im Namen. Die Bedeutung sollte sich aus dem Namen ergeben.

MIB-Variable	Klasse	Attribut
`ID`	**channel**	`ID`
`OpenTime`	**channel**	`OpenTime`
`Transport`	**channel**	`TransportProtocol`
`NearEndAddr`	**engInterface**	`Reference`
bzw.	**basicEngObject**	`BindingEndpointID`
`FarEndAddr`	**engInterface**	`Reference`
bzw.	**basicEngObject**	`BindingEndpointID`
`Application`	**channel**	`ApplicationProtocol`
`ReadRequests`	**interrogationInfo**	`Count`
`ReadFailures`	**terminationInfo**	`Count`
`BytesRead`	**interrogationInfo**	`Bytes`
`LastReadTime`	**interrogationInfo**	`Last`
`WriteRequests`	**announceInfo**	`Count`
`[...]`	analog Lesen	

Tabelle 4.1: Attribute offener Kommunikationsverbindungen

Die *Application Management MIB* beschreibt auch ein einfaches Modell (siehe Abbildung 4.11) für das Auftragsmanagement. Ein Auftrag (*unit of work*) bzw. eine Transaktion setzt sich aus einer Anfrage (*request*) eines Clients (*invoker*) und der Antwort (*response*) eines Servers (*performer*) zusammen. Ein Strom von Transaktionen kann einer Datei, einer Netzverbindung oder der Standardeingabe/-ausgabe zugeordnet sein. Eine Anwendung kann Client oder Server für ein- oder mehrere Transaktionsströme sein. Die MIB definiert mehrere Tabellen mit Status- und Leistungsinformationen zu Transaktionsströmen. Dabei können verschiedene Typen von Transaktionen berücksichtigt werden. Die Tabellen besitzen u.a. Attribute zum Zählen von Transaktionen, zum Bestimmen des Zeitpunkts der letzten Transaktion und zur Messung der Zeitdauer zwischen Absenden der Anfrage und Empfang der Antwort auf Client-Seite bzw. zwischen Empfang einer Anfrage und Absenden der Antwort auf Server-Seite.

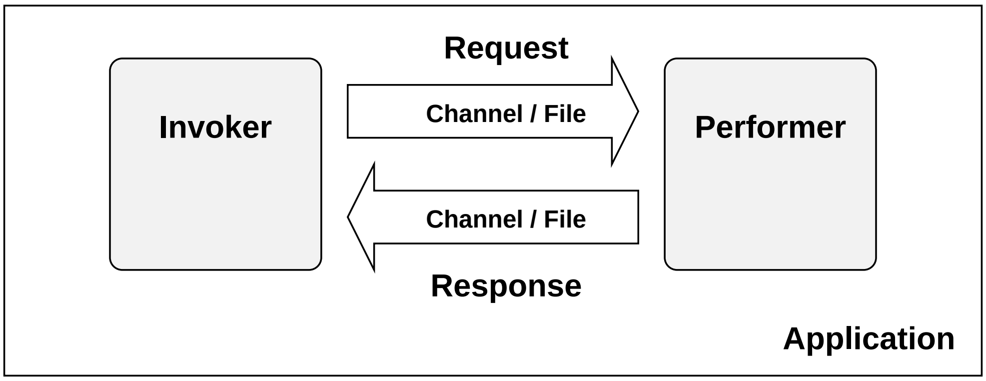


Abbildung 4.11: Modell einer Transaktion

Es handelt sich hierbei um Managementinformation auf einer höheren Ebene als bei den vorher besprochenen Tabellen. Während dort nur die Anzahl der Zugriffe auf Dateien/Verbindungen und die übertragene Datenmenge in Bytes gemessen werden kann, wird hier Leistungsinformation zu konkreten, auch für Menschen unterscheidbaren Aufträgen definiert. Ein Beispiel mag dies verdeutlichen. Bei einem FTP-Server kann die Menge an übertragenen Bytes über dessen Kommandoschnittstelle Bedeutung haben. Ein menschlicher Administrator wird aber vielmehr Interesse daran haben, wieviele Aufträge, das heißt `get`- und `put`-Kommandos über die Schnittstelle abgewickelt werden.

Im Objektmodell kann diese Information wiederum über Computational Interfaces und deren Interaction Infos modelliert werden. Beispielsweise hat ein Server (**compObject**) eine Schnittstelle (**operationInterface**), an der er Aufträge aus einem Transaktionsstrom entgegennimmt. Zu jedem Transaktionstyp gehört eine Instanz der Klasse **interrogationInfo**. Deren Attribute geben Auskunft über die Anzahl der Aufträge bzw. Transaktionen (`Count`) und über die Bearbeitungszeit des letzten Auftrags (`LastDelay`). Natürlich kann eine Anwendung mehrere Transaktionsströme bedienen. Hierzu wird die Anwendung entweder in mehrere Computational Objects aufgeteilt oder ein Computational Object instantiiert mehrere Schnittstellen. Die zugehörigen Engineering Interfaces können an einen Kanal oder an eine Dateischnittstelle des Betriebssystems gebunden sein. Es wird an dieser Stelle verzichtet, die genaue Abbildung der Tabellenattribute auf die Klassen und Attribute des Objektmodells anzugeben.

Die letzten beiden Tabellen der *Application Management MIB* behandeln den Status und die Kontrolle von Anwendungsprozessen. Die erstere ergänzt nur leicht die entsprechende Tabelle der *System Application MIB*. Die Beziehung zum Objektmodell wurde daher in Abschnitt 4.5.3 bereits beschrieben. An Steuermöglichkeiten definiert die MIB „Pushbutton-Variablen“ zum Stoppen (*suspend*), Weiterlaufen (*resume*) und Beenden (*terminate*) von Prozessen. Diese Funktionalität ist durch die entsprechenden Methoden der Klasse **capsule** abgedeckt.

Bei der *Application MIB* wird auf eine graphische Gegenüberstellung der Basisklassen mit den MIB-Tabellen verzichtet. Einerseits ist die Übereinstimmung der bereitgestellten Information nicht leicht ersichtlich (**interactionInfo**), andererseits enthält die MIB sehr viele reine Index-Tabellen deren Abbildung auf Beziehungen zwischen den MOCs nur schwierig dargestellt werden kann. Trotzdem wurde gezeigt, daß die für das Anwendungsmanagement wichtige Information und Funktionalität aus der *Application Management MIB* in den ODP-Basisklassen hinreichend modelliert ist.

4.6 Zusammenfassung

Das integrierte Management erfordert die Verschattung der Heterogenität der zu managenden Ressourcen. Daher wurde zur Suche von generischen Basisklassen für das zu spezifizierende Objektmodell ein Top-Down-Vorgehen gewählt, das die Anforderungen des Managements für Klassen von Ressourcen berücksichtigt und nicht die Eigenschaften spezieller Ressourcen. Die generischen Klassen wurden dabei aus der Analyse der Konzepte des RM-ODP abgeleitet, das eine solide und anerkannte Basis darstellt, da es einen völlig systemunabhängigen Architekturrahmen für Anwendungen in offenen, verteilten Systemen vorgibt. Aus dem Computational Viewpoint konnten vor allem MOCs für das Software-Management gewonnen werden, da dieser einen dienstorientierten Blick auf ein verteiltes System wirft. Der Engineering Viewpoint liefert dagegen eher MOCs zu Betriebssystemressourcen wie Prozessen oder Kommunikationsverbindungen. Die MOCs wurden mit möglichst allgemeiner Information und Funktionalität in Form von Attributen und Methoden angereichert, die ein effizientes Management der Ressourcen auf der Ebene der generischen Basisklassen ermöglichen soll. Ohne große Schwierigkeiten konnte im nächsten Schritt ein vorhandenes Objektmodell für das Management von UNIX-Workstations integriert werden, was die Wahl der generischen Klassen als Basis der Vererbungshierarchie rechtfertigt. Aus der Analyse der Anforderungen aus verschiedenen Managementfunktionsbereichen wurden anschließend generische Klassen für das Management von verteilten Systemdiensten in das Modell eingefügt. Die Tragfähigkeit des Objektmodells konnte durch einen Vergleich mit der Information verschiedener vorhandener SNMP-MIBs für das System- und Anwendungsmanagement nachgewiesen werden. Im folgenden Kapitel wird das Modell um MOCs zur Administration der konkreten verteilten Systemdienste NFS und NIS erweitert.

Kapitel 5

Modellierung ausgewählter Systemdienste

Dieses Kapitel beschäftigt sich mit dem Management der verteilten Systemdienste NFS und NIS. Eine kurze Einführung zu NFS und NIS enthalten die Kapitel 3.2 und 3.3. In Kapitel 4.4 wurden generische Klassen für das Management von Diensten mit Client/Server-Struktur eingeführt, die von den Basisklassen zum Anwendungsmanagement abgeleitet wurden. In den folgenden Abschnitten werden diese generischen Klassen für die zwei untersuchten Dienste NFS und NIS weiter verfeinert. Ziel dieses Kapitels ist es, das Objektmodell aus Kapitel 4 für das Management von speziellen Systemdiensten zu verfeinern. Zu NFS wird zusätzlich ein dynamisches Modell zur Spezifikation von asynchronen Meldungen entworfen.

5.1 Das Network-File-System (NFS)

Die Managementinformation und Funktionalität der Klassen **Server** und **Client** in Form von Attributen und Methoden ergaben sich aus den Anforderungen des Dienstmanagements (vgl. Abschnitt 2.1.4). In einem ersten Schritt wird anhand der Funktionsbereiche für das Dienstmanagement *top down* analysiert, welche Managementinformation NFS zur Verfügung stellen soll und welche Funktionalität für ein aktives Management dieses Dienstes erforderlich ist. Hierzu müssen die Aufgaben für die Administration von NFS betrachtet werden. Gleichzeitig wird *bottom up* überprüft, ob die geforderte Information tatsächlich auch von einem Agenten ermittelt werden kann. Grundlage für die Untersuchungen waren die Implementierungen von NFS und NIS für das Betriebssystem AIX 4.2.x von IBM. Aus den gewonnenen Informationen wird im nächsten Schritt ein erstes Objektmodell für das Management des Dienstes erstellt. Der letzte Schritt ist die Optimierung des Modells bezüglich objektorientierter Techniken und die Anbindung an die generischen Basisklassen.

5.1.1 Analyse der Managementinformation und -funktionalität

Konfigurationsmanagement

Zur Bereitstellung des Dienstes müssen die zugehörigen Prozesse gestartet werden. Diese sind für einen NFS-Server der Hauptprozeß nfsd mit einer geeigneten Anzahl Threads, der Mount-Dämon rpc.mountd und die Hintergrundprozesse für das Locking rpc.lockd und rpc.statd. Außerdem muß sichergestellt werden, daß der Portmapper-Dienst aktiv ist. Ein Client benötigt eine geeignete Anzahl biod-Prozesse und ebenfalls den rpc.lockd und rpc.statd.

Normalerweise werden die Prozesse des NFS-Server beim Booten des Systems durch ein Skript gestartet. AIX unterstützt das erstmalige Einrichten von NFS durch das Administrationskommando `mknfs`. Dieses Kommando aktiviert das Starten der NFS-Prozesse für den Client und auch für den Server, falls die Datei /etc/exports vorhanden ist. Abhängig von der über NFS abgewickelten Last kann beim Server die Anzahl der nfsd-Prozesse bzw. beim Client die Anzahl der biod-Prozesse variiert werden.

Damit ein Client auf ein Dateisystem eines Servers zugreifen kann, muß der Server dieses zuvor exportieren. Ein Server kann ganze Filesysteme oder Teile davon (Unterverzeichnisse) exportieren. Es können nur lokale Dateisysteme exportiert werden. Für jedes exportierte Dateisystem wird ein Eintrag, bestehend aus Pfadnamen und Exportoptionen, in der Datei /etc/exports erzeugt. Die optionalen Attribute beschränken den Zugriff der Clients auf das Dateisystem. Hiervon werden auch Aspekte des Sicherheitsmanagements berührt.

ReadOnly: (Boolean) Wenn dieses Attribut gesetzt ist, dürfen alle Clients nur lesend auf das Dateisystem zugreifen. Versuchte Schreiboperationen werden mit einer Fehlermeldung abgeblockt.

ReadWrite: (Hostliste) Dieses Attribut ist eine Liste der Rechner (Hostnamen), die Schreiboperationen auf dem Filesystem durchführen dürfen. Alle anderen besitzen nur Leserechte. Wird das Attribut mit einer leeren Liste angegeben, haben alle Clients Schreibrechte.

Access: (Hostliste) Falls vorhanden wird jeglicher Zugriff auf die angegebene Liste von Rechnern begrenzt. Ein Mount-Kommando eines Clients, der nicht in der Liste steht, wird abgeblockt.

Root: (Hostliste) Eine Liste der Rechner, deren Superuser *root* auch Superuser-Rechte auf diesem Dateisystem erhalten. Aus Sicherheitsgründen besitzt ein Superuser auf einem entfernten Rechner, der auf das NFS-Dateisystem zugreift, die gleichen Rechte wie ein unbekannter Benutzer. Hierzu wird die UID 0 (*root*) auf -2 (*nobody*) abgebildet. Mit dieser Option können ausgewählten Clients Superuser-Rechte auf dem Dateisystem eingeräumt werden.

Anon: (UID) Hiermit kann die User-ID festgelegt werden, mit der ein anonymer Benutzer auf Dateien zugreift. Der Defaultwert ist -2 (*nobody*). Ein Auftrag wird als anonym behandelt, wenn er keine auf dem Serversystem gültige Benutzerkennung aufweist. Wird `Anon` mit dem Wert -1 belegt, weist der NFS-Server anonyme Aufträge sofort zurück.

Secure: (Boolean) Erlaubt nur Zugriffe von Clients, die mit dem Protokoll *Secure-RPC* arbeiten.

Die Liste aller exportierten Dateisysteme eines Servers einschließlich Exportoptionen kann mit dem Kommando `exportfs` ohne Angabe von Parametern ausgegeben werden.

Ebenfalls zum Konfigurationsmanagement gehört die Auswahl der Dateisysteme, auf die ein Client per NFS zugreifen soll. Bevor ein Client ein NFS-Dateisystem nutzen kann, muß er es mittels `mount` an die Stelle eines Verzeichnisses (*mount point*) innerhalb seines lokalen Verzeichnisbaums einbinden. Gewöhnlicherweise werden die einzubindenden Dateisysteme in eine bestimmte Konfigurationsdatei (AIX: /etc/filesystems) eingetragen, damit das Einhängen beim Systemstart automatisch durchgeführt wird. Das Filesystem wird durch Angabe des Server-Rechnernamens (`Host`) und der absoluten Pfadangabe `RemotePath` bezogen auf den Server-Verzeichnisbaum bestimmt. Bei dem Pfad muß es sich um ein exportiertes oder um ein Unterverzeichnis eines exportierten Dateisystems handeln. Das Mount-Kommando kennt wiederum verschiedene Optionen:

ReadOnly: (Boolean) Falls *true* erlaubt der Client nur lesenden Zugriff auf das Dateisystem. Der Default-Wert ist *false*.

Background: (Boolean) Falls der NFS-Server nicht verfügbar ist, blockiert das Mount-Kommando normalerweise, da der zugehörige RPC im Vordergrund so oft wiederholt wird, bis der Server antwortet oder ein bestimmtes Limit für die Anzahl der Wiederholungen erreicht wird. Wird `Background` hingegen auf *true* gesetzt, blockiert das Mount-Kommando nicht, da die Operation von einem eigenen Prozeß im Hintergrund wiederholt wird.

Hard: (Boolean) Wenn *true* (Default) wird eine Dateioperation solange wiederholt, bis eine Antwort vom Server empfangen wird. Der zugehörige Prozeß wird solange blockiert, bis ein ausgefallener oder überlasteter Server wieder korrekt arbeitet. Durch Setzen von *false* (soft) wird die Operation nach einer bestimmten Anzahl von Wiederholungen mit einem Fehler abgebrochen.

Timeout: (Zehntelsekunden) Bestimmt die Zeitspanne, nach der ein Auftrag wiederholt gesendet wird, wenn der NFS-Server nicht antwortet.

Retrans: (Integer) Bestimmt die Anzahl der Wiederholungsversuche beim Zugriff auf ein Dateisystem mit der Option soft, bevor ein Timeout-Fehler gemeldet wird.

Interrupt: (Boolean) Default-Wert: *false*. Normalerweise sind RPC-Operationen nicht unterbrechbar. Wenn `Interrupt` auf *true* gesetzt wird, kann ein Benuter einen blockierten Client durch ein Kill-Signal unterbrechen. In diesem Fall wird von der RPC-Schicht ein Fehler zurückgemeldet.

Statusmanagement

Vom Management wird gefordert, daß NFS-Server und Client Informationen über ihren Status bereitstellen. In der untersuchten Implementierung gibt es allerdings keine direkten Abfragemöglichkeiten für den Status. Auch keine Meldungen über Statusänderungen werden in ein Logfile geschrieben. Durch Beobachtung der zugehörigen Prozesse kann aber eine Aussage über den Status gemacht werden.

Der NFS-Server ist definitionsgemäß verfügbar, wenn seine Prozesse nfsd, rpc.mountd, rpc.lockd, rpc.statd sowie der Portmapper existieren und lauffähig sind. AIX bietet zwar mit dem Kommando `lssrc -s nfsd` die Möglichkeit, den Status des NFS-Servers (*active bzw. disabled*) auszulesen. Dabei wird aber lediglich überprüft, ob der Hauptprozeß nfsd existiert. Der Client gilt als verfügbar, wenn die Prozesse biod, rpc.lockd und rpc.statd vorhanden sind.

Neben der Existenz der Prozesse muß sichergestellt werden, daß eine Kommunikation auf der RPC-Ebene möglich ist. Hierzu kann auf Mittel des Netzmanagements zurückgegriffen werden (vgl. [Sch97]). Außerdem implementiert jeder RPC-basierte Server eine Prozedur `null()`, die lediglich eine Antwort an den Aufrufer zurücksendet. Mit dieser Methode können alle o.g. Prozesse auf Funktions- und Kommunikationsfähigkeit überprüft werden.

Der NFS-Server unterhält während des Betriebs eine Liste der Clients, die gerade ein Dateisystem des Servers eingebunden haben. Jeder Eintrag dieser Liste besteht aus dem Rechnernamen des Clients und dem Pfad des importierten Dateisystems. Die Überwachung der Liste kann ebenfalls dem Statusmanagement zugerechnet werden, da ein Server nur beendet werden sollte, wenn kein Client Dateisysteme des Servers importiert hat. Die Liste kann mit dem Kommando `showmount` ausgelesen werden.

Auftragsmanagement

Beim NFS-Server besteht keine Möglichkeit, um einzelne Aufträge zu überwachen oder zu steuern. Der Server besitzt keine eigene Warteschlange für Aufträge. Als Warteschlange kann der Socketpuffer, aus dem die nfsd-Prozesse die Aufträge lesen, angesehen werden. Die Größe des NFS-Socketpuffers kann unter AIX mit dem Kommando `nfso nfs_socketsize` dynamisch verändert werden. Diese Tuningmaßnahme gehört aber eher zum Konfigurations- oder Leistungsmanagement. Aufgrund der kurzen Verweilzeit der Aufträge im Server ist ein Auftragsmanagemnent insgesamt nicht sinnvoll.

Fehler- und Leistungsmanagement

Das Werkzeug `nfsstat` stellt statistische Daten über Art und Anzahl der bearbeiteten Aufträge sowie über aufgetretene Fehler sowohl für den Server als auch für den Client bereit. Unterschieden werden Informationen zu Aufträgen auf der RPC-Schicht und auf der NFS-Anwendungsschicht. Abbildung 5.1 enthält die typische Ausgabe von `nfsstat` für einen Server. Die Zähler liefern neben Leistungsdaten auch wichtige Anhaltspunkte für das Fehlermanagement. Sie werden im folgenden kurz beschrieben:

- Server, RPC-Ebene

  calls: Anzahl aller empfangenen Aufträge auf RPC-Ebene seit Start des Servers bzw. letztem Rücksetzen der Zähler.

  badcalls: Anzahl der auf der RPC-Ebene zurückgewiesenen Aufträge aufgrund fehlender oder falscher Authentifizierung.

  nullrecv: Wird inkrementiert, wenn ein nfsd vom Scheduler aufgeweckt wird, aber kein Auftrag zur Bearbeitung ansteht. Eine große Zahl deutet auf zuviele Instanzen von nfsd hin.

```
# nfsstat -s

Server rpc:
calls      badcalls   nullrecv   badlen     xdrcall
155        0          0          0          0

Server nfs:
calls      badcalls
155        0
null       getattr    setattr    root       lookup     readlink   read
24 15%     42 27%     2  1%      0  0%      35 22%     0  0%      0  0%
wrcache    write      create     remove     rename     link       symlink
0  0%      0  0%      3  1%      0  0%      0  0%      0  0%      0  0%
mkdir      rmdir      readdir    fsstat
0  0%      0  0%      18 11%     31 20%
```

Abbildung 5.1: Ausgabe von nfsstat

badlen/xdrcall: Anzahl der beschädigt empfangenen Pakete. `Badlen` zählt Pakete mit falscher Länge, `xdrcall` solche mit Fehlern im RPC-Header.

- Server, NFS-Ebene

  calls: Anzahl aller Aufträge auf NFS-Ebene.

  badcalls: Gleiche Bedeutung wie auf der RPC-Ebene.

  null, getattr, ...: Die restlichen Zähler zeigen die (prozentuale) Verteilung der NFS-Aufträge auf die einzelnen vom Server implementierten Dateioperationen.

- Client, RPC-Ebene

  calls: Anzahl aller Aufträge an alle NFS-Server.

  badcalls: Anzahl aller fehlerhaften RPC-Anforderungen. Im wesentlichen werden die Aufträge gezählt, auf die innerhalb einer bestimmten Zeitspanne keine Antwort erhalten wurde. Gründe hierfür sind ausgefallene oder überlastete Server, Netzprobleme und von Servern zurückgewiesene Aufträge.

  retrans: Anzahl der wiederholt gesendeten Aufträge aufgrund fehlender Antwort.

  timeout: Anzahl der aufgetretenen Timeouts beim Warten auf Antwort. Es gilt: $timeout + badcalls >= retrans$.

  badxid: Anzahl der empfangenen Antworten, zu denen kein Auftrag im Client vorliegt. Tritt bei überlasteten oder langsamen Servern auf.

  wait: Anzahl der Aufträge, die auf Zuteilung eines UDP-Kommunikationsendpunktes warten mußten.

- Client, NFS-Ebene

  calls: siehe oben.

  badcalls: Anzahl der von Servern zurückgewiesenen Aufträge.

nclget: Für jede NFS-Operation benötigt eine Anwendung ein sog. *client handle* vom Kernel. `nclget` zählt die Anzahl der angeforderten *handles*.

nclsleep: Zählt, wie oft eine Anwendung auf die Zuteilung eines *client handle* warten mußte, weil keine mehr frei waren.

5.1.2 Erstes Objektmodell des NFS-Dienstes

Abbildung 5.2 zeigt den ersten Ansatz für das Objektmodell des NFS-Dienstes. Die im vorherigen Abschnitt definierte Managementinformation und -funktionalität wurde auf die im folgenden beschriebenen NFS-spezifischen Objektklassen verteilt.

Die Klassen **Filesystem** und **MountPoint** wurden aus dem bestehenden Objektmodell für das Systemmanagement übernommen. Ein NFS-Server wird durch die Klasse **NFSServer** modelliert. Diese enthält den Status und die für das Konfigurations- und Leistungsmanagement beschriebenen Attribute. Es gibt Methoden zum Einrichten und Löschen des Dienstes, zum Starten und Stoppen und zum Verändern der Zahl der Server-Prozesse.

Exportiert ein Server ein Filesystem, wird eine neue Instanz der Klasse **NFSFilesystem** erzeugt, welche eine Unterklasse von **Filesystem** ist. Die Klasse **NFSFilesystem** enthält als Attribute die Exportoptionen. Zusätzlich ist eine Methode `unexport()` zum Entfernen des exportierten Dateisystems vorgesehen. Die vom Server verwaltete Liste mit den Einträgen für die Dateisysteme, die ein Client zur Zeit importiert hat, wird durch die Klasse **remoteMountTabEntry** modelliert.

Ein NFS-Client wird durch die Klasse **NFSClient** beschrieben und ist bezüglich der Attribute und Methoden analog zur Klasse **NFSServer** modelliert. Importiert ein Client ein NFS-Dateisystem wird eine Instanz der Klasse **NFSMountPoint** erzeugt. Diese Klasse erbt von der Klasse **MountPoint** und enthält als Attribute die Optionen für das Mount-Kommando.

Dieses Modell kann nur als erster Ansatz verstanden werden, da eine Analyse zeigt, daß einige Aspekte des Dienstes nicht sauber modelliert wurden. Weiterhin enstand dieses Modell aus einer reinen Bottom-Up-Sicht auf den Dienst. Die vorher definierten generischen Managementklassen wurden dabei noch nicht berücksichtigt. Im nächsten Schritt wird das Modell optimiert und an das Modell der generischen Klassen angebunden.

5.1.3 Optimierung des Objektmodells

Das optimierte Modell ist in Abbildung 5.3 dargestellt. Die Schritte, die zu diesem Modell führen, werden im folgenden beschrieben.

NFS-Server

Die Klasse **NFS_Server** wird im optimierten Modell von der generischen Klasse **Server** abgeleitet. Es gelten folgende Entsprechungen für die Attribute der generischen Klasse und der alten Klasse **NFSServer**:

Request: Entspricht dem Zähler `calls` aus der Statistik von `nfsstat`

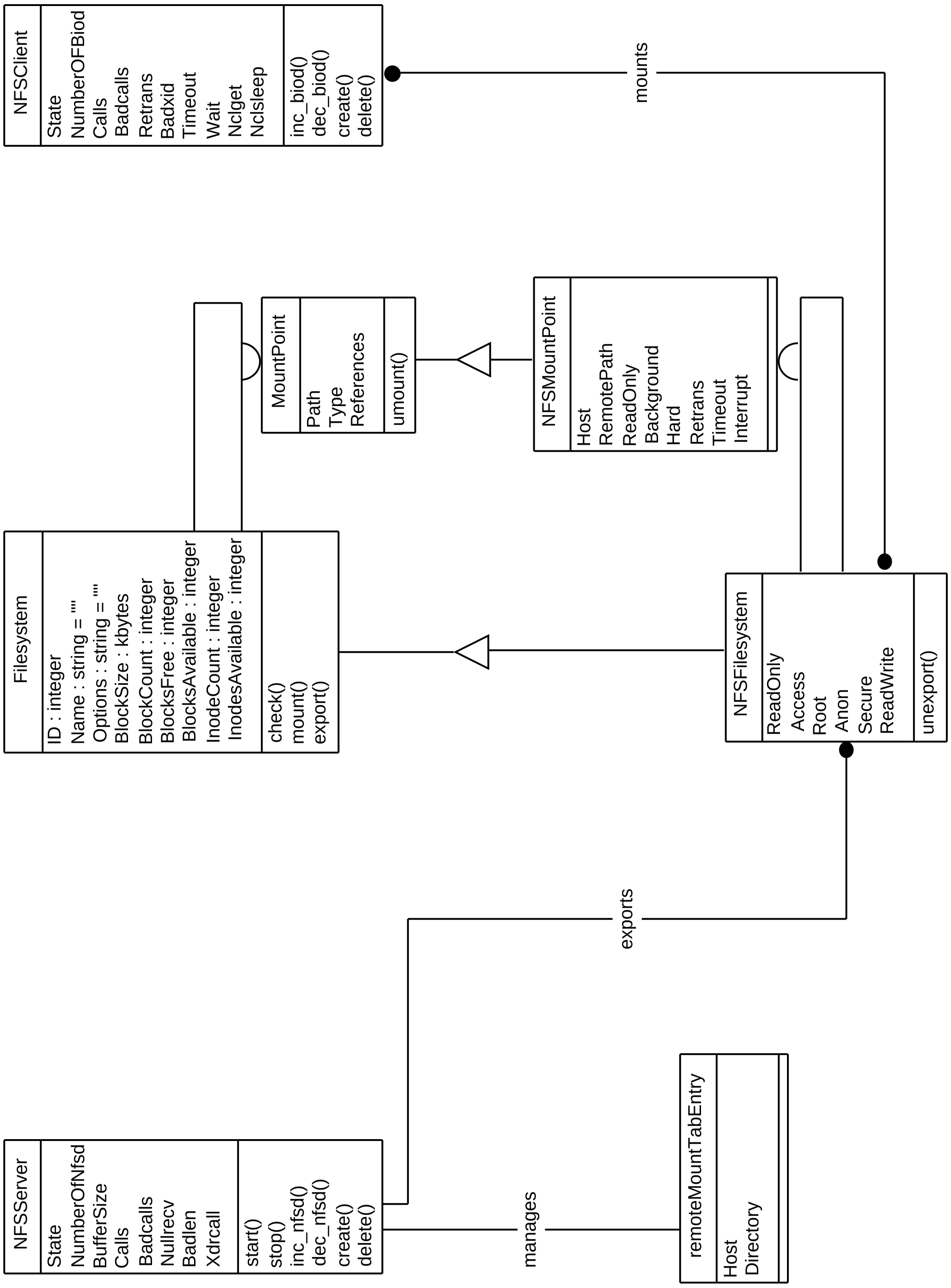


Abbildung 5.2: Erster Ansatz eines Objektmodells für NFS

Reject: Die Anzahl der zurückgewiesenen Aufträge kann beim NFS-Server nicht bereitgestellt werden. Hierzu müßte die Anzahl der verlorenen Pakete bei Überlauf des UDP-Socketpuffers ermittelt werden.

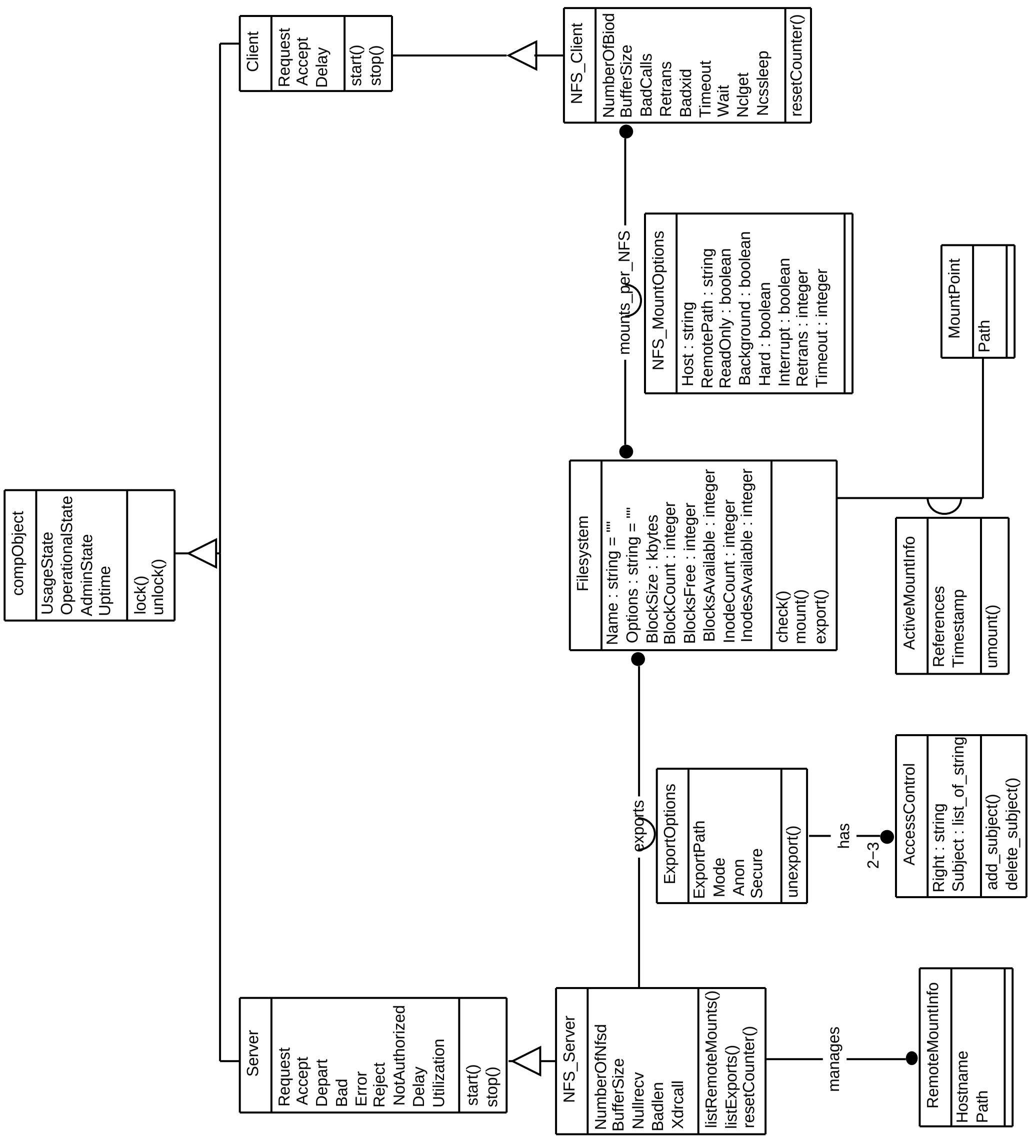


Abbildung 5.3: Optimiertes Objektmodell für NFS

Accept: Entspricht ebenfalls dem Zähler `calls` der NFS-Schicht, da für `Reject` der Wert 0 angenommen werden muß.

Utilization : Für die untersuchte Implementierung kann nicht ermittelt werden, wieviele nfsd-Serverprozesse gerade einen Auftrag bearbeiten.

Depart: Da `Utilization` nicht ermittelt werden kann, kann `Depart` aufgrund der kurzen Verweilzeit der Aufträge im Server mit `Accept` gleichgesetzt werden.

Delay: Die Bearbeitungszeit des letzten Auftrags ist ebenfalls für diese Implementierung nicht ermittelbar. Es gibt aber NFS-Implementierungen, die ein Log über die letzten n bearbeiteten Aufträge führen. Dort wird auch die Bearbeitungszeit angegeben.

Bad: Dieses Attribut entspricht der Summe aus `badlen` und `xdrcall`.

Error: Für dieses Attribut liefert `netstat` keine Information. Möglicherweise können solche Fehler durch Überwachen der Daemon-Klasse des UNIX-Systemlogs syslog identifiziert werden.

NotAuthorized: Entspricht dem Zähler `badcalls`.

Der Zustand des Servers wird über die Statusattribute, die von der Klasse **compObject** geerbt werden, ausgedrückt.

OperationalState: Die Betriebsbereitschaft kann, wie oben beschrieben, durch Kontrolle der Existenz der Serverprozesse und durch die `Null`-Prozedur der RPC-Schicht ermittelt werden.

UsageState: Da keine Aussagen über die Nutzung gemacht werden können, hat dieses Attribut stets den Wert «unknown».

AdminState: Ein NFS-Server kann nur durch das Betriebssystem gesperrt werden. Hierzu muß allen Prozessen das Signal «Stop» gesendet werden. Dies entspricht dem Zustand «locked». Der Normalzustand während des Betriebs ist «unlocked».

Durch das schreibbare Attribut `NumberOfNfsd` kann die Anzahl der nfsd-Serverprozesse dynamisch verändert werden. Dies erlaubt das AIX-Kommando `chnfs`. Ebenfalls kann die Größe des Socketpuffers durch das schreibbare Attribut `BufferSize` verändert werden. Das Attribut `Nullrecv` verbleibt ebenfalls in der Klasse **NFS_Server**. Es folgt die Beschreibung der Methoden:

create: (Methode ererbt von **Object**.) Erstmaliges Einrichten des Servers durch Anpassen der Bootskripten. AIX-Kommando: `mknfs -B`

delete: (Methode ererbt von **Object**.) Stoppen aller Serverprozesse, Entfernen der Einträge aus den Bootskripten, Löschen der Datei /etc/exports. AIX-Kommando: `rmnfs -B`

start: (Methode ererbt von **Server**.) Starten aller Serverprozesse. AIX-Kommando: `mknfs -N`

stop: (Methode ererbt von **Server**.) Stoppen aller Serverprozesse. AIX-Kommando: `rmnfs -N`

listRemoteMounts: Listet alle von Clients importierten Dateisysteme des Servers auf. AIX-Kommando: `showmount -a`

listExports: Listet alle von diesem Server exportierten Dateisysteme auf. AIX-Kommando: `lsnfsexp`

resetCounter: Setzt die Zähler für die Auftragsstatistiken zurück. AIX-Kommando: `nfsstat -sz`

Jeder Client, der ein Dateisystem des Servers importiert, wird im optimierten Modell durch eine Instanz der Klasse **RemouteMountInfo** repräsentiert. Die Liste wird über die 1:n-Aggregation zwischen **NFS_Server** und **RemoteMountInfo** modelliert. Es entfällt die Klasse **remoteMountTabEntry**.

Im ersten Modell wurde der Export eines Dateisystems durch einen Server durch Instantiierung der abgeleiteten Klasse **NFSFilesystem** beschrieben. Durch den Export wird aber in Wirklichkeit kein neues Filesystem erzeugt, sondern nur eine Berechtigung für Clients zum Import eines bestehenden Filesystems geschaffen. Aus diesem Grund wird die neue Klasse **ExportOptions** als Beziehungsklasse zwischen **NFS_Server** und **Filesystem** eingeführt und die Methode `export()` der Klasse **Filesystem** zugeordnet. Ein Export entspricht jetzt dem Schaffen einer *exports*-Assoziation zwischen einer Instanz eines NFS-Servers und eines Filesystems. Hierdurch wird die Klasse **ExportOptions** instantiiert, die als veränderbare Attribute die Exportoptionen für das Dateisystem besitzt. Streng genommen kann durch obige Assoziation nur der Export kompletter Filesysteme modelliert werden. Um das Modell nicht unnötig zu komplizieren, wird durch das zusätzliche Attribut `ExportPath` festgelegt, ob das gesamte Filesystem oder nur ein Teilbaum davon exportiert wird.

Das ebenfalls neue Attribut `Mode` der Klasse **ExportOptions** kann die Werte «read-only», «read-write» und «read-mostly» einnehmen. Letzterer Wert schränkt das Schreibrecht auf bestimmte Clients ein. Berechtigungslisten (*access control lists*) werden im neuen Modell durch die wiederverwendbare generische Klasse **AccessControl** modelliert. Das Attribut `Right` bezeichnet die vergebene Berechtigung. `Subject` ist die Liste der Subjekte, die die angegebene Berechtigung besitzen. Zum Hinzufügen und Löschen von Subjekten von der Berechtigungsliste sind die Methoden `add_subject()` und `delete_subject()` vorgesehen. Der Klasse **ExportOptions** sind genau zwei bzw. drei Berechtigungslisten zugeordnet. Eine für die Liste der zugangsberechtigten Clients (`Access`), eine für die Clients mit Superuser-Rechten (`Root`) und eine für die Clients mit Schreibberechtigung, falls das Dateisystem mit dem Modus «read-mostly» exportiert ist. Die Methode zum Aufheben des Exports (`unexport()`) wird ebenfalls der Beziehungsklasse zugeordnet, da ein „Unexport“ nur sinnvoll ist, wenn ein entsprechender Export – also eine Instanz von **ExportOptions** – existiert.

Filesystem

Im optimierten Modell wird auch die Modellierung der Beziehung zwischen einem Filesystem und dem Mount-Point geringfügig geändert. Die Klasse **MountPoint** besitzt nur noch das Attribut `Path`, welches die Stelle in einem Verzeichnisbaum festlegt, an der ein neues Dateisystem eingehängt werden kann. Durch den eigentlichen „Mount-Vorgang“ wird eine 1:1-Assoziation zwischen dem Filesystem und dem Mount-Point sowie eine Instanz der zugehörigen Beziehungsklasse **ActiveMountInfo** geschaffen. Diese Klasse enthält die Methode `umount()` zum Abhängen eines Filesystem analog zu `unexport()` in **ExportOptions**.

Client

Die Managementobjektklasse **NFS_Client** wird von der generischen Klasse **Client** abgeleitet. Die bei der Klasse **NFS_Server** gemachten Aussagen bezüglich der geerbten Statusattribute und der Methoden `create()`, `delete()`, `start()` und `stop` und `resetCounter()` können analog auf die Klasse **NFS_Client** angewendet werden. Es folgt eine kurze Beschreibung der übrigen Attribute:

Request: Entspricht dem Zähler `calls` der Auftragsstatistik, die durch das Kommando `netstat -c` ausgelesen werden kann.

Accept: Die Zahl der akzeptierten Aufträge errechnet sich aus folgenden Zählern durch die Formel: $accept = calls - badcalls - retrans$.

Delay: Die Bearbeitungszeit des letzten Auftrags kann bei dieser Implementierung nicht ermittelt werden.

BadCalls, Retrans, Badxid, Timeout, Wait, Nclget, Nclsleep: Diese Attribute entsprechen den gleichnamigen Zählern der Auftragsstatistik.

NumberOfBiod: Über dieses schreibbare Attribut kann die Anzahl der biod-Prozessse dynamisch verändert werden. AIX-Kommando: `chnfs`

BufferSize: AIX läßt eine dynamische Änderung der Größe des NFS-Socketpuffers zu. Kommando: `nfso nfs_socketsize`

Das Importieren eines NFS-Dateisystems wird im optimierten Modell durch das Schaffen einer Assoziation zwischen **Filesystem** und **NFS_Client** modelliert. Analog zur Klasse **ExportOptions** enthält die neue Beziehungsklasse **NFS_MountOptions** Attribute für die NFS-spezifischen Optionen des Imports. Das Attribut `Host` bezeichnet den Hostnamen des zugehörigen NFS-Servers. Ein Client kann optional statt des ganzen exportierten Filesystems auch nur einen Teilbaum davon importieren. Das Attribut `RemotePath` ist der entsprechende Pfad im Verzeichnisbaum des Servers.

5.1.4 Modell für asynchrone Meldungen

Neben der Definition von Managementinformation zur Überwachung und Funktionalität zur Steuerung und Manipulation einer Ressource stellen asynchrone Meldungen einen dritten wichtigen Bereich für das Management dar. Meldungen (*notifications*) werden vom Managed Object asynchron ohne Aufforderung an den Manager geschickt, wenn innerhalb des Objekts ein für das Management relevantes Ereignis, z.B. ein Fehler oder eine Statusveränderung, eintritt. Bisher wurde die Modellierung asynchroner Meldungen aus zwei Gründen vernachlässigt. Einerseits können asynchrone Meldungen im statischen OMT-Objektmodell nicht dargestellt werden. Andererseits wurde festgestellt, daß die untersuchte NFS-Implementierung keinerlei asynchrone Meldungen ausgibt bzw. in ein Logfile schreibt. Aus den Anforderungen an die Überwachung des Betriebs eines NFS-Servers lassen sich allerdings sinnvolle Meldungen für bestimmte Ereignisse definieren, wie folgende Beispiele unterschiedlicher Funktionsbereiche belegen.

Statusmanagement: Änderung des Betriebszustandes von «enabled» auf «disabled».

Fehlermanagement: Empfang eines fehlerbehafteten Auftrags.

Leistungsmanagement: Verlust von Aufträgen wegen Überlaufs des NFS-Socketpuffers.

Sicherheitsmanagement: Versuch eines unberechtigten Clients, ein Dateisystem zu importieren.

Einige Meldungen könnten durch den Agenten, der das Managed Object NFS-Server realisiert, erzeugt werden. Beispielsweise kann der Agent die Änderung der Betriebsbereitschaft auf «disabled» melden, wenn der Serverprozeß nfsd abnormal terminiert. Dies erfordert allerdings ein ständiges Überwachen der Prozeßliste durch den Agenten. Für den Manager handelt es sich trotzdem um eine asynchrone Meldung, unabhängig davon, ob der Agent oder die Ressource selbst diese erzeugt. Zur Erzeugung anderer Meldungen, z.B. Warnungen über nicht autorisierte Aufträge, müßten aber die Prozesse, die den NFS-Server realisieren, geeignet instrumentiert werden.

In [Sch97] werden einige Regeln und zugehörige Ereignismeldungen für die Überwachung der Erbringung des NFS-Dienstes definiert. Der Schwerpunkt liegt hierbei auf der Laufzeitüberwachung durch einen NFS-Monitor. Eine Regel lautet beispielsweise sinngemäß: „Wenn der Portmapper-Prozeß nicht aktiv ist, erzeuge die Ergeignismeldung «nfsmonPortmapperNotRunning»“. Die Informationsquellen für die Überwachung sind Managementobjekte aus Standard-SNMP-MIBs. Spezifische Ereignismeldungen für den NFS-Dienst werden nicht vorgesehen.

In dieser Arbeit wird mit Hilfe von Zustandsdiagrammen (*state charts*), welche OMT für die dynamische Modellierung bereitstellt, versucht, spezifische Ereignisse für den NFS-Dienst zu definieren. Bei Eintritt eines Ereignisses findet ein Zustandsübergang innerhalb des Managed Object NFS-Server auf, welcher das Senden einer asynchronen Meldung nach sich ziehen kann. Abbildung 5.4 zeigt die Notation für die Modellierung asynchroner Meldungen.

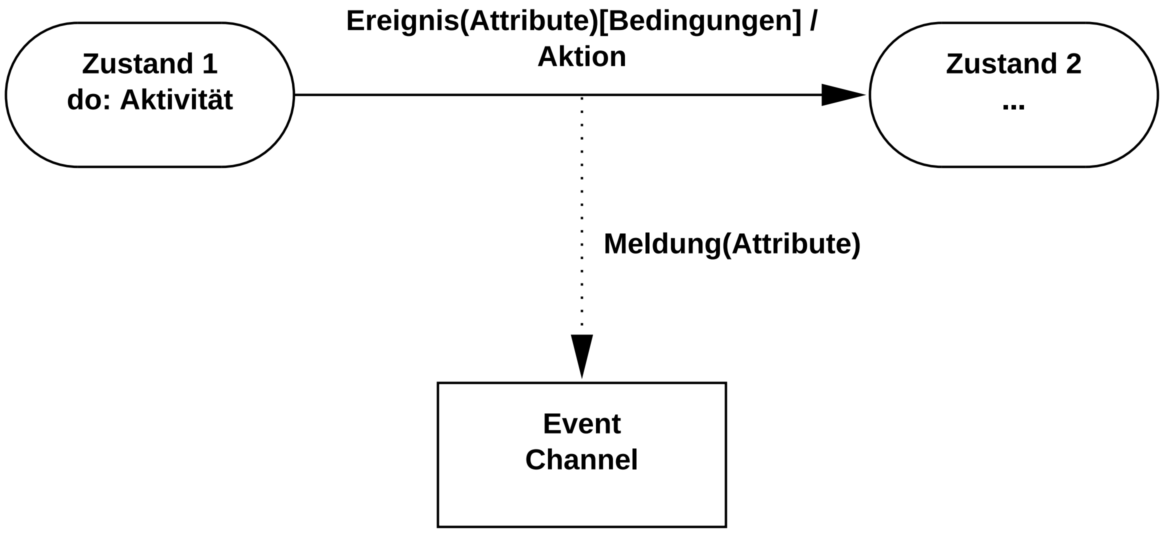


Abbildung 5.4: Notation für die Modellierung asynchroner Meldungen

Die Zustandsdiagramme sind einer Managementobjektklasse zugeordnet. Sie können ineinander geschachtelt werden. Dadurch kann die Aktivität eines Oberzustands zu einem Zustandsdiagramm mit detaillierteren Unterzuständen aufgefächert werden. Das Auftreten eines Ereignisses führt zu einem Zustandsübergang, wenn die optionale Bedingung zutrifft. Für ein Ereignis können Attribute definiert sein. Beim Zustandsübergang kann sowohl eine Aktion

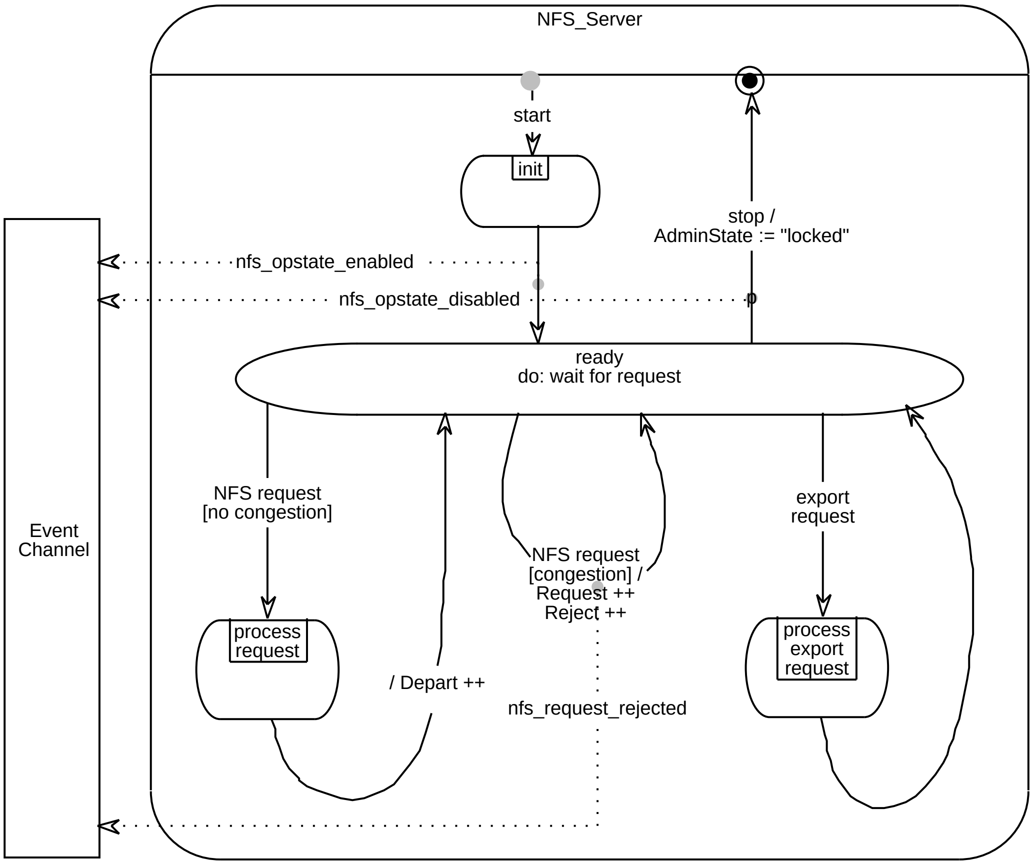


Abbildung 5.5: Zustandsdiagramm für den NFS-Server

ausgeführt werden, als auch eine Meldung mit optionalen Attributen an eine Objektklasse **EventChannel** gesendet werden. Eine Meldung ist nichts anderes als ein Ereignis für die Klasse **EventChannel**. Eine mögliche Weiterverarbeitung der Meldungen, wie z.B. Filterung, Korrelation, Logging, etc., wird nicht modelliert.

Für die Definition der asynchronen Meldungen werden im folgenden die drei Phasen Dienstbereitstellung, Betrieb und Beendigung betrachtet. Abbildung 5.5 zeigt das Zustandsdiagramm für die Klasse **NFS_Server**.

Dienstbereitstellung

Aus dem Ausgangszustand führt das Ereignis «start», welches durch die Methode `start()` ausgelöst wird, zum Zustand «init», welcher in Abbildung 5.6 detaillierter dargestellt ist. Neben dem Starten aller für den NFS-Server erforderlichen Prozesse sollte bei der Initialisierung des Dienstes auch eine Überprüfung der Konfiguration durchgeführt und Konfigurationsfehler gemeldet werden.

Zum Sicherheitsmanagement gehört die Überprüfung der User-ID für die Kennung «nobody».

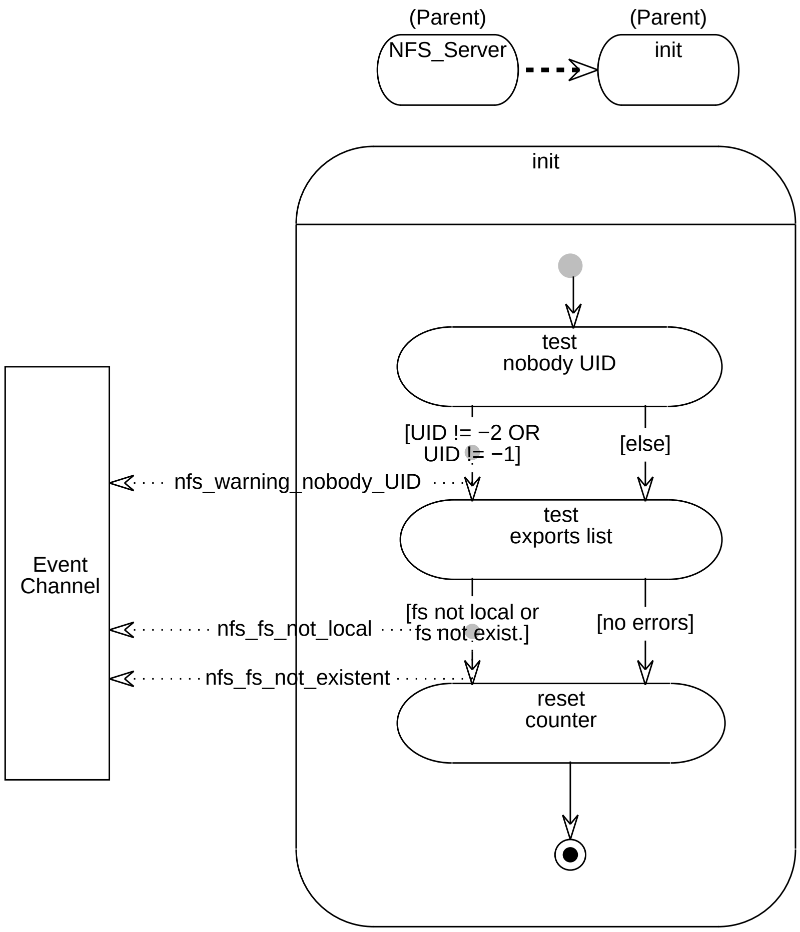


Abbildung 5.6: Zustandsdiagramm zur Initialisierung des NFS-Servers

Unter dieser Kennung werden NFS-Operationen unbekannter Benutzer bzw. nicht explizit autorisierter Superuser ausgeführt. Da Werte ungleich -1 oder -2 eine Sicherheitslücke darstellen, wird gegebenfalls die Warnmeldung «nfs_warning_nobody_UID» erzeugt.

Im nächsten Schritt wird die Liste der exportierten Dateisysteme aus der Datei /etc/exports überprüft. Die NFS-Spezifikation schreibt unter anderem die Existenz und Lokalität der exportierten Filesysteme vor, um Zyklen in der Kette Diensterbringer und Nutzer zu verhindern. Enthält die Datei Fehler, werden die Meldungen «nfs_fs_not_local» oder «nfs_fs_not_existent» erzeugt.

Vor dem Übergang in den Zustand «ready» der Betriebsphase werden zum Schluß die Zähler der Performance- und Fehlerattribute zurückgesetzt. Die Änderung der Betriebsbereitschaft auf den Wert «enabled» wird durch die Meldung «nfs_opstate_enabled» angezeigt.

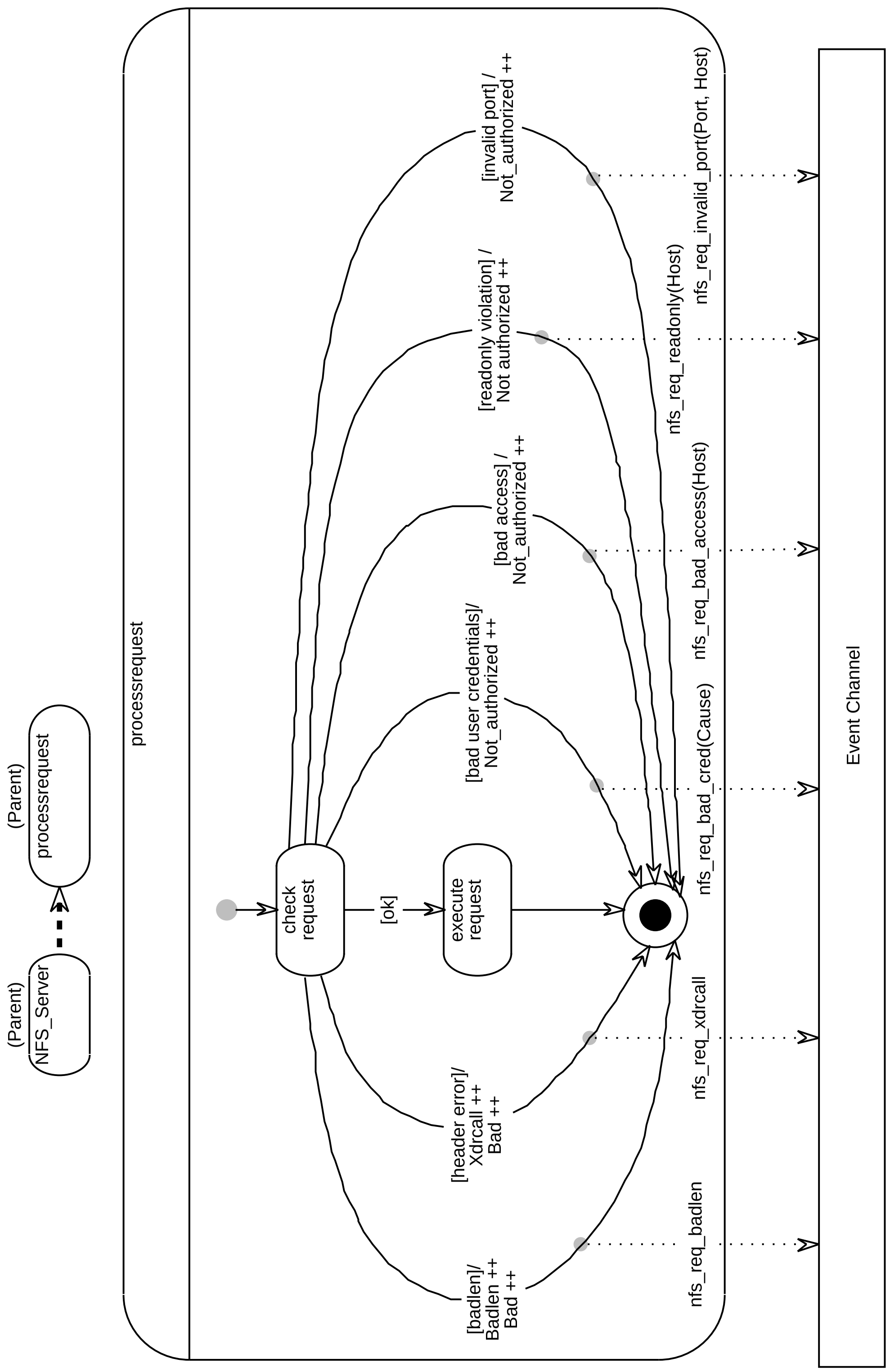


Abbildung 5.7: Zustandsdiagramm für das Bearbeiten eines Auftrags

Betriebsphase

Während des Betriebs des NFS-Dienstes sind vor allem Fehler-, Leistungs- und Sicherheitsaspekte die Grundlage für die Definition asynchroner Meldungen. Im Zustand «ready» wartet der NFS-Server auf Aufträge. In bestimmten Fällen sollte bei Eintreffen eines Auftrags eine Meldung erzeugt werden, wie folgende Aufzählung beschreibt:

Treffen mehr Aufträge für den Server ein, als die nfsd-Prozesse bearbeiten können, kommt es zur Überlastung (*congestion*) des Servers und zum Verlust von Aufträgen. Unter dieser Bedingung sollte das Eintreffen weiterer Aufträge die Meldung «nfs_request_rejected» auslösen. Im Modell ist die Parallelverarbeitung von Aufträgen durch mehrere nfsd-Serverprozesse nicht berücksichtigt. Eine genauere Abbildung der dynamischen Strukturen innerhalb eines NFS-Servers ist für diesen Zweck aber auch nicht erforderlich.

Das Eintreffen eines RPC-Auftrags bei einem nicht ausgelasteten Server führt zum Übergang von «ready» zu «process request». Dessen Unterzustände zeigt Abbildung 5.7. Vor dem Bearbeiten eines Auftrags durch den Server wird dieser auf der RPC-Schicht überprüft («check request»). Im Normalfall wird ein Auftrag nicht beanstandet und somit ausgeführt, womit wieder in den Zustand «ready» gewechselt wird. Für folgende Fehler bzw. unberechtigte Aufträge werden asynchrone Meldungen definiert:

badlen / header error: Das Eintreffen von fehlerbehafteten Aufträgen führt neben der Erhöhung entsprechender Zähler zum Auslösen von Meldungen wie «nfs_req_badlen» oder «nfs_req_xdrcall».

bad user credentials: Jeder NFS-Auftrag enthält eine Datenstruktur (*credentials*), die die User-ID und Group-IDs des Auftraggebers enthalten. Anhand dieser Datenstruktur bestimmt UNIX die Zugriffsrechte auf Dateien. Fehler in dieser Datenstruktur führen zur Zurückweisung des Auftrags und zur Meldung «nfs_req_bad_credentials(cause)». *Cause* beschreibt die Ursache wie z.B. „Benutzer unbekannt“ oder „Zu viele Gruppen“.

bad access: Aufträge, die von Client-Maschinen stammen, denen der Zugriff nicht gestattet ist. Meldung: «nfs_req_bad_access(Host)»

readonly violation: Schreibversuche auf Dateisystemen, die mit der Option «read-only» exportiert wurden. Meldung: «nfs_req_readonly(Host)»

invalid port: Aufträge eines NFS-Clients, der einen nicht privelegierten UDP-Port benutzt. Meldung: «nfs_invalid_port(Port, Host)»

Meldungen, die von einzelnen Aufträgen erzeugt werden, können zu einer Überflutung des Managers führen, wenn fehlerhafte oder nicht berechtigte Aufträge gehäuft auftreten. Vorstellbar wäre auch die Definition von Schwellwerten für obige Klassen von Aufträgen. Dann könnte eine entsprechende Meldung erzeugt werden, wenn eine Absolutanzahl oder eine bestimmte Rate von Aufträgen einer Klasse überschritten wird.

In einem eigenen Diagramm in Abbildung 5.8 wird die Behandlung von Export- bzw. Unexport-Kommandos durch den Server modelliert, da diese nicht über die RPC-Schnittstelle abgewickelt werden. AIX sieht hierfür eigene Kommandos `mknfsexp` und `rmnfsexp` vor. Eine Überwachung des NFS-Dienstes durch eine Managementanwendung sollte auch diese Kommandos abdecken. Trifft während der Laufzeit des Servers eine Anforderung zum Export eines

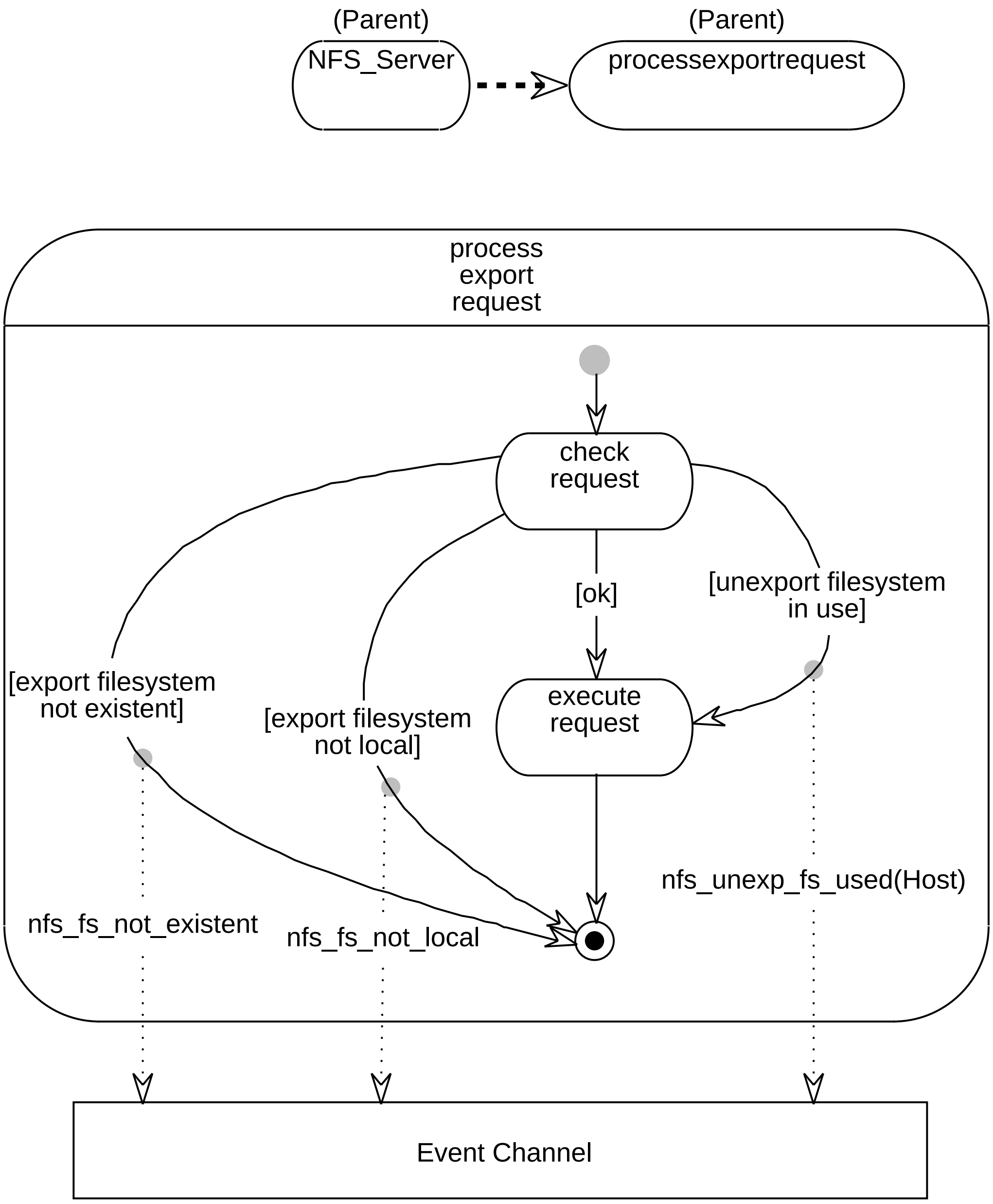


Abbildung 5.8: Zustandsdiagramm für die Prüfung von Export-Kommandos

Dateisystems auf, sollte auch hier die Existenz und Lokalität des Dateisystems überprüft werden. Im Fehlerfall wird wiederum eine Meldung an den Manager gesendet. Soll während des Betriebs der Export eines Filesystems durch ein Unexport-Kommando zurückgenommen werden, sollte der Server anhand seiner Tabelle der von Clients importierten Dateisysteme überprüfen, ob dieses Dateisystem gerade in Benutzung ist. Für diesen Fall wird die Meldung «nfs_unexp_fs_used» vorgesehen.

Beendigung

Die Beendigung des Dienstes durch Stoppen des Servers wird dem Manager durch die Meldung «nfs_opstate_disabled» angezeigt.

Diskussion

Es sei nochmal betont, daß keine NFS-Implementierung derzeit obige Meldungen tatsächlich erzeugt. Es sind Vorschläge, die vor allem die Überwachung des Betriebs erleichtern. Ein aufwendiges und Kommunikationsressourcen verschlingendes Polling kritischer Attribute durch den Manager kann hierdurch vermieden werden. Die Implementierung würde für die meisten Meldungen keinen großen Aufwand bedeuten. Außerdem wurde demonstriert, daß sich die Zustandsdiagramme von OMT gut für diese Aufgabe eignen. Durch eine anschauliche Beschriftung der Zustände und der Ereignisse, die zu einem Zustandsübergang führen, ist die Definition von asynchronen Meldungen eines Managed Object für einen Menschen leichter nachzuvollziehen, als z.B. über die in Textform vorliegenden *notification templates* des OSI-Managements.

Die Untersuchung zeigt auch, daß es schwierig ist, asynchrone Meldungen für die generischen Oberklassen einzuführen. Die meisten Meldungen sind sehr spezifisch für den Dienst NFS. Die obligatorische Meldung der Änderung der Betriebsbereitschaft könnte natürlich von der Klasse **compObject** vererbt werden. Auch könnte die Klasse **Server** generische Meldungen für das Auftreten von Fehlern oder Sicherheitsangriffen enthalten, die in spezifischen Servern durch zusätzliche Attribute verfeinert werden. Da *Meldungen* im OMT-Modell *Ereignisse* für eine Klasse **EventChannel** darstellen, kann hierfür auf die von OMT unterstützte Ereignisgeneralisierung zurückgegriffen werden, die das Vererben und Verfeinern von Attributen erlaubt. Unbefriedigend ist natürlich, daß für ein vollständiges Informationsmodell, welches auch asynchrone Meldungen enthält, unter OMT mehrere unterschiedliche Diagramme benötigt werden. Außerdem kann das Werkzeug StP aus den Zustandsdiagrammen keinen Code erzeugen. Daher können die Definitionen für asynchrone Meldungen nur als Kommentare in die IDL-Beschreibung der Klassen aufgenommen werden. Die Implementierung der asynchronen Meldungen im Agenten auf Basis des CORBA-Event-Services muß daher manuell erfolgen.

5.2 Der Network Information Service (NIS)

Im zweiten Szenario wird die Anwendbarkeit der generischen Klassen für das Management von Systemdiensten auf den *Network Information Service* (NIS) untersucht. Einen Überblick über den Dienst enthält Kapitel 3.3. Wiederum wird eine Top-Down-Analyse des Dienstes durchgeführt, um die für das Management von NIS benötigte spezifische Information und Funktionalität zu finden. Gleichzeitig wird *bottom up* überprüft, ob die geforderte Information von einem Agenten für die untersuchte Implementierung bereitgestellt werden könnte.

5.2.1 Objektmodell

Bei der Erstellung des Modells für NFS wurde so vorgegangen, daß im ersten Schritt zunächst ein Prototyp modelliert wurde. Dieser Prototyp wurde anschließend optimiert und in das Modell der generischen Klassen integriert. Da NIS von der Struktur NFS ähnlich ist, kann die bei der Modellierung von NFS gewonnene Erfahrung genutzt werden, um den Zwischenschritt zu vermeiden. Im folgenden werden zunächst die für das Management von NIS benötigten MOCs identifiziert und ihre Beziehungen untereinander beschrieben. Anhand der Managementfunktionsbereiche werden die Klassen anschließend um NIS-spezifische Information und Funktionalität ergänzt. Gleichzeitig wird erläutert, auf welche Weise diese vom Dienst bereitgestellt wird.

Das Objektmodell für das Management von NIS zeigt die Abbildung 5.9. Der obere Teil besteht aus den generischen Klassen **compObject** sowie **Server** und **Client**. Der untere Teil ist spezifisch für den Dienst NIS. Für die Modellierung der NIS-Server wird eine Klasse **NIS_Server** von **Server** abgeleitet. Diese wird weiter in die Klassen **NIS_Slave** und **NIS_Master** spezialisiert. Zwischen **NIS_Master** und **NIS_Slave** existiert eine 1:n-Assoziation. Eine NIS-Domäne wird im Modell durch die Klasse **Domain** repräsentiert. Ein NIS-Server stellt seinen Dienst für genau *eine* Domäne zur Verfügung. Zu einer Domäne gibt es aber mindestens einen, i.a. aber mehrere Server. Dies wird durch die n:1-Assoziation «provides_service_for» mit der Beziehungsklasse **ServiceInfo** zwischen **NIS_Server** und **Domain** ausgedrückt. Eine Domäne besteht aus mindestens einer NIS-Datenbank. Dies wird im Modell durch die Aggregationsbeziehung zwischen **Domain** und **Map** beschrieben. Der Master ist Eigentümer (Assoziation «owns») der Maps seiner Domäne. Hierbei ist unterstellt, daß die letzten beiden Beziehungen die gleiche Menge von Maps beinhalten. Der NIS-Client wird durch die von **Client** abgeleitete Klasse **NIS_Client** modelliert. Er ist an mindestens eine Domäne gebunden. Hierfür gibt es im Modell die 1:n-Assoziation «is_bound_to» mit der Beziehungsklasse **BindingInfo**.

Konfigurationsmanagement

Zur Bereitstellung des Dienstes ist es erforderlich, die Konfiguraton der NIS-Server und Clients auslesen und verändern zu können.

Eine der wichtigsten Konfigurationsinformationen bei NIS ist der Name der Domäne. Dieser ist trivialerweise im Attribut `Name` von **Domain** festgehalten. Das erstmalige Anlegen eines Masters impliziert ebenfalls das Anlegen einer neuen Instanz von **Domain** und zugehöriger **Map**-Instanzen. AIX sieht hierfür das Administrationskommando `mkmaster` vor. Dieses führt das Kommando `ypinit -m` aus, um die Maps im Verzeichnis /var/yp/«domainname» zu kreieren. Außerdem wird der Prozeß ypserv gestartet und abhänging von den booleschen Attributen `Passwd` und `Updated` auch die optionalen Prozesse rpc.yppasswdd und rpc.ypupdated. Zusätzlich wird das Boot-Skript rc.nfs modifiziert, um den Dienst nach dem nächsten Reboot des Systems automatisch zu starten. Sobald ein Master für eine Domäne vorhanden ist, können Slave-Server angelegt werden. Die Hostnamen aller Server einer Domäne können durch die Methode `listServers()` ausgelesen werden. Dies wird durch das AIX-Kommando `ypcat ypservers` realisiert.

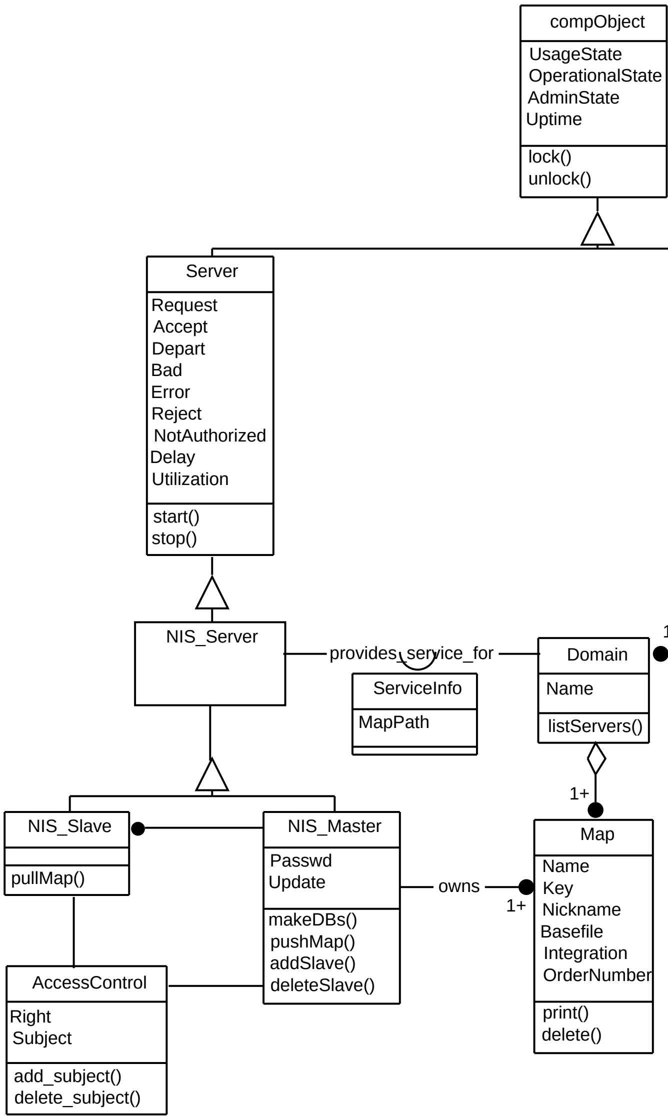


Abbildung 5.9: Objektmodell für das Management von NIS

Zu einer NIS-Map gibt es folgende nur lesbare Informationen. Das Attribut `Name` enthält den vollständigen Namen der Map. `Key` beschreibt den Schlüssel, anhand dessen die Map indiziert ist. Einige Maps besitzen einen Kurznamen (`Nickname`). `Basefile` enthält den Namen der Konfigurationsdatei, die als Quelle für die Map dient. `Integration` gibt an, ob die Map die lokale Datei eines Clients erweitert («append») oder ersetzt («replace»). Die `OrderNumber` ist der Zeitstempel, anhand dessen beim Verteilen der Map entschieden wird, ob eine Übertragung auf den Slave-Server erforderlich ist. Der Inhalt einer Map kann über die Methode `print()` ausgegeben werden. Dies entspricht dem Kommando `ypcat «MapName»`.

Die Erstellung und Verteilung neuer Maps nach Änderung der Konfigurationsdateien ist eine wichtige Aufgabe des Managements für einen NIS-Master. Hierzu werden in der Klasse **NIS_Master** die Methoden `makeDBs()` und `pushMap()` vorgesehen, die auf die UNIX-Kommandos `make -f /var/yp/Makefile` bzw. `yppush` zurückgreifen. Ein Master muß ferner seine Slaves kennen, damit er sie bei der Verteilung der Maps berücksichtigen kann. Die Methoden `addSlave()` und `deleteSlave()` entsprechen in der Realität der Verwaltung der Map ypservers und im Modell dem Anlegen und Löschen von Instanzen der Beziehung zwischen **NIS_Master** und **NIS_Slave**.

Auch ein Slave kann die Übertragung einer Map beim Master explizit anfordern, falls diese beispielsweise fehlerhaft ist. Die Methode `pullMap()` der Klasse **NIS_Slave** entspricht dem UNIX-Kommando `yppull`.

Damit auf einem Client die Dienstnutzung initialisiert werden kann, muß die Domäne festgelegt werden, an die sich der Client per Default beim Starten des Prozesses ypbind bindet. Hierfür gibt es das veränderbare Attribut `DefaultDomain` in der Klasse **NIS_Client**. Vor der erstmaligen Nutzung von NIS auf einem Client ist ein Editieren der Konfigurationsdateien, die durch NIS erweitert werden, erforderlich. Für diese Managementaufgabe ist es aber kaum möglich, Attribute bzw. Methoden zu definieren.

Die von **Server** bzw. **Client** geerbten Methoden `start()` und `stop()` veranlassen das Starten bzw. das Stoppen der oben beschriebenen für einen NIS-Server bzw. Client benötigten Prozesse.

Statusmanagement

Der Status von NIS-Server und Client wird durch die von der Klasse **compObject** vererbten Statusattribute für Betriebsbereitschaft, Nutzung und Administrationszustand ausgedrückt.

- Master-Server

  OperationalState: Ein NIS-Master ist betriebsbereit («enabled»), wenn die Prozesse ypserv und portmapper vorhanden sind. Wird durch die booleschen Attribute `Passwd` und `Update` gefordert, daß der Server auch den Paßwort- oder Update-Dienst anbieten soll, müssen auch die Dämonen yppasswd bzw. ypupdated gestartet sein. Neben der Prüfung auf Existenz wird die Kommunikationsfähigkeit der Prozesse durch einen RPC, der die `Null`-Prozedur aufruft, überprüft. Ist eine der Voraussetzungen nicht gegeben, ist die Betriebsbereitschaft «disabled».

  UsageState: Aufgrund der sehr kurzen Bearbeitungszeit von Aufträgen durch den Server kann keine Aussage über die momentane Nutzung getroffen werden. Der `UsageState` nimmt für einen Server daher den Wert «unknown» ein.

  AdminState: Ein temporäres Sperren eines NIS-Servers ist außer über UNIX-Signalmechanismen nicht möglich. Während des Betriebs ist der Administrationszustand daher «unlocked». Beim Stoppen des Servers durch den Administrator (AIX: `stopsrc -s ypserv`) geht der Zustand unmittelbar in «locked» über.

- Slave-Server

  Für den Slave-Server haben die Attribute analoge Bedeutung. Für den Betriebszustand werden nur die Prozesse ypserv und portmapper überprüft.

- Client

  OperationalState: Der Dienst ist auf einem Client verfügbar, wenn der Prozeß ypbind gestartet ist. Zusätzlich muß eine Bindung zu einem Server für die Default-Domäne existieren. Den Rechnernamen des NIS-Servers enthält das Attribut `Server` der zugehörigen Instanz von **BindingInfo**. Diese Information wird durch das UNIX-Kommando `ypwhich` bereitgestellt. Natürlich muß zumindest ein Server für die Default-Domäne betriebsbereit und erreichbar sein. Dies läßt sich wiederum durch einen „Null-RPC“ überprüfen.

  UsageState: Die eigentlichen Nutzer des NIS-Dienstes sind Anwendungsprozesse, die über Systemaufrufe Konfigurationsinformationen abfragen. Auf der Seite des Clients ist daher eine Aussage über die Nutzung unsinnig. Das Attribut nimmt den Wert «not applicable» an.

  AdminState: Es gelten die Aussagen über den Server analog für den Prozeß ypbind.

Sicherheitsmanagement

Da NIS sicherheitsrelevante Konfigurationsinformationen in seinen Datenbanken verwaltet, sollte der Zugriff auf diese Informationen nur auf den Rechnern möglich sein, die ohnehin zu dieser Domäne gehören. Falls ein Rechner eine IP-Verbindung zu irgendeinem NIS-Server aufbauen kann und der Name der NIS-Domäne bekannt ist, kann grundsätzlich ein auf diesem Rechner laufende NIS-Client alle Inhalte der Maps abfragen. Diesem Umstand kann durch Anlegen einer Liste von IP-Subnetzen oder IP-Adressen in der Datei /var/yp/securenets vorgebeugt werden. Existiert diese Datei, dürfen nur Rechner und Subnetze dieser Liste Informationen vom NIS-Server anfordern. Andere Implementierungen benutzen für diesen Mechanismus die Dateien /etc/hosts.allow und /etc/hosts.deny.

Diese Zugangskontrolliste wird im Objektmodell wieder durch die generische Klasse **AccessControl** modelliert. Implizit sollen die 1:1-Assoziationen von Master und Slave zur Klasse **AccessControl** die gleiche Instanz adressieren. Unterschiedliche Listen auf NIS-Servern der gleichen Domäne sind nicht sinnvoll, da nicht beeinflußt werden kann, an welchen Server sich ein Client bindet.

Das Attribut `NotAuthorized`, welches die Anzahl zurückgewiesener NIS-Aufträge beinhaltet, kann durch Parsen des UNIX-Systemlogs syslog implementiert werden. Der Server schreibt für jede Anfrage, die von einem unberechtigten Rechner stammt, einen Eintrag in dieses Log.

Fehler- und Leistungsmanagement

Die verbreiteten Implementierungen von NIS-Servern stellen keinerlei Informationen zum Leistungsmanagement bereit. Da einzelne Aufträge auch von außen durch einen Agenten nicht beobachtbar sind, können die Performance-Attribute, die die Klasse **NIS_Server** von **Server** erbt, nicht implementiert werden.

Für jede Anfrage, die eine vom Server nicht unterstützte Domäne oder eine nicht vorhandene Map adressiert, schreibt der Server ebenfalls eine Meldung in das Systemlog. Die Anzahl dieser falschen Anfragen können vom Attribut `Bad` bereitgestellt werden. Interne Fehler des NIS-Servers können wie beim NFS-Server durch Überwachen der Dämon-Klasse von syslog erkannt werden und in `Error` gezählt werden.

5.3 Zusammenfassung

In diesem Kapitel wurde das Objektmodell bezüglich des Managements der konkreten Client/Server-basierten Systemdienste NFS und NIS erweitert. Hierzu wurden die in Kapitel 4.4 eingeführten generischen Klassen **Server** und **Client** verfeinert und dienstspezifische zusätzliche MOCs eingeführt. Obwohl die in den generischen Klassen definierte Managementinformation sinnvoll auf die betrachteten Dienste anwendbar ist, hat die Bottom-Up-Analyse ergeben, daß die verbreiteten Implementierungen von NFS und NIS leider nur einen Teil davon zugänglich machen. Dies zeigt, daß der Managementaspekt bei der Entwicklung der Software bisher vernachlässigt wurde. Trotzdem erlaubt das Modell, daß Werkzeuge auf unterschiedlichen Ebenen arbeiten. Auf der Ebene der Klasse **compObject** ist eine Überwachung des Status aller Software-Komponenten eines verteilten Systems möglich. Ein Werkzeug für das Dienstmanagement könnte auf der Ebene der Klassen **Server** und **Client** operieren. Spezielle Tools würden sich auf unterster Ebene auf die dienstspezifischen Klassen abstützen.

Für den Dienst NFS wurde mit Hilfe von Zustandsdiagrammen des dynamischen Modells Vorschläge für asynchrone Ereignismeldungen gemacht, die das Management erheblich erleichtern würden. Prinzipiell ist ein Managementmodell, bestehend aus OMT-Objektmodell und -Zustandsdiagrammen, ähnlich mächtig wie das OSI-Informationsmodell.

Damit Anwendungen und Systemdienste die geforderte Managementinformation bereitstellen, müßten sie entsprechend instrumentiert werden. Wenn dies der Fall wäre, könnte ein Agent auch detailliertere Informationen durch Instantiierung der MOCs **compInterface** und **interactionInfo** (siehe Abb. 4.2 in Kapitel 4.2.1) anbieten.

Kapitel 6

Implementierung

In diesem Kapitel wird die prototypische Implementierung einiger Managementobjektklassen aus Kapitel 4 in einem CORBA-Agenten beschrieben. Zusätzlich soll zu dem Agenten eine Managementanwendung mit graphischer Oberfläche in Form eines Java-Applet entstehen. Das Vorgehen bei der Implementierung wurde in Abbildung 2.4 vorgestellt. Die folgenden Abschnitte beschreiben die einzelnen Schritte. Nach einem Überblick über die Modellierung mit dem CASE-Tool StP (6.1) wird die Generierung der IDL-Objektbeschreibung durch das Werkzeug beschrieben (6.2). Hierauf folgt die Vorstellung der verwendeten CORBA-Entwicklungsumgebung und die Beschreibung der Implementierung der Agentenobjekte (6.3). Abschnitt 6.4 beschäftigt sich mit der Realisierung des Management-Applet. Die Beschreibung der Testumgebung für den Prototypen in Abschnitt 6.5 sowie Erfahrungen zur Implementierung im letzten Abschnitt runden das Kapitel ab.

6.1 Erstellung des OMT-Modells mit Hilfe des CASE-Tools StP

6.1.1 Überblick über StP

Das CASE-Tool *Software through Pictures* wurde von der Firma *Interactive Development Environments* entwickelt, welche mittlerweile von *Aonix* übernommen wurde. Unter einer gemeinsamen Arbeitsoberfläche vereint StP eine Palette von Werkzeugen, die den gesamten Entwurfsprozeß bei der Entwicklung objektorientierter Software-Systeme unterstützen. Es können unter anderen die Modellierungstechniken OMT, Booch und neuerdings auch UML verwendet werden, für die *Aonix* jeweils eigene StP-Komponenten anbietet. Die einzelnen Produkte sind in eine gemeinsame, Client-Server-basierte Architektur (*StP Core*) eingebunden und sind multiuser-fähig. Alle Werkzeuge arbeiten auf einer gemeinsamen Datenbasis (*Repository*), die von einem DBMS verwaltet wird. Die folgende kurze Beschreibung bezieht sich auf die in dieser Arbeit eingesetzte Version von StP (StP-Core: Release 2.2, StP/OMT: Release 3). Für Details sei auf die Handbücher zu StP ([Int96a, Int96b, Int96c]) verwiesen. Abbildung 6.1 zeigt den Aufbau von StP:

StP Desktop: Von der gemeinsamen Arbeitsoberfläche, die über das Kommando `stp &` gestartet wird, können alle Editoren und Werkzeuge der StP-Produkte aufgerufen werden.

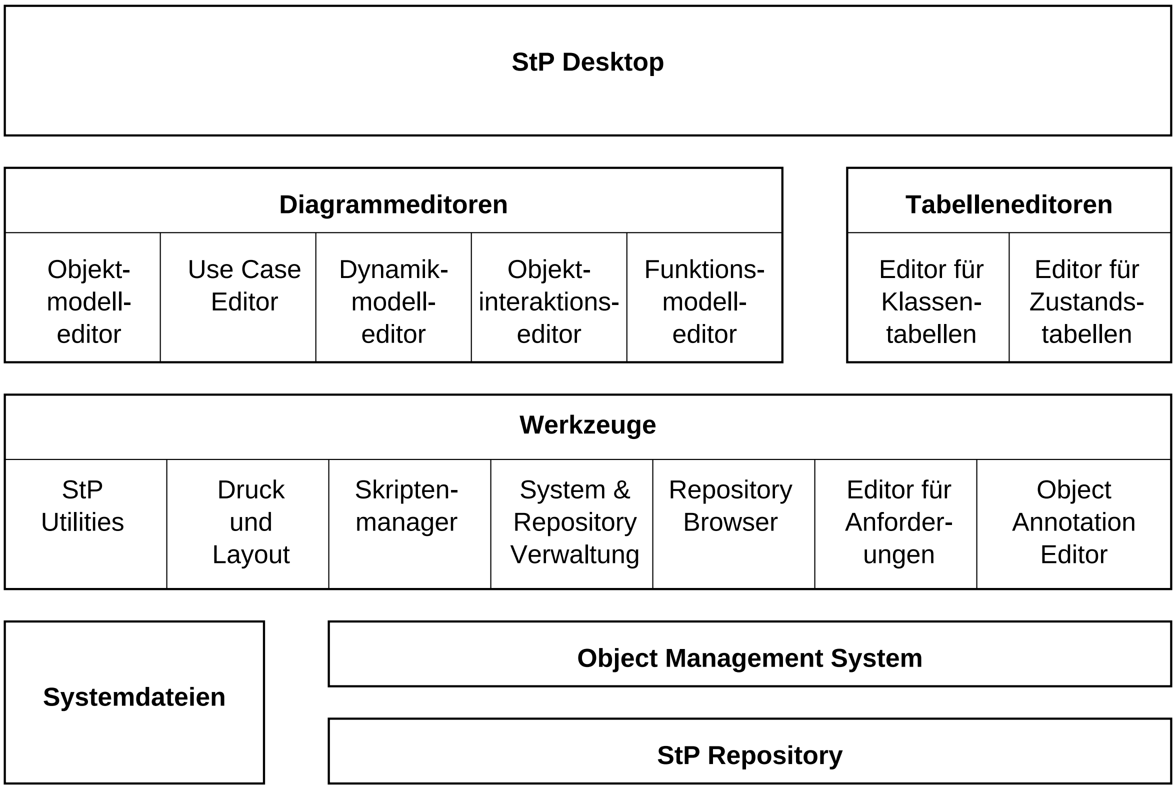


Abbildung 6.1: Die Architektur von *Software through Pictures*

Dies schließt die Projekt- und Benutzerverwaltung sowie die Administration der Datenbasis mit ein.

Diagrammeditoren: Je nach StP-Produkt und verwendeter Modellierungstechnik stehen auf dem Desktop diverse Editoren zum Erstellen von Diagrammen zur Verfügung. Für OMT sind dies u.a. die Editoren für das Objektmodell, das Dynamische Modell und das Funktionsmodell.

Tabelleneditoren: Informationen zu den Objekten der Diagramme, die nicht graphisch dargestellt werden, werden mit den Tabelleneditoren erfaßt. Die Klassentabelle enthält z.B. die Typdefinitionen für Attribute und Methodenparameter zu den Klassen des Objektmodells.

Werkzeuge: In der Architektur von StP eine Ebene tiefer angesiedelt sind die Werkzeuge, die dem Benutzer und den Modelleditoren Basisdienste bereitstellen. StP enthält zentrale Funktionen für den Ausdruck von Diagrammen, Tabellen und Reports sowie die Umwandlung in Fremdformate (Postscript, FrameMaker, Interleaf).

Der Skriptenmanager ermöglicht Anfragen auf der Datenbasis der Modelle, um automatisierte Reports zu erstellen. Auch die Code-Erzeugung basiert auf editierbaren QRL-Skripten (*Query and Reporting Language*).

Weiterhin gibt es Werkzeuge zur Verwaltung und Anzeige des Repository, einen Editor zur Formulierung von Anforderungen des Projekts und einen Editor (*Object Annotation*

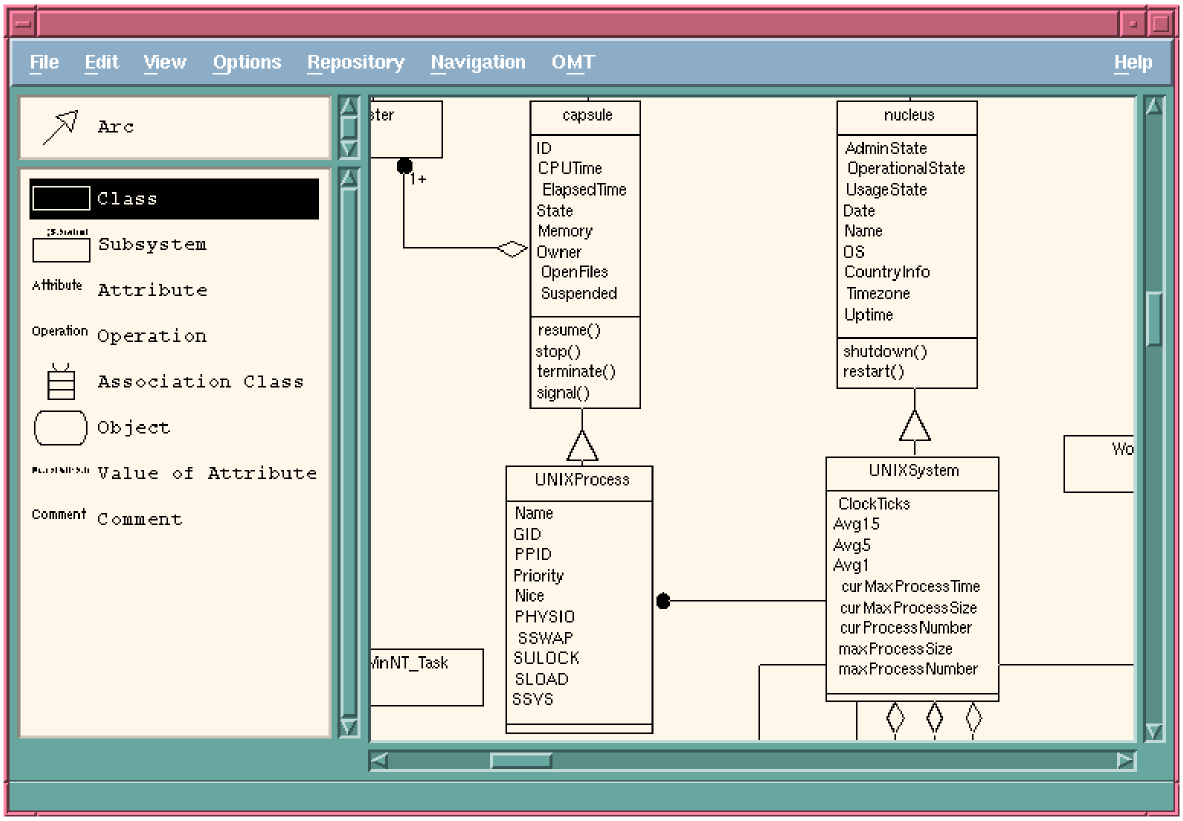


Abbildung 6.2: Der StP-Objektmodelleditor

Editor), um beliebigen Objekten zusätzliche Attribute (Werte, Textbeschreibungen, Code-Fragmente, etc.) zuordnen zu könnnen.

Basis: Sie besteht aus den StP-Systemdateien im ASCII-Format, dem *Object Management System* und der Datenbank (*Repository*). Zu jedem Diagramm und jeder Tabelle legt StP eine ASCII-Datei an, in der die Referenzen aller Objekte (auch mehrfach) gespeichert werden.

Das *Object Management System* stellt ein allgemeines, statisches Datenbankschema (*Persistent Data Model*) zur Speicherung aller Informationen aus den Modellen bereit. Dadurch ermöglicht es allen Editoren den Zugriff auf die gemeinsame Datenbasis, die in der Datenbank gespeichert wird. Zur Erreichung der Konsistenz wird ein Objekt, unabhängig davon, wie oft es in diversen Diagrammen auftaucht, nur einmal im Repository abgelegt. Ein beschädigtes Repository kann aus den redundanten Systemdateien rekonstruiert werden.

6.1.2 Der Objektmodelleditor

Der zentrale Editor in StP/OMT ist der Objektmodelleditor. Da viele Diagramme eines OMT-Modells auf den Klassen des Objektmodells basieren, sollte dieses als erstes erstellt werden. Abbildung 6.2 zeigt den Objektmodelleditor.

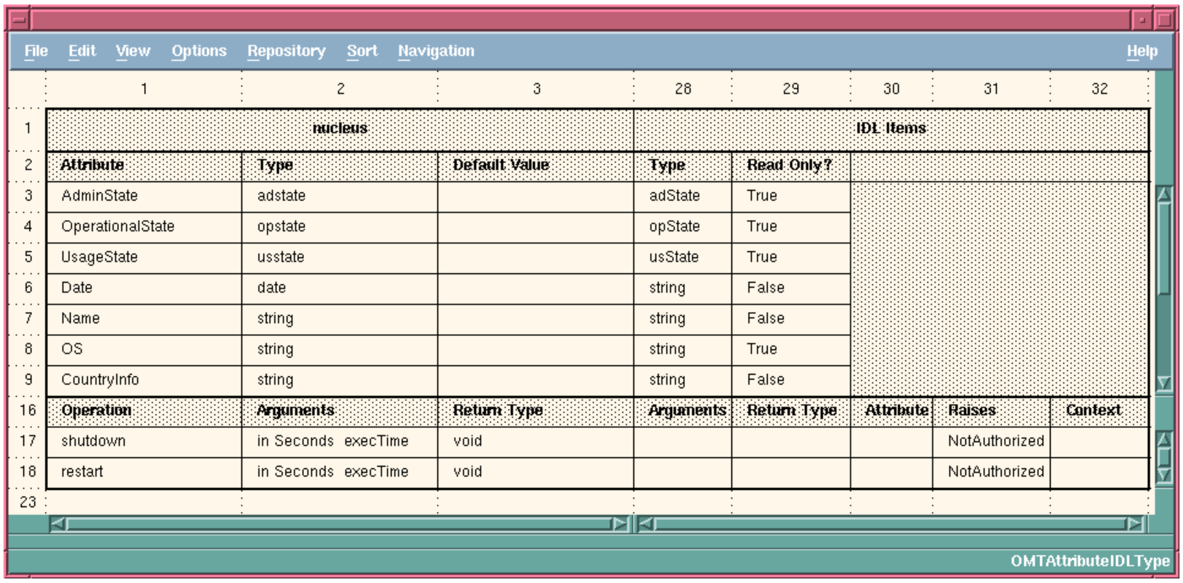
Der Aufbau der Editoren ist einheitlich und soll am Beispiel des Objektmodelleditors kurz beschrieben werden. Das Menü *File* und *Edit* stellt gewohnte Funktionen bereit. *View* enthält u.a. die Funktionen *Zoom* und *Filter*. Letztere erlaubt die Anwendung und Definition von Filtern, um unerwünschte Elemente eines Diagramms auszublenden. Der Punkt *Repository* bietet Funktionen zur syntaktischen und semantischen Überprüfung des bearbeiteten Modells an. Bezogen auf ein Objektmodell kontrolliert der Syntax-Check, ob die OMT-Notation für das Diagramm eingehalten wurde. Der Semantik-Check validiert, ob alle Objekte des Diagramms korrekt im Repository definiert sind. Fehler werden in einem Message-Log angezeigt. Vom Menüpunkt *Navigation* kann die Bearbeitung eines Objekts in einem anderen Editor fortgesetzt werden. Das letzte Menü (hier: *OMT*) enthält spezifische Modellierungsfunktionen des jeweiligen Editors.

Im linken Teil des Arbeitsbereichs wird die Symbolliste eingeblendet. Aus dieser können die für den Diagrammtyp möglichen Symbole in den Zeichenbereich auf der rechten Seite gezogen werden.

Zur Erstellung eines Objektmodells hat sich folgende Vorgehensweise bewährt. Als erstes werden Symbole für die benötigten Klassen im Diagramm angeordnet und mit den Klassennamen beschriftet. Der zweite Schritt ist die Definition der Attribute und Methoden. Damit aus den Klassen Code erzeugt werden kann, genügt es nicht, die Attribute und Methoden mit dem Objektmodelleditor in die Klassen einzutragen. Für jede Klasse muß über den Klassentabelleneditor eine sog. Klassentabelle angelegt werden, in der die Typen, Parameter und Rückgabewerte zu den Attributen und Methoden festgelegt werden. StP erlaubt hier zwei unterschiedliche Wege. Die Attribute und Methoden einer Klasse können mit dem Objektmodelleditor modelliert werden. Beim erstmaligen Anlegen der zugehörigen Klassentabelle übernimmt StP aber lediglich die Namen der Attribute und Methoden. Alternativ dazu kann für eine leere Klasse zunächst die Klassentabelle erstellt werden. Mit dem Klassentabelleneditor werden dann Attribute und Methoden einschließlich aller erforderlichen Informationen definiert. Auf den Klassentabelleneditor wird im Abschnitt 6.2 näher eingegangen. Im Objektmodelleditor werden schließlich die Attribute und Methoden aus der Klassentabelle in die Klasse übernommen. Da StP die Informationen im Diagramm und in der Klassentabelle nicht konsistent hält, empfiehlt es sich, Änderungen immer zunächst in der Klassentabelle vorzunehmen und darauf die Klasse im Diagramm neu erstellen zu lassen. Basis für die Code-Erzeugung ist in jedem Fall die Klassentabelle.

Der letzte Schritt ist die Modellierung der Beziehungen zwischen den Klassen. Bei Assoziationen und Aggregationen ist auf die Angabe der Multiplizität zu achten. Diese und andere Eigenschaften einer Assoziation können in einem Fenster festgelegt werden, welches bei Doppelklick auf die Assoziationslinie erscheint.

StP bietet relativ wenig Unterstützung bei der übersichtlichen Anordnung der Symbole. Diese sollte manuell erst dann vorgenommen werden, wenn die Spezifikation der Klassen abgeschlossen ist. Durch Hinzufügen oder Entfernen von Elementen verändert sich nämlich die Größe des Klassensymbols, was zu erheblichen Verschiebungen im Diagramm führen kann. Auch die Möglichkeiten zum Ausdruck großer Diagramme sind in der verwendeten Version etwas unbefriedigend. Ausschnitte eines Diagramms können durch Markierung aller sich im Ausschnitt befindenden Objekte und unter Benutzung des Filters «*Hide unselected symbols*» gedruckt werden. Dabei skaliert StP aber den Ausschnitt nicht neu, was bedeutet, daß wenige, weit auseinanderliegende Objekte im Ausdruck unleserlich klein erscheinen. Hilfreich ist es, se-



	1	2	3	28	29	30	31	32
1	nucleus			IDL Items				
2	Attribute	Type	Default Value	Type	Read Only?			
3	AdminState	adstate		adState	True			
4	OperationalState	opstate		opState	True			
5	UsageState	usstate		usState	True			
6	Date	date		string	False			
7	Name	string		string	False			
8	OS	string		string	True			
9	CountryInfo	string		string	False			
16	Operation	Arguments	Return Type	Arguments	Return Type	Attribute	Raises	Context
17	shutdown	in Seconds execTime	void				NotAuthorized	
18	restart	in Seconds execTime	void				NotAuthorized	
23								

Abbildung 6.3: Der StP-Klassentabelleneditor

mantisch zusammengehörende Teile des Objektmodells in mehreren kleinen, übersichtlichen Diagrammen zu modellieren. Diese Diagramme können dann manuell zu einem großen Diagramm für das gesamte Modell zusammengefaßt werden.

Im nächsten Abschnitt wird erklärt, welche Voraussetzungen zur erfolgreichen Erzeugung von IDL-Beschreibungen für die Klassen des Objektmodells berücksichtigt werden müssen. Anschließend werden die Optionen bei der Code-Generierung beschrieben und diese für eine Klasse am Beispiel erläutert.

6.2 Generierung von IDL-Objektbeschreibungen

Nach Erstellung des Objektmodells können von StP für einzelne Klassen oder für alle Klassen eines Diagramms IDL-Objektbeschreibungen erzeugt werden. Diese Funktion findet sich auf dem StP-Desktop im Menü *Commands* der Kategorie *Access Repository – Classes* und wird in [Int96d] ausführlich beschrieben. Zur Code-Erzeugung ist es erforderlich, daß für jede Klasse eine Klassentabelle angelegt wurde, welche die formalen Definitionen der Attribute und Methoden enthält. Neben allgemeinen Informationen werden in der Klassentabelle auch spezifische Parameter der Programmiersprache(n), für die Code erzeugt werden soll, festgelegt. Abbildung 6.3 zeigt den Klassentabelleneditor am Beispiel der Klasse **nucleus**.

Attribute

Allgemeine Eigenschaften von Attributen (*Analysis Items*) werden in den ersten drei Spalten spezifiziert. Die erste Spalte (`Attribute`) enthält den Namen des Attributs. In der zweiten Spalte (`Type`) wird der Datentyp eingetragen. StP überprüft nicht, ob es sich um einen gültigen

Typ handelt. Optional kann dem Attribut in der dritten Spalte (`Default Value`) ein Default-Wert zugeordnet werden.

Zu Attributen gibt es zwei IDL-spezifische Spalten (*IDL Items*). In der Spalte `Type` kann ein spezieller IDL-Datentyp für das Attribut spezifiziert werden. Dadurch wird der allgemeine Datentyp aus Spalte zwei überschrieben. Ein Attribut könnte z.B. vom Typ Integer sein. Da dieser Typ in IDL nicht existiert, wird er daher durch den IDL-Typ long ersetzt.

In der Spalte `Read Only?` wird festgelegt, ob das Attribut von einem anderen CORBA-Objekt verändert werden darf oder nicht. Davon hängt es ab, ob das Schlüsselwort `readonly` der Attributsdefinition im IDL-Code hinzugefügt wird.

Methoden

Für Methoden enthalten die *Analysis Items* die Felder `Operation` für den Namen der Methode, `Arguments` für die Definition der Parameter, welche der Methode übergeben werden, und `Return Type` für den Typ des Rückgabewertes. StP überprüft aber nicht die Syntax der Definitionen.

Die ersten beiden Spalten der *IDL Items* dienen wiederum dazu, IDL-spezifische Typdefinitionen für Parameter und Rückgabewert angeben zu können. Der einzige mögliche Wert für das Feld `Attribute` ist «oneway», welcher besagt, daß die Methode von einem CORBA-Objekt asynchron aufgerufen wird und somit auch kein Wert zurückgegeben wird. Die Aufrufsemantik ist in diesem Fall „*best effort*“.

IDL-Ausnahmen, die die Methode erzeugen kann, werden in der Spalte `Raises` angegeben. Das CORBA-Äquivalent zu Umgebungsvariablen, von deren Belegung die Ausführung der Methode beinflußt wird, kann im Feld `Context` definiert werden. Das Context-Objekt besteht aus Tupeln für Bezeichner und Wert.

Hinzufügen von IDL-Code und Kommentaren durch den Benutzer

Die erzeugten IDL-Objektbeschreibungen werden vom IDL-Compiler nur akzeptiert, wenn Attribute, Parameter und Rückgabewerte einfache IDL-Datentypen besitzen. Benutzerdefinierte und komplexe Datentypen müssen vor Verwendung deklariert werden (IDL-Schlüsselwort `typedef`). Diese Deklarationen in Form von IDL-Code können mit dem *Object Annotation Editor* als Anmerkungen ganzen Klassen oder einzelnen Attributen bzw. Methoden zugeordnet werden. Bei der Erzeugung des IDL-Codes für die Klasse werden die Anmerkungen der IDL-Datei hinzugefügt.

Beispielsweise soll das Attribut `Operational State` vom Typ opState sein. Der Typ opState ist eine Aufzählung der Werte «enabled», «disabled» und «unknown». Der Klasse wird nun die IDL-Typdeklaration für opState als Anmerkung hinzugefügt:

```
enum opState { UNKNOWN, ENABLED, DISABLED };
```

Im Kapitel 5.1.4 wurde das dynamische Modell dazu benutzt, um für eine Managementobjektklasse asynchrone Meldungen zu definieren. Da anhand des dynamischen Modells kein Code erzeugt werden kann, gehen die Definitionen in der IDL-Objektbeschreibung verloren. Über den Anmerkungsmechanismus besteht die Möglichkeit, diese Definitionen als Kommentare in die IDL-Datei der Klasse einzufügen.

Die Dialogbox zur IDL-Code-Erzeugung

Das Kommando Generate IDL for <object> des *Class Browser* öffnet die Dialogbox, die in Abbildung 6.4 abgebildet ist. Hier werden Parameter eingestellt, die die Code-Erzeugung insgesamt beinflussen.

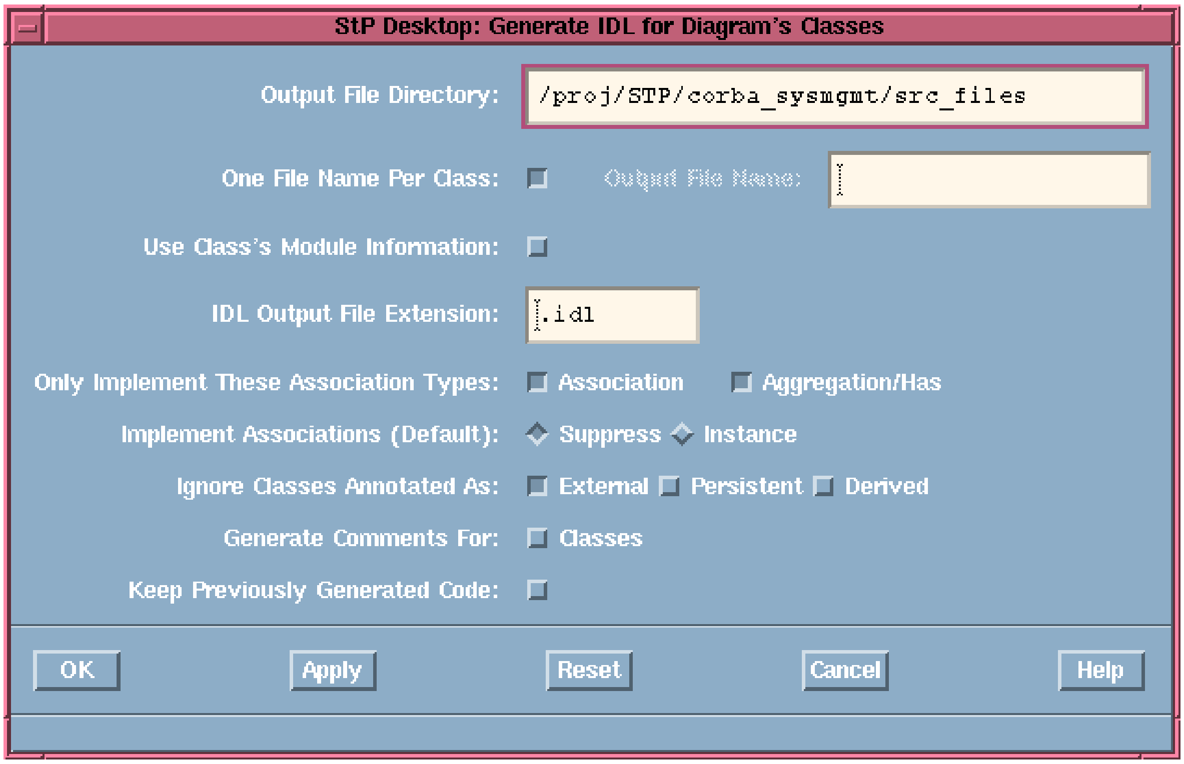


Abbildung 6.4: Die Dialogbox zur IDL-Code-Erzeugung

Nach Angabe des Verzeichnises, in das die IDL-Dateien geschrieben werden sollen, muß entschieden werden, ob für jede Klasse eine eigene IDL-Datei erstellt werden soll. Der IDL-Compiler kann den Code für eine vererbte Unterklasse jedoch nur verarbeiten, wenn dieselbe IDL-Datei auch die Deklarationen für die Oberklasse(n) enthält. Normalerweise fügt StP ein Include-Kommando für die IDL-Datei der Oberklasse ein. Alternativ dazu kann der Code für ausgewählte bzw. für alle Klassen eines Diagramms auch in *einer* Datei erzeugt werden, für die ein Name angegeben werden muß.

Die Option „*Use Class's Module Information*“ hat für OMT-Modelle keine Bedeutung. Im nächsten Feld kann die Extension der IDL-Dateien bestimmt werden.

Die nächsten beiden Optionen legen fest, ob StP auch für Assoziationen bzw. Aggregationen Code einfügen soll. Wird «Instance» selektiert, werden assozierte Objektinstanzen als zusätzliche Attribute der Klasse eingebettet. Beispielsweise entsteht aus der 1:n-Assoziation zwischen der Klasse **UNIXSystem** und **UNIXProcess** das folgende Attribut:

```
attribute sequence<UNIXProcess> assnUNIXProcess;
```

Diese Einbettung von Instanzen assoziierter Objektklassen führt bei der Implementierung der CORBA-Objekte jedoch zu Problemen, da ein Mechanismus zur Initialisierung der „Beziehungsattribute“ fehlt. Für den Prototypen wurde daher auf die Code-Erzeugung für Assoziationen und Aggregationen durch «Suppress» verzichtet. Beziehungen aus dem Objektmodell und deren Einschränkungen durch die Multiplizitätsangaben müssen daher bei der Implementierung der Server-Objekte extra beachtet werden. Wünschenswert wäre die Abbildung von Beziehungen auf den CORBA Relationship-Service. Dies wird aber weder in der eingesetzten Version von StP unterstützt, noch enthält der *VisiBroker for Java* diesen Service.

Für die übrigen Parameter können die Default-Einstellungen verwendet werden. Die genaue Bedeutung kann [Int96d] entnommen werden.

Beispiel

Abbildung 6.5 enthält die von StP erzeugten IDL-Schnittstellenbeschreibungen für die Klasse **nucleus**.

Nach der beschriebenen Vorgehensweise werden für alle Klassen, die implementiert werden sollen, IDL-Dateien generiert.

6.3 Implementierung der CORBA-Objekte für den Agenten

Wie in Abbildung 2.4 dargestellt, ist der nächste Schritt bei der Implementierung des CORBA-Agenten das Übersetzen der IDL-Objektbeschreibungen in Code-Gerüste der Implementierungssprache. Der hierzu erforderliche IDL-Compiler ist Teil der CORBA-Entwicklungs- und Laufzeitumgebung *Visibroker for Java*, die im folgenden Abschnitt vorgestellt wird.

6.3.1 Überblick über VisiBroker for Java

Für die Entwicklung des Prototypen wurde das Produkt *VisiBroker for Java* der Firma *Visigenic*[1], das seit April 1996 erhältlich ist, eingesetzt. Die Implementierung basiert auf der Version 3.0 vom September 1997. Zu diesem Zeitpunkt wurde der Markt der Java-ORBs vor allem von drei Produkten beherrscht: *Visigenic's VisiBroker for Java, IONA's OrbixWeb* und *Sun's Joe*. Die Auswahl des geeignetsten Produkts war Aufgabe vor Beginn der Implementierung. Die Entscheidung wurde anhand des Vergleichs der ORBs in [VD97] und durch einen Test der frei verfügbaren Evaluierungsversionen von *OrbixWeb* und *VisiBroker* getroffen. Folgende Anforderungen sollte das Produkt erfüllen:

[1] WWW-Homepage: `http://www.visigenic.com`

```
// StP -- created on Thu Jan 22 11:00:24 1998
#ifndef _nucleus_idl_
#define _nucleus_idl_

// stp class declarations
interface nucleus;
// stp class declarations end

// stp auto-include 13051
#include "engObject.idl"
// stp auto-include end

// stp class definition 13051
interface nucleus : engObject
{
// stp class members
enum opState { UNKNOWN, ENABLED, DISABLED };
enum usState { UNKNOWN, IDLE, ACTIVE, BUSY };
enum adState { UNKNOWN, UNLOCKED, SHUTTING_DOWN,
  LOCKED, NOT_APPLICABLE };

    readonly attribute adState AdminState;
    readonly attribute opState OperationalState;
    readonly attribute usState UsageState;
    attribute string Date;
    attribute string Name;
    readonly attribute string OS;
    attribute string CountryInfo;
    attribute string Timezone;
    readonly attribute long Uptime;

    void shutdown(in Seconds  execTime) raises(NotAuthorized);
    void restart(in Seconds  execTime) raises(NotAuthorized);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Abbildung 6.5: IDL-Code der Klassen **nucleus** und **UNIXSystem**

- Der ORB sollte mit der CORBA-Spezifikation 2.0 der OMG konform sein. Ebenso sollte die Abbildung von IDL nach Java auf dem standardisierten *Java language mapping* der OMG basieren.

- Sowohl Client als auch Server sollten in Java programmiert werden können. Als Client sollte der ORB auch Java-Applets unterstützen.

- Sowohl die Entwicklungswerkzeuge als auch die Laufzeitumgebung (ORB) sollten für die wichtigsten UNIX-Plattformen (IBM AIX, Sun Solaris und HP-UX) zur Verfügung stehen, um CORBA-Agenten in heterogener Umgebung entwickeln und betreiben zu können.

- Das Produkt sollte möglichst viele der bereits standardisierten CORBA-Services beinhalten.
- Guter Support durch den Hersteller sollte gewährleistet sein.

Joe scheiterte an der zweiten Anforderung, da zu diesem Zeitpunkt nur die Entwicklung von Java-Clients möglich war. *OrbixWeb* und *VisiBroker* boten im wesentlichen die gleiche Funktionalität. Die Entscheidung fiel aus folgenden Gründen auf den *VisiBroker*, der die obigen Anforderungen am besten erüllte:

- Die Entwicklungsumgebung und der ORB sind auf allen Plattformen lauffähig, die das JDK ab Version 1.1.2 unterstützen, da sie vollständig in Java implementiert sind. Einzige Einschränkung: Ein zur Laufzeit erforderlicher Dämon, der sog. *Smart Agent*, stand nur für Solaris bzw. Windows NT zur Verfügung. Dieser muß aber nur auf einem Rechner innerhalb des Netzes vorhanden sein. Auf seine Funktion wird in Abschnitt 6.4.1 eingegangen.
- *Visigenic* liefert mit dem optionalen Produkt *Gatekeeper* einen Mechanismus, der es CORBA-Applets unter Umgehung der Kommunikationseinschränkungen für Java-Applets ermöglicht, auf Server-Objekte auf beliebigen Rechnern zuzugreifen. Diese Technik basiert auf Proxy-Objekten kombiniert mit einem Tunneling von IIOP über HTTP (siehe Abschnitt 6.5 und [Vis97a]).
- Als einziges Produkt bot *VisiBroker* zumindest die CORBA-Services *Naming* und *Event*.
- Die Firma *Netscape* hatte angekündigt, den ORB von *Visigenic* sowohl in ihre Browser *Navigator* und *Communicator* als auch in ihre Web-Server zu integrieren. Bis dato wurde diese Integration allerdings nicht vollständig und nur mit der alten ORB-Version 2.5 vollzogen.

Alle Leistungsmerkmale des *VisiBroker* aufzuführen, würde an dieser Stelle zu weit führen. Daher werden in diesem Überblick nur die wichtigsten Funktionen genannt, die für die Implementierung des Prototypen benötigt werden. Für Details sei auf die Manuale [Vis97b] und [Vis97c] verwiesen.

Der IDL-Compiler `idl2java` zur Übersetzung der IDL-Objektbeschreibungen des Objektmodells wird im Abschnitt 6.3.2 beschrieben.

Der ORB ermöglicht die transparente Kommunikation zwischen den verteilten CORBA-Objekten. Ruft ein Client eine Methode auf einem ihm bekannten Objekt auf, lokalisiert der ORB das entsprechende Objekt, leitet den Aufruf weiter und gibt das Ergebnis an den Client zurück. Der ORB ist kein separater Prozeß, sondern eine Sammlung von Java-Klassen in einem Archiv, die zur Laufzeit von der Java-VM zu den Clients bzw. Servern hinzugebunden werden. Diese Klassen müssen auf allen Rechnern installiert sein, auf denen CORBA-Objekte laufen sollen.

Der *Smart Agent* ist ein Dämon zur einfacheren Lokalisation von CORBA-Objekten. Es handelt sich hierbei um eine proprietäre Erweiterung der CORBA-Spezifikation. Server-Objekte registrieren sich bei ihrer Aktivierung bei dem Agenten. Mit seiner Hilfe können sich Clients über den Objektnamen und dem optionalen Hostnamen an ein Server-Objekt binden, das heißt, dessen IOR ermitteln. Diese Funktionalität ist nicht mit dem CORBA Naming-Service zu verwechseln. Tatsächlich sollte dieser Dienst eine standardisierte Lokalisierungsfunktion bereitstellen und den *Smart Agent* überflüssig machen. Unglücklicherweise ist der Betrieb des *Visigenic* Naming-Service auch nur bei vorhandenem *Smart Agent* möglich, da der ORB den Bootstrap-Mechanismus über die Operation `list_initial_references()` noch nicht unterstützt. Für den Fall, daß sich der Einsatz des *Smart Agent* verbietet, weil z.B. auf Objekte über einen Fremd-ORB zugegriffen wird, muß der umständliche und komplizierte Umweg über sog. *stringified IORs*, die in fest definierten Dateien abgelegt werden, gegangen werden. Der *Smart Agent* wird über das Kommando `osagent [-v] &` gestartet. Er ist ausschließlich unter Solaris bzw. Windows NT lauffähig, muß aber nur einmal im Netz vorhanden sein, solange er von allen Objekten per UDP-Broadcast erreichbar ist.

Bei Verwendung des *Smart Agent* kann auch eine automatische Objektaktivierung genutzt werden. Normalerweise müssen die CORBA-Objekte über ein Server-Programm kreiert und aktiviert werden (siehe Abschnitt 6.3.5), bevor ein Client auf sie zugreifen kann. Wird auf jedem Rechner mit CORBA-Objekten der sog. *Object Activation Daemon* (OAD) gestartet, kann der *Smart Agent* Objekte bei Bedarf, das heißt bei Aufruf durch einen Client, starten. Dazu müssen die Objekte mit Name und IDL-Schnittstelle mit Hilfe des Werkzeugs `oadutil` einmalig beim OAD, der das CORBA *Implementation Repository* implementiert, registriert werden.

Weiterhin bietet *VisiBroker for Java* ein *Interface Repository* (IR) zur Registrierung von Objektschnittstellen, welches über Kommandozeilen-Tools oder zur Laufzeit von Objekten abgefragt werden kann.

Für den Betrieb des *VisiBroker* müssen folgende Umgebungsvariablen gesetzt bzw. angepaßt werden.[2]

PATH: Ergänzen um die Pfade zu den JDK- und VisiBroker-Executables. Es ist unbedingt darauf zu achten, daß sich nur der Java-Compiler und die Java-VM des JDK 1.1.2 oder höher im Suchpfad befinden. Anderenfalls treten schwer zu lokalisierende Fehler auf.

CLASSPATH: Ergänzen um die Pfade zu den Klassenarchiven für den ORB und die CORBA Services.

VBROKER_ADM: Verzeichnis, auf das der *Smart Agent* schreibend zugreifen darf. Enthält Log-Dateien, Interface und Implementation Repository.

OSAGENT_ADDR: (optional) IP-Adresse des Rechners, auf dem der *osagent* läuft. Wird andernfalls über einen UDP-Broadcast ermittelt.

OSAGENT_PORT: (optional) Portnummer des *osagent*, falls nicht der Default-Port 14000 verwendet wird.

LD_LIBRARY_PATH: Bei Verwendung des JNI (siehe Abschnitt 6.3.4) muß das Verzeichnis für die erzeugten Bibliotheken angegeben werden.

[2]Ein geeignetes Shellskript findet sich in Anhang B.

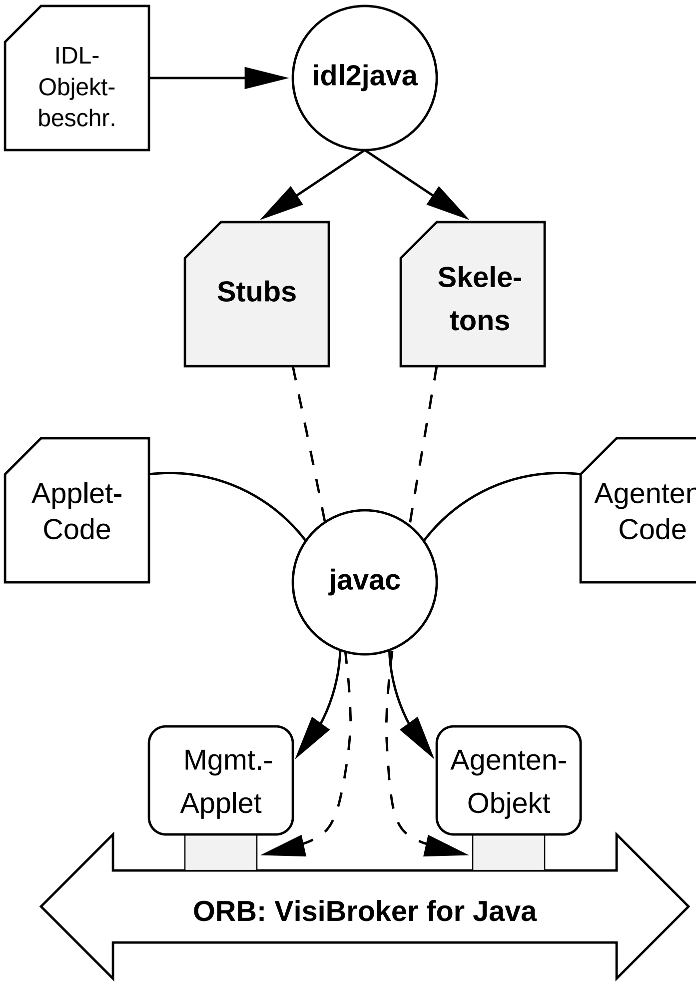


Abbildung 6.6: Implementierung der CORBA-Objekte

6.3.2 Übersetzung der IDL-Dateien in Java-Code

Aus den IDL-Beschreibungen der Managementobjektklassen entstehen bei der Implementierung CORBA-Server-Objekte, die den Agenten bilden. Der IDL-Compiler `idl2java` übersetzt die IDL-Objektbeschreibungen. Seit Version 3.0 des *VisiBroker for Java* ist die Abbildung von IDL auf Java konform mit dem von der OMG verabschiedeten *language mapping*. Bei der Übersetzung erzeugt der IDL-Compiler *Client Stubs* und *Server Skeletons* in Form von Java-Dateien. Die Skeletons ermöglichen die Kommunikation der Server-Objekte mit dem ORB. Hierzu werden sie zusammen mit dem Code, der die Managementinformation und Funktionalität eines Objekts realisiert, mit dem Java-Compiler `javac` übersetzt. Es entsteht ein Agentenobjekt, welches beim *Implementation Repository* des ORB registriert werden kann. Die Stubs ermöglichen es einem Client, über den ORB auf ein Agentenobjekt zuzugreifen. In unserem Fall ist der Client das Management-Applet (siehe Abschnitt 6.4). Das lauffähige Applet entsteht durch Übersetzen des Applet-Codes und der Stubs durch den Java-Compiler. Abbildung 6.6 verdeutlicht die weitere Implementierung. Die schattierten Komponenten werden automatisch generiert.

Der VisiBroker-IDL-Compiler und -ORB sind reine Java-Programme. Für die Implementierung wurde das *Java Development Kit* für AIX von IBM in der Version 1.1.3 eingesetzt.

Bei `idl2java` handelt es sich um ein Shellskript, welches die Java-VM mit der Compiler-Klasse aufruft. Zur Übersetzung einer IDL-Datei genügt im Normalfall das Kommando `idl2java <dateiname>`. Die IDL-Datei muß die Endung .idl besitzen. Optionale Parameter zu `idl2java` sind in [Vis97c] erläutert. Am Beispiel der Klase **nucleus** werden die von `idl2java` erzeugten Java-Dateien erklärt:

nucleus.java: Dies ist die Java-Interface-Deklaration zum IDL-Interface der Klasse **nucleus**. Das Java-Interface enthält für jedes Attribut der Klasse eine Methode zum Auslesen und zum Schreiben, falls das Attribut nicht als *read only* gekennzeichnet war. Für das schreibbare Attribut `Name` werden also folgende Methoden erzeugt:

```
public java.lang.String Name();
public void Name(java.lang.String Name);
```

Zusätzlich enthält das Interface Deklarationen für jede Methode der IDL-Klassenbeschreibung.

nucleusHelper.java: Diese Klasse definiert Hilfsfunktionen wie `narrow()`, `bind()` oder `id()`, die auf den zugehörigen CORBA-Objektinstanzen arbeiten.

nucleusHolder.java: Dies ist eine Hilfsklasse, die intern zur Übergabe von Parametern an die CORBA-Objektinstanz benötigt wird.

_st_Nucleus.java: Diese Klasse enthält den Code für den *Client Stub*.

_nucleusImplBase.java: Diese Klasse ist das zugehörige *Server Skeleton*.

_tie_nucleus.java: Diese Klasse wird benötigt, wenn der sog. Tie-Mechanismus zur Implementierung des Objekts eingesetzt wird.

_nucleusOperations.java: Dieses Java-Interface wird ebenfalls für den Tie-Mechanismus benötigt.

_example_nucleus.java: Code-Gerüst, welches als Basis der Implementierung für das Server-Objekt dienen kann.

Weiterhin werden im Unterverzeichnis nucleusPackage Hilfsklassen für Attribute erzeugt, deren IDL-Datentypen nicht auf einfache Java-Datentypen abgebildet werden können. Dies ist bei der Klasse **nucleus** bei den Statusattributen der Fall. Außerdem werden aus Gründen der Kompatibilität zu älteren Versionen von *VisiBroker* noch einige weitere Java-Klassen generiert, die hier nicht weiter beschrieben werden. Mit Ausnahme der Datei _example_nucleus.java muß *keine* der automatisch generierten Dateien für die Implementierung verändert werden.

6.3.3 Implementierungsansätze für die Agentenobjekte

Zur Realisierung des Agentenobjekts **nucleus** muß eine Klasse erstellt werden, die die Methoden implementiert, die im Java-Interface nucleus.java deklariert sind. Der Konvention nach sollte diese Klasse nucleusImpl heißen. Die zugehörige Java-Datei hat demnach den Namen nucleusImpl.java. Normalerweise wird zur Implementierung der CORBA-Objekte der sog.

BOA-Ansatz (*BOA implementation approach*) benutzt. Beim BOA-Ansatz ist es erforderlich, daß die selbst entwickelte Klasse (hier:**nucleusImpl**) die Skeleton-Klasse (hier: **_nucleusImplBase**) erweitert (*extends*). Da Java aber für Methodenimplementierungen nur Einfachvererbung unterstützt, könnte das Objekt keine Methodenimplementierungen an eine Unterklasse im Objektmodell vererben. Diese Problematik wird anhand der Unterklasse **UNIXSystem** von **nucleus** in Abbildung 6.7 dargestellt.

Soll die Klasse **UNIXSystemImpl** die Implementierung der Methoden von der Klasse **nucleusImpl** erben, kann sie nicht gleichzeitig die Skeleton-Klasse **_UNIXSystemImplBase** erweitern. Die Lösung für dieses Problem ist der sog. Tie-Ansatz, der einen Delegierungsmechanismus realisiert. Dabei wird ein Pseudo-Objekt (Proxy) generiert, das von der Skeleton-Klasse erbt. Anstatt aber die im IDL-Interface definierten Operationen selbst zu implementieren, leitet es entsprechende Aufrufe an ein separates Implementierungsobjekt weiter, das wiederum von einer beliebigen Klasse erben kann. Der Tie-Mechanismus kann auch eingesetzt werden, falls *ein* Java-Objekt mehrere IDL-Interfaces implementieren soll. Ein Anwendungsfall wäre z.B. ein Server-Objekt eines Dienstes, das neben seiner Dienstschnittstelle auch eine Managementschnittstelle implementiert.

Für das hier betrachtete Beispiel bedeutet der Tie-Mechanismus, daß die Implementierungsklasse von der Proxy-Klasse **_tie_UNIXSystem** vertreten wird, welche die Skeleton-Klasse erweitert. Die Proxy-Klasse empfängt die Methodenaufrufe für das Server-Objekt und delegiert sie an die Klasse **UNIXSystemImpl** weiter, welche den Code für die Methoden enthält. Diese implementiert (*implements*) also die Methoden, die in dem Java-Interface UNIXSystem-

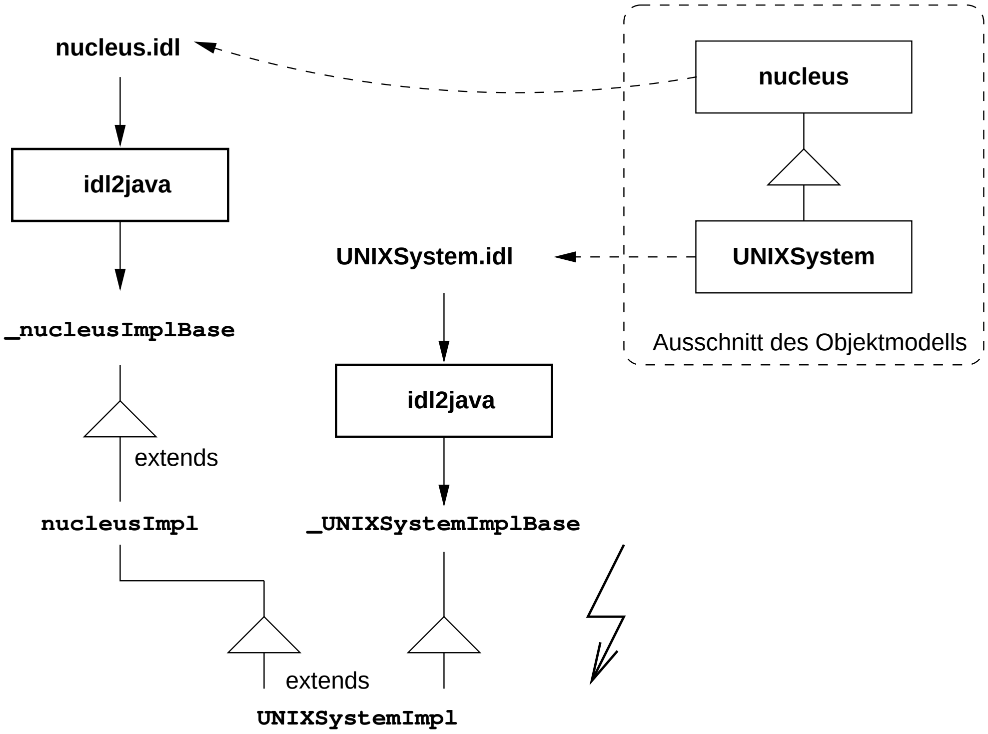


Abbildung 6.7: Vererbungskonflikt der Skeleton- und Implementierungsklassen beim BOA-Ansatz

Operations.java definiert sind, und kann gleichzeitig die Implementierungen von Methoden aus **nucleusImpl** erben. Die Auflösung des Konflikts über den Tie-Ansatz ist in Abbildung 6.8 dargestellt.

Durch das Proxy-Objekt wird zur Laufzeit zusätzlicher Overhead eingeführt. Der Tie-Mechanismus sollte also nur verwendet werden, wenn ein Objekt die Methodenimplementierungen einer Oberklasse erben muß. Falls das Objekt die Methoden der Oberklasse selbst implementiert, das heißt, falls es sich um abstrakte Methoden handelt oder die Methoden überschrieben werden, sollte der normale BOA-Ansatz eingesetzt werden. Hierbei erweitert (*extends*) die Implementierungsklasse die Skeleton-Klasse und implementiert (*implements* die Schnittstelle der Oberklasse. Die beiden Implementierungsansätze werden ausführlich in [Vis97b] beschrieben.

Im folgenden Beispiel für die Implementierung der Klasse **UNIXSystem** des Objektmodells wird der BOA-Ansatz gewählt. Abbildung 6.9 zeigt Ausschnitte der Datei UNIXSystemImpl.java:

Die Klasse **UNIXSystemImpl** erweitert die Skeleton-Klasse und implementiert das Interface **nucleus**. Beim Instantiieren dieser Klasse wird durch Aufruf des Konstruktors der Skeleton-Klasse und Übergabe eines Namens ein persistentes CORBA-Objekt kreiert. Für jedes Attribut der Klassen **nucleus** und **UNIXSystem** gibt es eine Methode zum Auslesen des Attributwerts. Für schreibbare Attribute gibt es zusätzlich eine Methode zum Setzen des

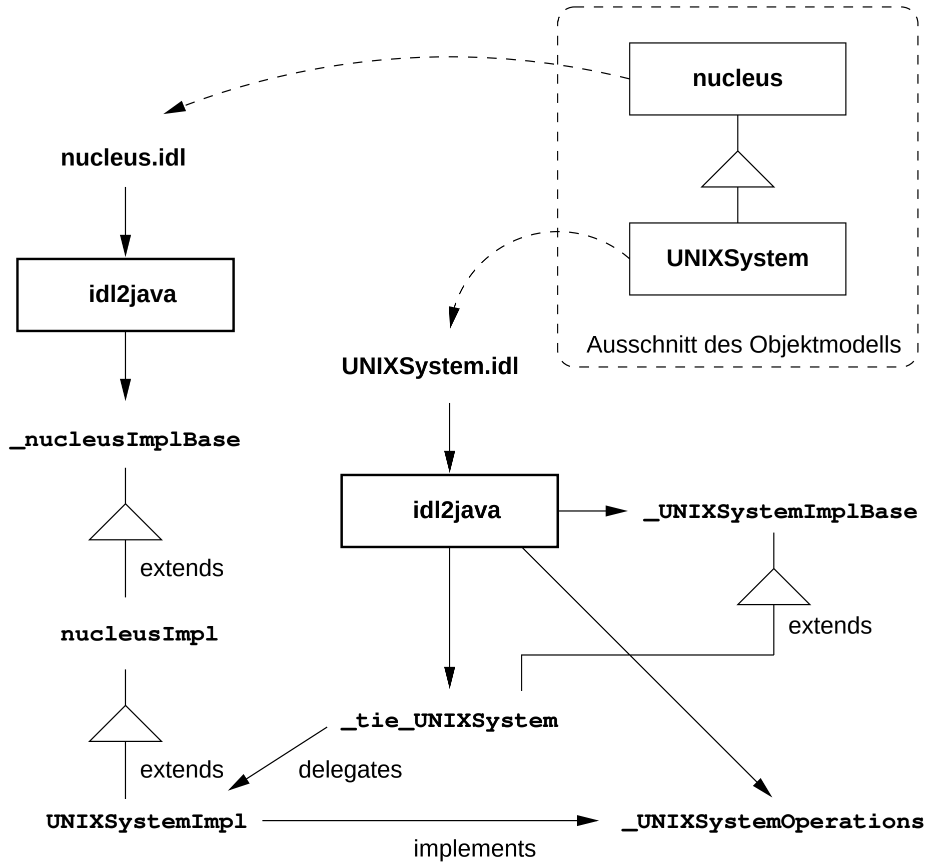


Abbildung 6.8: Vererbungen des Tie-Ansatzes

```
public class UNIXSystemImpl extends _UNIXSystemImplBase
  implements nucleus {

  // Konstruktor fuer ein persistentes CORBA-Objekt
  public UNIXSystemImpl(java.lang.String name) {
    super(name);
  }

  // Attribute von nucleus

  public int Uptime() {
    // Code, um Attribut Uptime auszulesen
  }

  public java.lang.String Name() {
    // Code, um Hostnamen auszulesen
  }

  public void Name(java.lang.String Name) {
    // Code, um Hostnamen zu schreiben
  }

  // [... weitere Attribute ausgelassen]

  // Attribute von UNIXSystem

  public int curProcessNumber() {
    // Code, um Nummer des aktiven Prozesses auszulesen
  }

  // [... weitere Attribute ausgelassen]

  // Methoden

  public void shutdown( int execTime ) {
    // Code fuer shutdown
  }

  // [... weitere Attribute ausgelassen]
}
```

Abbildung 6.9: Ausschnitt aus der Implementierungsdatei UNIXSystemImpl.java

Attributwerts. Außerdem erbt **UNIXSystemImpl** u.a. die Methode `shutdown()`. Als Vorlage zur Implementierung könnte dieses Gerüst aus der Datei _example_UNIXSystem, die der IDL-Compiler erzeugt, übernommen werden. Der Code zur Implementierung der Methoden wurde in der Abbildung weggelassen.

Bei der Realisierung des Prototypen konnte die Managementinformation und -funktionalität der Agentenobjekte über zwei grundsätzlich verschiedene Techniken implementiert werden. Bei Verwendung der Sprache Java stehen Betriebssystemaufrufe zur Ermittlung von Informationen für das Systemmanagement nicht direkt zur Verfügung. Aufgrund der Portabilität von Java werden viele für das Management wichtige Systemdetails durch die *Virtual Machine* verschattet. Eine Möglichkeit war daher der Aufruf von externen UNIX-Kommandos und Programmen, um Managementinformation zu beschaffen oder Aktionen auszuführen. Sie wird im folgenden an einem Beispiel erläutert. Die andere Technik benutzt das *Java Native Interface*, um C-Prozeduren eines bestehenden SNMP-Agenten auszuführen. Diese Technik wird im Abschnitt 6.3.4 beschrieben.

Das Betriebssystem AIX stellt sehr viele High-Level-Administrationsbefehle auf der Ebene der UNIX-Kommandozeile bereit. Es ist möglich, aus der Java-VM heraus externe Programme oder Shellskripte aufzurufen. Die Ausgabe des externen Prozesses kann vom Java-Prozeß gelesen und ausgewertet werden. Im folgenden Beispiel wird die Methode `Name()` zum Auslesen des Hostnamens durch Aufruf des externen AIX-Kommandos `hostname` realisiert. Abbildung 6.10 zeigt den zugehörigen Java-Code.

```
public final String hostnameCmd = "/usr/sbin/hostname";

private Runtime shell = Runtime.getRuntime();  // Laufzeitumgebung
private Process process;                       // Prozess-Objekt
private InputStream input;                     // InputStream zum Lesen...
BufferedReader reader;                         // ...der Standardausgabe
private boolean ok = true;

public java.lang.String Name()
{
  String hostname = "";
  ok = true;
  try {
    process = shell.exec(hostnameCmd);         // fuehrt hostname aus
    try {
      if (process.waitFor() != 0)              // hole Rueckgabewert
        ok = false;
    }
    catch (InterruptedException e) { e.printStackTrace(); }
    input = process.getInputStream();
    reader = new BufferedReader(new InputStreamReader(input));

    hostname = reader.readLine();              // Lese Hostnamen
  }
  catch (IOException e) {                      // Fehlerbehandlung
    e.printStackTrace();
  }
  finally {
    try { error.close(); } catch (IOException e) { e.printStackTrace(); }
    if (!ok)
      throw new CommandExecErrorException("Command execution failed!");
  }
  return hostname;                             // gebe Hostnamen zurueck
}
```

Abbildung 6.10: Implementierung der Methode `Name()`

6.3.4 Das Java Native Interface (JNI)

Das *Java Native Interface* (JNI) wurde entwickelt, um Java-Programmen, die innerhalb der Java-VM laufen, die Möglichkeit zu geben, mit Programmen zusammenzuarbeiten oder Bibliotheksfunktionen zu nutzen, die in einer anderen Programmiersprache (z.B. C oder Assembler) geschrieben sind. JNI besorgt die Umwandlung der Datenformate bei der Übergabe von Parametern zwischen dem Java-Programm und der Fremdfunktion (*native method*). Der Fremdprozeß hat zudem die Möglichkeit auf Klassen und Objekte des Java-Programms zuzugreifen und kann Ausnahmen erzeugen. Der Einsatz des JNI ist erforderlich, wenn eine Java-Anwendung plattformabhängige Funktionen nutzen möchte, die nicht von den Standardklassen des JDK unterstützt werden. Ein weiteres Szenario für die Nutzung des JNI ist der Fall, daß bestimmte Funktionen bereits von einer bestehenden Bibliothek bereitgestellt werden und diese von dem Java-Programm genutzt werden sollen.

Für den CORBA-Agenten treffen beide Szenarien zu. Wie oben bereits erwähnt, ist es zum Teil erforderlich, Betriebssystemaufrufe zur Ermittlung von Managementinformation zu nutzen. Außerdem besteht zu dem Objektmodell für das Management von UNIX-Workstations, welches in das generische Objektmodell integriert wurde (siehe Kapitel 4.3), bereits ein SNMP-Agent. Dieser Agent ist in C geschrieben und überwiegend auch für AIX 4.2 portiert (siehe [Sch96d]). Er ist zu einem Großteil modular aufgebaut, das heißt, Funktionen zum Lesen und Schreiben von Attributen sowie zum Ausführen von Aktionen werden von einzelnen Modulen implementiert. Hierdurch können Funktionen, die von einem Agentenobjekt aufgerufen werden sollen, zu einer Bibliothek gebunden werden, die über das JNI angesprochen werden kann.

Für den Prototypen wurde das Release 1.1 des JNI eingesetzt. Die genaue Spezifikation findet sich in [Jav97]. Ferner gibt es von der Firma *JavaSoft* ein WWW-Tutorial[3] zum JNI. Anhand eines einfachen Beispiels soll die Verwendung des JNI innerhalb des Agentenobjekts für die Klasse **UNIXSystem** erläutert werden. Das Attribut `Uptime` gibt die Zeit seit dem letzten Neustart des Betriebssystems in Sekunden an. Sie soll von der C-Routine get_sysUptime.c (Quellcode in Anhang B) des SNMP-Agenten ermittelt werden. Folgende Schritte sind erforderlich:

Definition der nativen Methode in der Java-Klasse

Fremdfunktionen werden innerhalb des Java-Codes wie normale Methoden deklariert. Der einzige Unterschied ist das zusätzliche Schlüsselwort *native*. Parameter und Rückgabewert werden ebenfalls als Java-Datentypen bzw. Objekte angegeben. Abbildung 6.11 zeigt die Deklaration für die Methode `get_sysUptime()`. Der C-Code, der die Funktion implementiert, wird übersetzt und zu einer *shared library* gebunden (siehe unten). Diese Bibliothek wird beim Instantiieren der Klasse über einen *static initializer* (siehe Abbildung 6.11) automatisch geladen. Hierdurch wird die Implementierung der Fremdfunktion auf die Deklaration der *native method* abgebildet. Natürlich kann die Bibliothek mehrere Fremdfunktionen enthalten. Der Name der Bibliothek kann beliebig, aber ohne Extension, gewählt werden. Auf UNIX-Systemen hängt die Java-VM die Endung .so an den Namen an. Anschließend muß die Klasse mit dem Java-Compiler übersetzt werden.

[3] `http://java.sun.com/products/jdk/1.1/docs/guide/jni/index.html`

```
public class UNIXSystemImpl extends _UNIXSystemImplBase
  implements nucleus {

  // [...]
  // Nur JNI-relevanter Code (ohne Fehlerbehandlung).

  public native int get_sysUptime();    // native method declaration

  static {
    System.loadLibrary("libsysUptime"); // load shared library
  }
  public int Uptime() {
    return get_sysUptime();             // use native method
  }
}
```

Abbildung 6.11: Java-Code für die Nutzung des JNI

Erstellung der Header-Datei

Der nächste Schritt ist die Erstellung einer Header-Datei, welche den Funktionsprototypen zur Implementierung der nativen Methode enthält. Hierzu dient das Java-Utility `javah`. Folgender Aufruf generiert die Header-Datei UNIXSystemImpl.h aus der Java-Klassendatei UNIXSystemImpl.class: `javah -jni UNIXSystemImpl`. Abbildung 6.12 enthält den relevanten Teil der generierten Header-Datei. Für die Methode `get_sysUptime()` wurde die Signatur einer zu implementierenden C-Funktion mit dem Namen `Java_UNIXSystemImpl_get_1sysUptime` erzeugt. Dieser Name setzt sich aus dem Präfix «Java_», dem Namen der Klasse und dem modifizierten Methodennamen zusammen. Diese Funktion stellt einen *Wrapper* für die bestehende Funktion `get_sysUptime()` des SNMP-Agenten dar. Jeder Fremdfunktion werden standardmäßig zwei Parameter übergeben. Der erste ist ein Zeiger auf die JNI-Schnittstelle (*JNI interface pointer*). Über diesen Zeiger (`JNIEnv *`) kann die Fremdfunktion auf Parameter und Objekte zugreifen, die gegebenenfalls vom Java-Programm übergeben wurden. Der zweite Parameter (`jobject`) ist ein Zeiger auf das Objekt (*„this" pointer*), welches die native Methode enthält. Da die Methode `get_sysUptime()` keine Parameter übergibt, werden die beiden Zeiger nicht benötigt.

```
#include <jni.h>
/* Header for class UNIXSystemImpl */
/*
 * Class:     UNIXSystemImpl
 * Method:    get_sysUptime
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_UNIXSystemImpl_get_1sysUptime
  (JNIEnv *, jobject);
```

Abbildung 6.12: Ausschnitt aus der Header-Datei UNIXSystemImpl.h

Implementierung der Wrapper-Funktion

Jetzt gilt es, die Funktion zu implementieren, deren Signatur im letzten Schritt durch `javah` erstellt wurde. Da es sich in diesem Fall nur um einen *Wrapper* um die bereits bestehende Funktion des SNMP-Agenten handelt, muß lediglich die „alte“ Funktion `get_sysUptime()` aufgerufen und das Ergebnis zurückgegeben werden. Der Code hierzu findet sich in Abbildung 6.13. Die Umwandlung und Übergabe des Ergebniswerts von C nach Java wird durch das JNI erledigt. Hierzu muß in jedem Fall die bei der Java-VM mitgelieferte Header-Datei jni.h eingebunden werden. Der letzte Schritt ist die Übersetzung dieser Datei und die Erstellung der *shared library*.

```
#include <jni.h>                /* immer einbinden */
#include "UNIXSystemImpl.h"     /* generierte Header-Datei */
#include "get_sysUptime.h"      /* Header-Datei der SNMP-Agenten-Funktion */

JNIEXPORT jint JNICALL          /* Wrapper-Funktion aus UnixSystemImpl.h */
Java_UNIXSystemImpl_get_1sysUptime (JNIEnv *env, jobject obj)
{
    return *get_sysUptime(0);   /* Aufruf Funktion des SNMP-Agenten */
}
```

Abbildung 6.13: Implementierung des Wrappers UNIXSystemImpl.c

Erstellung der Bibliothek

Zum Erstellen der *shared library* für die C-Funktion, die zur Laufzeit von der Java-VM zu der Klasse **UnixSystemImpl** hinzugeladen wird, ist die Übersetzung der Wrapper-Funktion aus dem letzten Schritt erforderlich. Zusätzlich wird das Object-File get_sysUptime.o vom SNMP-Agenten benötigt, da dieses in die Bibliothek eingebunden wird. Unter Solaris generiert der C-Compiler `cc` bzw. `gcc` eine *shared library*, wenn der Compiler-Parameter „-G“ angegeben wird. Die Bibliothek würde durch folgendes Kommando erstellt werden:

```
gcc -G -I/<path_to_JDK>/include -I/<path_to_JDK>/include/solaris \
       UNIXSystemImpl.c -o libsysUptime.so
```

Unter AIX ist die Erstellung leider sehr viel komplexer. Deshalb ist beim JDK 1.1 von IBM ein JNI-Beispiel[4] mitgeliefert, welches auch ein Makefile zur Erstellung der Bibliothek enthält. Auf dessen Basis wurde für den Prototypen ein einfaches Shellskript create_lib_AIX entwickelt, welches im Anhang B zu finden ist. Damit die Java-VM zur Laufzeit die Bibliothek findet, muß der Pfad in der Umgebungsvariable LD_LIBRARY_PATH gesetzt sein.

Zusammenfassung

In Abbildung 6.14 ist nochmals das Zusammenspiel zwischen Java-Methode und C-Funktionen über das JNI dargestellt. Beim Instantiieren der Java-Klasse wird als erstes die Bibliothek libgetUptime.so geladen. Ein Aufruf der Methode `get_sysUptime()` wird durch das JNI auf den

[4] Pfad: /<path_to_JDK>/jni_example/c

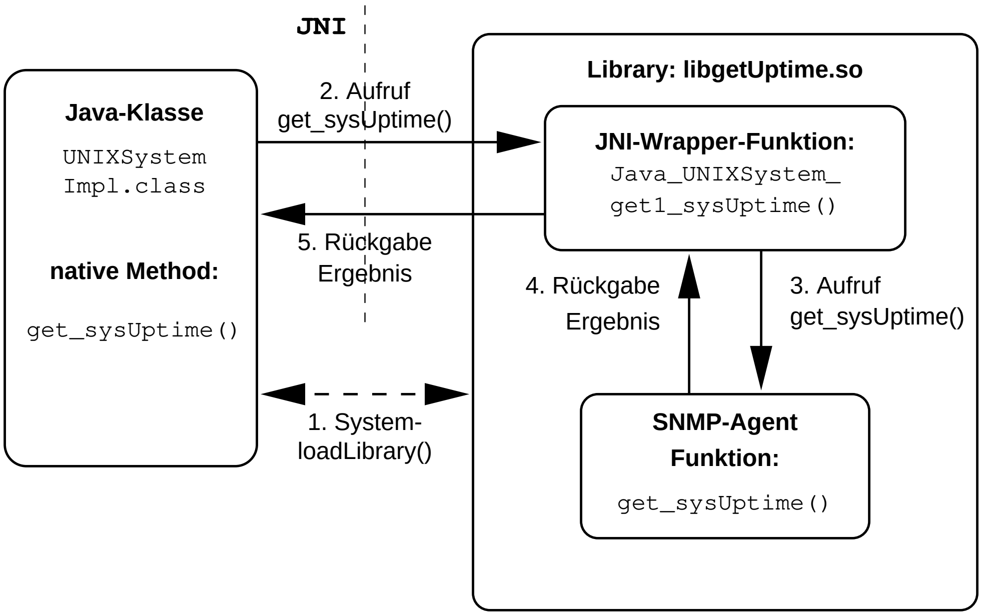


Abbildung 6.14: Der JNI-Wrapper für C-Funktionen

Aufruf der Wrapper-Funktion `Java_UNIXSystem_get1_sysUptime()` abgebildet. Diese Funktion ruft die ursprüngliche Funktion `get_sysUptime()` des SNMP-Agenten auf. Das Ergebnis ist ein C-Integerwert, der vom JNI im letzten Schritt in das Java-Format für Integerzahlen umgewandelt und an das Java-Objekt zurückgegeben wird.

6.3.5 Initialisierung der Agentenobjekte

Nach Implementierung der Methoden zum Lesen und Schreiben der Attribute sowie der Operationen einer Objektklasse liegt ein fertiges Agentenobjekt vor. Bevor aber ein Client, d.h. eine Managementanwendung über CORBA auf das Objekt zugreifen kann, sind folgende Schritte notwendig:

1. Initialisieren des ORB.
2. Initialisieren des BOA.
3. Kreieren von ein oder mehreren Instanzen des Agentenobjekts.
4. Registrieren des Objekts beim BOA.
5. Aktivieren des Objekts.

Für den Prototypen werden diese Schritte vom Server-Programm SystemAgentServer.java für alle Objekte durchgeführt. Die Implementierung wird aber wiederum nur anhand der Klasse **UNIXSystemImpl** erläutert. Abbildung 6.15 enthält den dazugehörigen Code.

Der Aufruf `org.omg.CORBA.ORB.init(args, null);` initialisiert den ORB und gibt eine Referenz auf das ORB-Objekt an den Aufrufer zurück. Der BOA des ORB wird durch Aufruf der Methode `BOA_init();` auf dem ORB-Objekt initialisiert. Als nächstes wird eine Instanz

```
/* Server fuer CORBA/JAVA Systemagent AIX4.2 */

public class SystemAgentServer {

  public static void main( String args[]) {

    try {
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
      System.out.println("ORB initialisiert.");

      org.omg.CORBA.BOA boa = orb.BOA_init();
      System.out.println("BOA initialisiert.");

      // create server object for UNIXSystem
      sysagent.UNIXSystem sys = new UNIXSystemImpl( "UNIXSystem" );
      boa.obj_is_ready(sys);
      System.out.println(sys + " is ready.");

      // create other agent objects
      // [...]

      // wait for requests
      boa.impl_is_ready();
    }
    catch(org.omg.CORBA.SystemException e) {
      System.err.println(e);
    }
  }
};
```

Abbildung 6.15: CORBA-Server SystemAgentServer.java

der Klasse **UNIXSystemImpl** erzeugt. Durch Angabe des Namens „UNIXSystem“ ensteht ein persistentes CORBA-Objekt, welches beim *Smart Agent* registriert wird. Dadurch kann ein Client später die IOR des Objekts über diesen Namen ermitteln. Das neu erzeugte Objekt ist jetzt in der Lage, Anfragen eines Clients entgegenzunehmen. Dies wird dem BOA durch den Aufruf `boa.obj_is_ready(sys);` angezeigt. An dieser Stelle werden normalerweise weitere Agentenobjekte erzeugt, registriert und initialisiert. Durch den abschließenden Aufruf `boa.impl_is_ready();` geht das Server-Programm, das die Agentenobjekte bereitstellt, in eine Endlosschleife, um auf Anfragen von Clients zu warten.

Für Agentenobjekte, die nach dem Tie-Ansatz implementiert sind, sieht die Aktivierung etwas anders aus. Nach Instantiierung der Implementierungsklasse für das Objekt wird zusätzlich ein Proxy-Objekt erzeugt und mit dem Implementierungsobjekt verbunden. Dieses Proxy-Objekt wird anschließend beim BOA ganz gewöhnlich registriert. Ein Code-Beispiel hierfür findet sich in Anhang B.

Die geschilderte Methode zum Erzeugen und Aktivieren von Agentenobjekten funktioniert nur für Klassen des Objektmodells, die pro System nur einmal instantiiert werden und statisch sind, d.h. während der Laufzeit des Agenten nicht gelöscht oder erzeugt werden. Für Managementobjekte, die zahlreich in einem System auftreten und dynamisch erzeugt und gelöscht werden können, bedient sich der Prototyp sog. *Factories*. Beispiele für solche Objekte sind u.a. Prozesse, Benutzerkennungen, Dateisysteme und Software-Pakete.

6.3.6 Factories

Ein Factory-Objekt unterscheidet sich architekturell nicht von einem anderem CORBA-Objekt. Der Unterschied zu gewöhnlichen Objekten liegt ausschließlich in der Semantik. Ein Factory-Objekt stellt Methoden bereit, um dynamisch neue Objekte zu erzeugen. Meist ist ein Factory-Objekt nur für das Erzeugen von Objekten einer bestimmten Klasse zuständig. Beispielsweise wird ein Objekt **FilesystemFactory** Managementobjekte für Dateisysteme kreieren. Auch zu Beziehungsklassen kann es Factories geben. Das Eingehen einer Beziehung entspricht dem Instantiieren der Beziehungsklasse. Exportiert z.B. ein NFS-Server ein Dateisystem, muß das Factory-Objekt **ExportOptionsFactory** ein Objekt der Klasse **ExportOptions** anlegen.

Da das Erzeugen von Objekten vom Mechanismus der entsprechenden Implementierungssprache abhängt, ist die Entwicklung eines generischen Factory-Objekts nicht möglich, das beliebige CORBA-Objekte anlegen kann. Auch der CORBA Lifecycle-Service standardisiert nur das Verschieben, Kopieren und Löschen von Objekten. Neben dem Erzeugen von Objekten und deren Registrierung und Aktivierung beim BOA übernehmen die Factories des Prototypen noch weitere Verwaltungsaufgaben. Aus Performancegründen werden die dynamischen Objekte nur *transient* registriert, d.h. eine Lokalisierung über den *Smart Agent* ist nicht möglich. Die Factory verwaltet über ein einfaches Verzeichnis (Hashtable) „ihre" Objekte selbst. Die Funktionen einer Factory werden am Beispiel des Objekts **AccountFactory** vorgestellt. Dessen IDL-Beschreibung enthält Abbildung 6.16.

```
// AccountFactory.idl

module sysagent {
  typedef sequence<Account> accountList;

  interface AccountFactory {

    Account create(in string name, in string passwd, in long uid,
                   in long gid, in string gecos, in string home,
                   in string shell);
    Account open (in string name);
    accountList list();
    void delete (in string name);
  };
};
```

Abbildung 6.16: IDL-Beschreibung: AccountFactory.idl

Die Methode `create()` dient zum Anlegen einer neuen Benutzerkennung. Gleichzeitig wird ein Managementobjekt der Klasse **Account** erzeugt, im internen Verzeichnis abgelegt und dessen IOR an den Aufrufer zurückgegeben. Durch die Methode `open()` kann ein Client die IOR eines Account-Objekts über das eindeutige Attribut `Name` von der Factory erfragen. Eine Liste aller IORs, die die Factory verwaltet, wird von der Methode `list()` zurückgegeben. Schließlich dient die Methode `delete()` zum Entfernen einer Kennung aus dem Verzeichnis der Factory. Zusätzlich wird das dynamische Objekt gelöscht, in dem es beim BOA abgemeldet wird. An dieser Stelle muß sorgfältig zwischen dem Löschen der Benutzerkennung auf dem System und dem Löschen des zugehörigen Agentenobjekts unterschieden werden. Ersteres wird erreicht, indem die Managementanwendung die Operation `delete()` auf dem Agentenobjekt ausführt, das seinerseits das UNIX-Kommando `rmuser` aufruft. Wenn das Agentenobjekt hierauf versucht, durch die BOA-Methode `deactivate_obj()` sich selbst zu deaktivieren, erzeugt der ORB eine Ausnahme für die Managementanwendung, auch wenn die Methode `delete()` das Attribut «oneway» besitzt.[5] Aus diesem Grund muß der Manager anschließend das Agentenobjekt explizit über die Methode `delete()` der Factory löschen.

Die Implementierung der Klasse **AccountFactoryImpl** ist im Anhang B zu finden. Die Factory-Objekte werden, wie in Abschnitt 6.3.5 beschrieben, vom Server-Programm SystemAgentServer.java kreiert und als persistente CORBA-Objekte registriert. Der Konstruktor der Factory erzeugt für alle auf dem System bereits bestehenden Benutzerkennungen transiente Agentenobjekte. Die Managementanwendung (siehe 6.4) bindet sich an die Factory, um eine Liste aller IORs für die Account-Objekte zu erhalten. Zum Lesen bzw. Verändern der Attribute einer Kennung greift sie schließlich direkt auf das zugehörige Account-Objekt zu.

Für sehr dynamische Managementobjekte wie Prozesse oder Druckjobs ist es äußerst problematisch, in Echtzeit zugehörige Agentenobjekte bereitzustellen. Ein Polling der Prozeßtabelle in kurzen Abständen durch ein Factory-Objekt würde zu immensen Ressourcenbedarf des Agenten führen, wenn gleichzeitig Agentenobjekte angelegt bzw. gelöscht werden müssen. Eine mögliche Lösung ist das Bereitstellen der Objekte bei Bedarf, d.h. wenn eine Anfrage des Managers vorliegt. Eine andere Möglichkeit ist die Aktualisierung in größeren Abständen. Dabei besteht aber die Gefahr, daß der Manager über ein Agentenobjekt versucht, auf ein bereits nicht mehr existierendes Managementobjekt zuzugreifen.

In den letzten Abschnitten wurde gezeigt, wie CORBA-Agentenobjekte Managementinformation und -funktionalität der Klassen des Objektmodells implementieren können. Im folgenden Abschnitt wird die Abbildung von asynchronen Meldungen auf den CORBA Event-Service dargestellt.

6.3.7 Abbildung asynchroner Meldungen auf den Event-Service

Die gewöhnliche Kommunikationsform innerhalb CORBA ist der synchrone Methodenaufruf eines Clients auf einem Server. Der CORBA Event-Service entkoppelt die Kommunkikation durch Einführung eines *event channel*. Hierüber ist eine asynchrone Kommunikation zwischen mehreren Sendern (*suppliers*) und Empfängern (*consumers*) nach dem sog. *publish/subscribe*-Paradigma möglich. Eine allgemeine Einführung zum CORBA Event-Service bietet [Sie96]. Ein sehr gutes Tutorial ist [Sch96a]. Die Referenz zur VisiBroker-Implementierung ist [Vis97d].

Asynchrone Meldungen der Agenten an einen oder mehrere Manager werden auf das *push model* des Event-Services abgebildet, welches in Abbildung 6.17 für zwei Sender und einen Empfänger dargestellt ist. „*Push*“ drückt aus, daß der Sender die Kommunikation initiiert, indem er eine Nachricht an den Event Channel sendet. Dieser verteilt wiederum die Nachricht unaufgefordert an alle Empfänger (Manager). Umgekehrt verläuft die Kommunikation beim *pull model*. Hier würde der Event Channel regelmäßig bei den Agenten nachfragen („*pull*“), ob Nachrichten vorliegen und diese zwischenspeichern. Der Manager würde bestimmen, wann er die Nachrichten abholt. Der Event-Service unterstützt auch das Mischen, bei dem das

[5] Diese Problematik wurde auch ohne konkrete Lösung in der Usenet-Newsgroup comp.object.corba diskutiert.

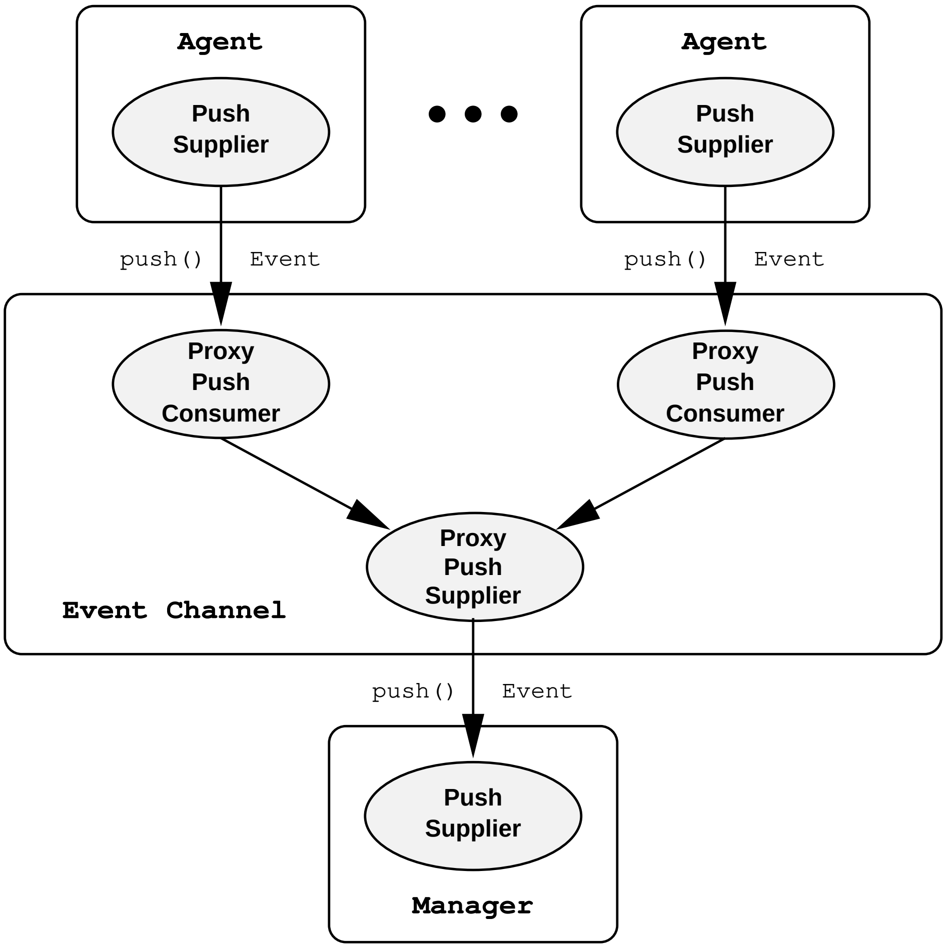


Abbildung 6.17: Abbildung asynchroner Meldungen auf das Push-Modell

Modell für die Sender unterschiedlich ist von dem der Empfänger. Für das Management liefert das Push-Modell die richtige Semantik, da ein Ereignis innerhalb eines Managed Object den Agenten veranlaßt, eine asynchrone Meldung an den Manager zu senden, die dieser sofort erhalten soll, um darauf reagieren zu können.

Wenn ein Agentenobjekt asynchrone Meldungen versenden soll, muß es das *PushSupplier*-Interface implementieren. Ein Manager implementiert dementsprechend das *PushConsumer*-Interface. Wie in Abbildung 6.17 dargestellt, ist jedem Agent im Event Channel ein *ProxyPushConsumer*-Objekt zugeordnet, welches Meldungen an alle *ProxyPushSupplier*-Objekte (hier nur eines) innerhalb des Kanals weiterleitet, die ihrerseits die Methode `push()` auf dem *PushConsumer*-Interface „ihres" Managers aufrufen, um die Nachricht zuzustellen. Zur Anbindung eines Agenten bzw. Managers an einen Event Channel bedarf es mehrerer Schritte. Tabelle 6.1 enthält die zugehörigen Methoden und Klassennamen.

Ein Event Channel, der zunächst noch keine Proxy-Objekte enthält, wird vom Hauptprogramm des Agenten (SystemAgentServer.java) erzeugt und beim BOA angemeldet. Hierfür stellt VisiBroker eine Bibliotheksfunktion bereit. Der Code ist im Anhang B zu finden. Der erste Schritt für ein Agentenobjekt ist, die IOR des eingerichteten Event Channel zu ermit-

Schritt	**Agent: PushSupplier**	**Manager: PushConsumer**
1. Lokalisieren des Event Channel	`EventChannelHelper :: bind()`	`EventChannelHelper :: bind()`
2. Referenz auf Proxy-Factory	`EventChannel :: for_suppliers()`	`EventChannel :: for_consumers()`
3. Proxy-Objekt im Event Channel anfordern	`SupplierAdmin :: obtain_push_consumer()`	`ConsumerAdmin :: obtain_push_supplier()`
4. Proxy-Objekt binden	`ProxyPushConsumer :: connect_push_supplier()`	`ProxyPushSupplier :: connect_push_consumer()`
Senden / Empfangen einer Meldung	`ProxyPushConsumer :: push()`	Implementiert eigene Methode `push()`

Tabelle 6.1: Anbindung an einen Event Channel

teln. Damit es im Event Channel ein Proxy-Objekt erzeugen kann, an das es seine Meldungen schicken kann, erfragt es im zweiten Schritt die IOR einer *ProxyPushConsumer-Factory*. Bei der Factory wird im nächsten Schritt ein solches Proxy-Objekt angefordert (*obtain*). Der letzte Schritt ist, den *PushSupplier* des Agenten mit dem Proxy-Objekt im Kanal zu verbinden (*connect*). Anschließend kann der Agent durch Aufruf der Methode `push()` auf dem Proxy-Objekt Meldungen verschicken. Analog zum Agenten verläuft die Anbindung eines Managers an den Kanal. Der Manager muß ebenfalls eine Methode `push()` implementieren, die das *ProxyPushSupplier*-Objekt im Event Channel zum Zustellen einer Meldung aufruft.

Die Methode `push()` erwartet die Nachricht in Form einer CORBA any-Datenstruktur. *VisiBroker* unterstützt nur die generische, untypisierte Ereigniskommunikation. Zu jeder benutzerdefinierten IDL-Datenstruktur erstellt der IDL-Compiler in der Klasse <datenstruktur>Helper.java die Methoden `insert()` und `extract()`, die Variablen mit dieser Datenstruktur in any-Objekte packen bzw. wieder extrahieren. Ein Event könnte daher als Verbund aus Feldern wie `Name`, `Source`, `Time`, `Severity`, `Cause`, etc. definiert werden.

Für den Prototypen wurde eine Klasse **FilesystemMonitor** implementiert, die den freien Platz bestimmter Filesysteme überwacht und Warnungen an den Manager schickt, sobald der freie Speicherplatz einen festgelegten Prozentsatz unterschritten hat. Die periodische Abfrage innerhalb des Agenten wird durch einen eigenen Java-Thread realisiert. Die Meldungen werden als Strings über den Event-Service an die Managementanwendung gesendet. Anhang B enthält den Quellcode zur Demonstration der Implementierung dieser Techniken.

6.4 Das Management-Applet

Wie im Vorgehensmodell in Kapitel 2.4 beschrieben, soll der Zugriff auf den Agenten komfortabel über eine Managementanwendung mit graphischer Oberfläche möglich sein. Die Anwendung stellt den Client dar, der über den ORB mit den Server-Objekten des CORBA-Agenten kommuniziert (vgl. Abbildung 2.4). Die Vorteile der Implementierung der Anwendung als Java-Applet wurden ebenfalls bereits in Kapitel 2.4 diskutiert. Die graphische Oberfläche kann leicht durch die von Java zur Verfügung gestellten GUI-Elemente realisiert werden. Außerdem ist das entstandene Management-Applet orts- und plattformunabhängig und kann leicht erweitert oder modifiziert werden. Im Vergleich zu einem normalen Java-Programm, welches installiert werden muß und die Java-VM als Laufzeitumgebung benötigt, hat es den Vorteil, in jedem Web-Browser lauffähig zu sein, der Java unterstützt. Der *Communicator* von *Netscape* integriert zudem ab Version 4.0 den ORB von *Visigenic*, so daß Applets unmittelbar Zugriff auf CORBA-Objekte haben. Applets haben aber auch Nachteile. Normalerweise werden Applets innerhalb des Browsers aus Sicherheitsgründen in einer sog. „*sandbox*“ ausgeführt, welche den Zugriff auf Systemressourcen stark einschränkt. Insbesondere dürfen Applets keine Verbindungen über das Netz zu anderen Rechnern aufbauen, als zu dem Server, von dem das Applet geladen wurde. A priori würde dies die Kommunikation des Managers mit Agenten, die in einem verteilten System verstreut sind, unmöglich machen. Die Lösung von *Visigenic* für dieses Problem ist ein zusätzliches Produkt namens *Gatekeeper*. Auf die Funktionsweise des *Gatekeeper* wird später eingegangen. Der zweite Nachteil ist, daß die beiden Web-Browser mit der weitesten Verbreitung, nämlich der *Netscape Navigator* und der *Microsoft Internet Explorer*, (noch) keine Applets unterstützen, die mit dem neueren JDK 1.1.x erstellt wurden[6]. Daher können weder das neue Event-Modell noch die erweiterten GUI-Möglichkeiten des JDK 1.1 verwendet werden.

6.4.1 Lokalisierung der Agentenobjekte

Das Management-Applet nutzt einen proprietären Mechanismus des *VisiBroker* zur Lokalisierung der Agentenobjekte. Der IDL-Compiler erzeugt zu einem IDL-Interface die Klasse `<interface_name>Helper.java`, die eine Methode `bind()` zur Verfügung stellt. Der Methode kann als optionaler Parameter der Hostname des Rechners, auf dem sich das Objekt befindet, und der Name des persistenten Objekts (siehe Abschnitt 6.3.5) übergeben werden. Sie liefert die IOR des Objekts zurück, die sie über den Verzeichnisdienst des *Smart Agent* (osagent) erhält. Voraussetzungen für ein erfolgreiches Binden sind ein gestarteter *osagent* sowie persistente und kommunikationsbereite Objekte. Abbildung 6.18 enthält den Code zur Lokalisierung des Objekts **UNIXSystem**. Mit der IOR kann der Client ganz normal Operationen auf dem Agentenobjekt ausführen. Die Arbeit des ORB, der einen Aufruf an das Server-Objekt weiterleitet und Parameter und Ergebnis übergibt, ist für den Client völlig transparent.

Die Lokalisierung von CORBA-Objekten ist gewöhnlich Aufgabe des standardisierten CORBA Naming-Service. Dieser ermöglicht auch die Objektlokalisierung, wenn Objekte über ORBs unterschiedlicher Hersteller zusammenarbeiten sollen. Für den Prototypen hätte die Verwendung des Naming-Service aber kaum Vorteile gebracht. Einerseits wäre die Implementierung der Agentenbjekte und des Applet um einiges komplexer geworden. Andererseits ist ein völliger Verzicht auf den proprietären *Smart Agent* aus den Gründen, die in Abschnitt 6.3.1 genannt wurden, mit dieser Version des ORB nicht möglich. Die Interoperabilität zwischen unterschiedlichen ORBs ist für die Testumgebung auch kein Argument. Zuletzt kann ange-

[6] Während *Netscape* die Unterstützung des JDK 1.1 für die Version 5.0 ihrer Browser zugesichert hat, ist es derzeit unklar, ob *Microsoft* überhaupt zukünftig Java in vollem Umfang unterstützen will.

```
sysagent.UNIXSystem mySystem;

// Bind to the UNIXSystem Server object
org.omg.CORBA.BindOptions bind_options = new org.omg.CORBA.BindOptions();
try {
  mySystem = sysagent.UnixSystemHelper.bind(orb, "UNIXSystem",
                                  myClient.serverHost, bind_options);
}
catch (org.omg.CORBA.SystemException se) {
  new StatusDialog(myClient.browserFrame, "Exception: " + se);
}
```

Abbildung 6.18: Lokalisieren eines Agentenobjekts

merkt werden, daß der Ort der Agentenobjekte, also der Hostname, gewöhnlich dem Client bekannt ist, da gerade Ressourcen eines bestimmten Systems überwacht werden. Erst Anfragen wie „Welche Rechner sind NFS-Server in diesem Netz?" erfordern einen mächtigeren Lokalisierungsmechanismus auf Basis des Naming-Service. In einer möglichen zukünftigen Erweiterung des Prototypen lassen sich aber die Aufrufe von `bind()` relativ einfach durch Anfragen an den Naming-Service ersetzen.

6.4.2 Aufbau des Applet

Das Management-Applet baut auf dem *CardLayout* auf, welches das AWT zur Verfügung stellt. *CardLayout* erlaubt, die graphische Oberfläche der Anwendung in mehrere Bildschirmseiten (*panels*) zu unterteilen, die unabhängig voneinander im Web-Browser dargestellt werden können. Bildlich kann man sich dies als Stapel von Karten vorstellen, von der jeweils nur eine sichtbar ist. Somit können die GUI-Elemente wie Ein-/Ausgabefelder, Listen und Buttons zur Darstellung und Manipulation der Managementinformation für einzelne oder eng zusammengehörige Klassen des Objektmodells auf einem Panel angeordnet werden. Im Idealfall hat der Anwender dann alle Informationen, die der Agent zu einer Ressource bereitstellt, auf einer Bildschirmseite parat. Das Haupt-Panel erscheint beim Start des Applet und fordert die Auswahl des zu überwachenden Systems. Nach erfolgreicher Bindung des Applet an die Objekte und Factories des Agenten, kann über entsprechende Buttons auf Unter-Panels, z.B. für allgemeine Systeminformationen, Benutzer, Dateisysteme, Überwachung von Servern, Ereignismeldungen, etc. gewechselt werden. Das *CardLayout* hat den großen Vorteil, daß die Anwendung leicht erweitert bzw. modifiziert werden kann. Es ist problemlos möglich, neue Panels in das Applet zu integrieren, wenn das Objektmodell um neue Klassen ergänzt wird. Änderungen an der von einem Agentenobjekt bereitgestellten Information oder Funktionalität ziehen wiederum meist nur Änderungen an *einem* Unter-Panel des Management-Applet nach sich. Die leichte Erweiterbarkeit des Prototypen sowohl auf Agenten- als auch auf Managerseite ist eine wichtige Anforderung, da dieser erst einen Teil der Klassen des Objektmodells aus Kapitel 4 realisiert. Auch die Verfeinerung von Objektklassen auf Agentenseite wird von dem Design der Anwendung gut unterstützt. Da die Panels eigene Java-Klassen sind, können Panels für spezialisierte Objekte die GUI-Elemente und deren Implementierung von Panels für allgemeinere Oberklassen erben.

Innerhalb eines Panels wird das äußerst mächtige, aber auch sehr komplexe *GridBagLay-*

out zur Anordnung der GUI-Elemente verwendet. Dieses erlaubt es, die einzelnen Elemente auf einem rechteckigen Gitter frei zu positionieren. Jedem einzelnen Gitterfeld werden dabei eigene Darstellungsattribute (*GridBagConstraints*) wie Schriftart, Ausrichtung, Farbe, Größe, etc. zugeordnet. Theoretisch ist es aufgrund des Rasters möglich, das Aussehen eines Panels vor der Implementierung genau zu planen. In der Praxis funktioniert dies aber wegen der unzähligen Attribute zu jedem Feld kaum. Dies impliziert ein sehr zeitaufwendiges „Trial-and-Error"-Verfahren durch Iterieren folgender Schritte: Modifikation von Attributen im Quelltext, erneute Übersetzung und Kontrolle der Bildschirmdarstellung. Die einfacheren Layout-Stile des AWT haben sich im Test jedoch als zu wenig mächtig erwiesen. Die Komplexität des *GridBagLayout* erfordert aber eigentlich den Einsatz eines Entwicklungswerkzeugs (*Java GUI-Builder*). Dieses stand während der Implementierung des Prototypen leider nicht zur Verfügung. Die Abbildungen 6.19 und 6.20 zeigen zwei Panels des Management-Applet für die Administration von Benutzerkennungen und zur Überwachung eines NFS-Servers.

Die Implementierung der Ereignisbehandlung zu den GUI-Elementen, z.B. die Reaktion auf einen gedrückten Button, kann leider nicht innerhalb der Panel-Klasse gekapselt werden. Das schwache Ereignismodell des JDK 1.0.x erfordert eine globale Behandlungsroutine in

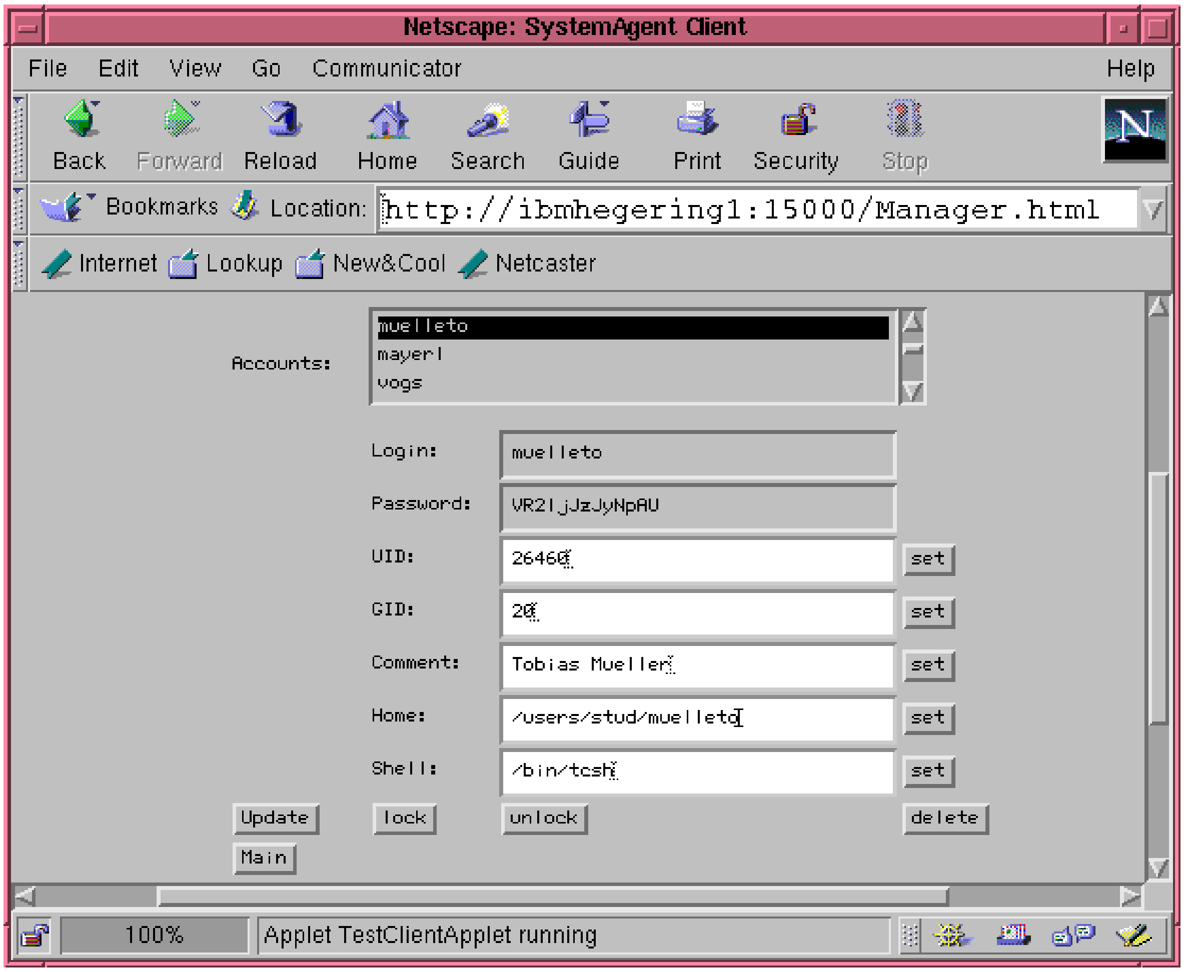


Abbildung 6.19: Panel des Management-Applet für die Benutzerverwaltung

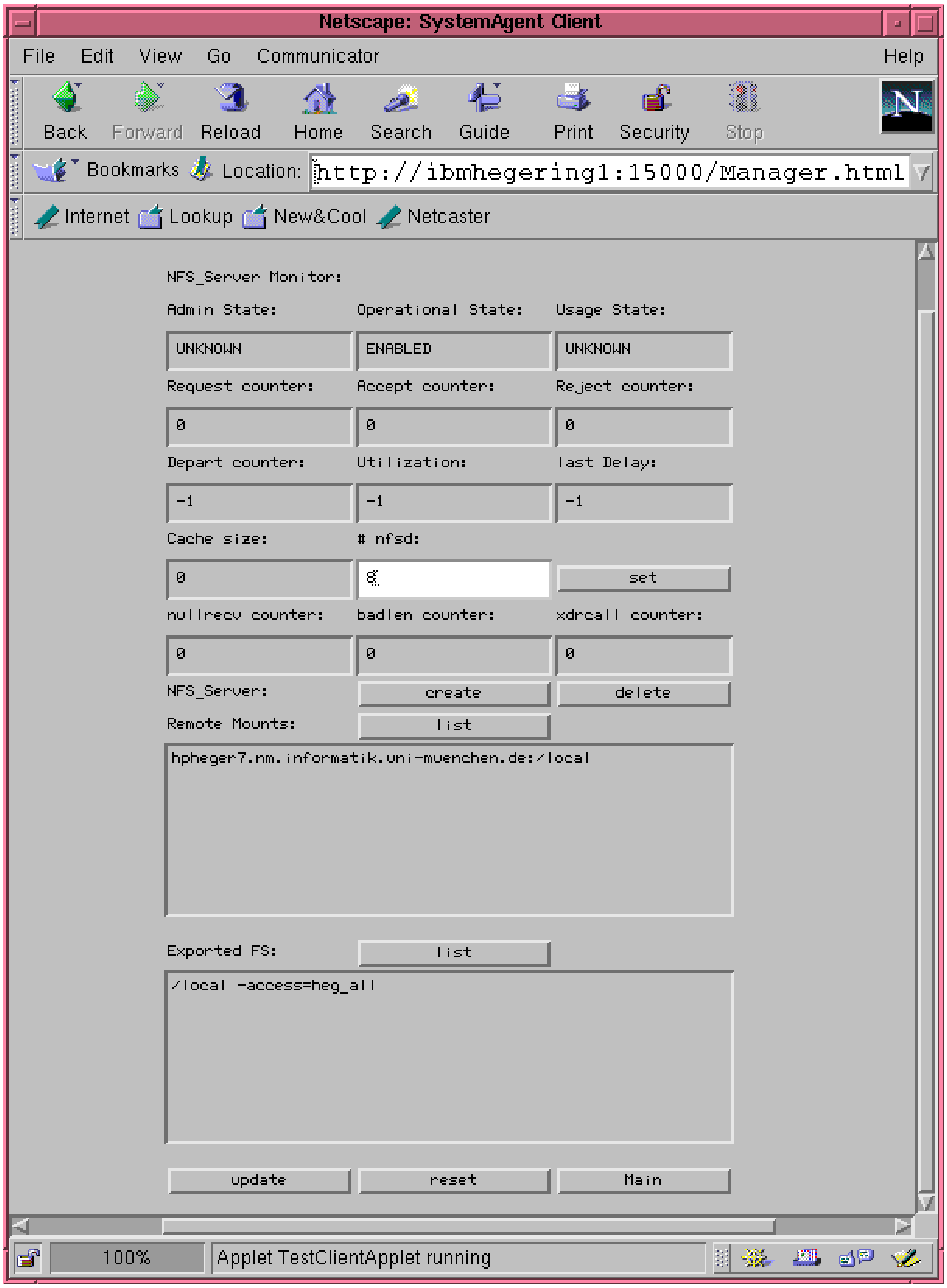


Abbildung 6.20: Panel des Management-Applet zur Überwachung eines NFS-Servers

Form einer großen, unübersichtlichen CASE-Anweisung. Sobald die Browser mehrheitlich das JDK 1.1.x unterstützen, sollte zur besseren Wartung des Applet-Codes das neue Ereignismodell implementiert werden.

6.5 Beschreibung der Testumgebung für den Prototypen

In diesem Abschnitt wird die Laufzeitumgebung des Prototyen, d.h. des CORBA-Agenten und des Management-Applet, beschrieben. Die Testumgebung ist in Abbildung 6.21 dargestellt. Für den CORBA-Agenten wurden einige Klassen des Objektmodells aus Kapitel 4 in Java implementiert, so daß die Tragfähigkeit der Realisierung demonstriert werden kann. Die Agentenobjekte ermitteln die Managementinformation zum Teil über AIX-spezifische UNIX-Kommandos oder durch Einbinden von C-Funktionen der AIX-Portierung eines SNMP-Agenten über das JNI. Daher ist der Agent trotz Implementierung in Java nur für IBM-Rechner unter AIX funktionsfähig. Zum Testen des Prototyps stand die Workstation ibmhegering1 unter AIX 4.2 zur Verfügung. Auf der Maschine war das IBM JDK 1.1.3 installiert.

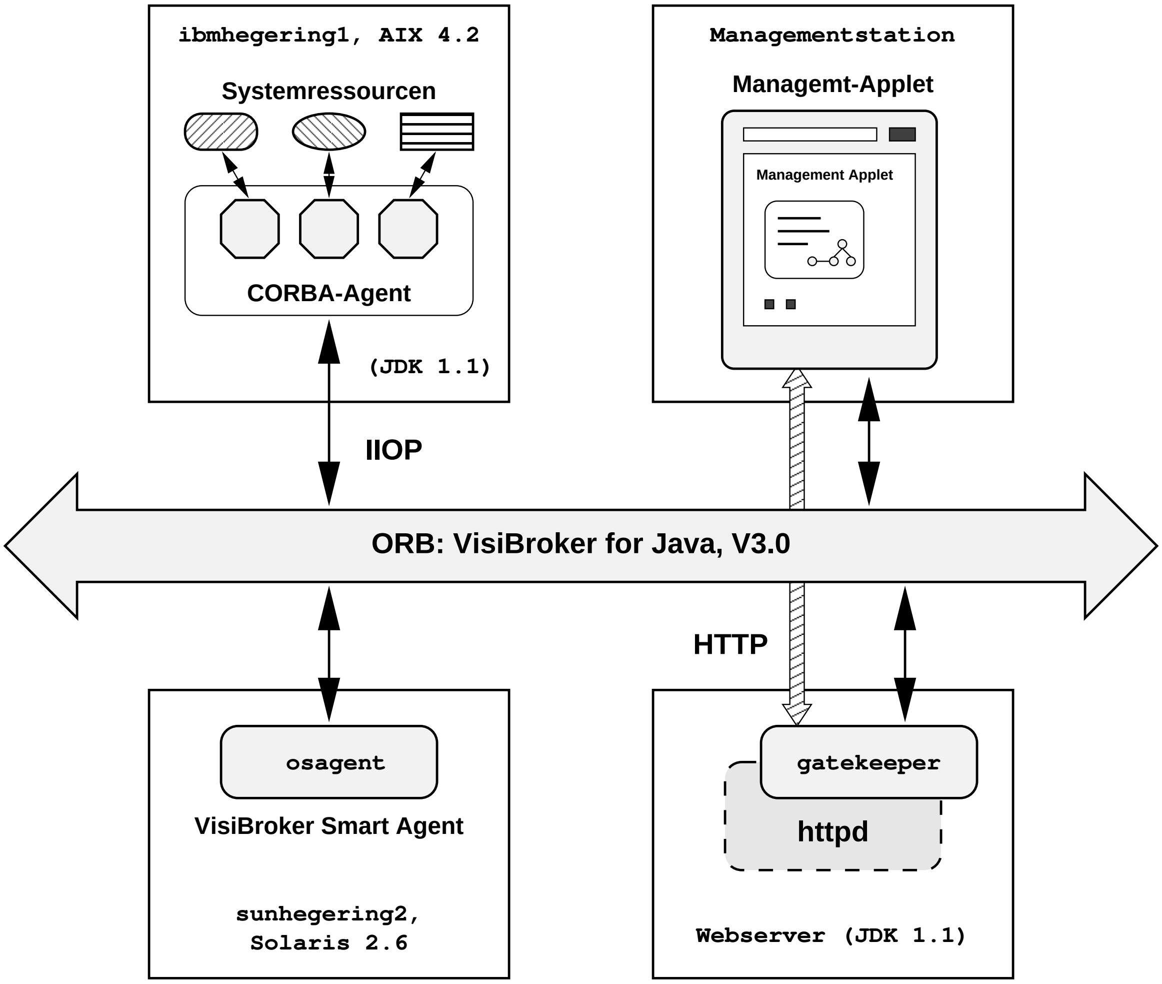


Abbildung 6.21: Testumgebung für den Prototypen

Das Server-Programm, welches die Agentenobjekte initialisiert, kann über das Kommando `vbj SystemAgentServer` gestartet werden.

Zur Lokalisierung der Agentenobjekte wird der *VisiBroker Smart Agent* verwendet (siehe Abschnitt 6.4.1). Dieser ist nur unter Solaris lauffähig. Für die Tests wurde er auf der Maschine sunhegering2 mit dem Kommando `osagent [-v] [&]` gestartet. Der Parameter «-v» veranlaßt die Ausgabe von Debug-Meldungen.

Das Management-Applet läuft innerhalb des Web-Browsers in einer abgeschotteten Umgebung (*sandbox*). Diese verbietet u.a. die Errichtung jeglicher Kommunikationsverbindungen zu Rechnern mit Ausnahme des Web-Servers, von dem das Applet geladen wurde. Der *Visigenic Gatekeeper* ermöglicht es, diese Beschränkung für die Kommunikation über CORBA zu umgehen. Wenn das Applet versucht, ein entferntes Objekt zu binden, tritt innerhalb der Java-VM eine Ausnahme (*security exception*) auf. Der Teil des ORBs innerhalb des Applet fängt diese Ausnahme auf und leitet die Anforderung an den *Gatekeeper* weiter. Dieser *muß* auf der Maschine gestartet sein, die auch den Web-Server enthält, damit diese Kommunikation erlaubt ist. Der *Gatekeeper* lokalisiert über den *Smart Agent* das betreffende Agentenobjekt und richtet bei sich ein entsprechendes Proxy-Objekt ein. Das Applet operiert von nun an auf dem Proxy-Objekt. Die Anfragen werden vom *Gatekeeper* jedoch an das eigentliche Agentenobjekt weitergeleitet. Der *Gatekeeper* muß sich im selben Subnetz wie der *Smart Agent* befinden. Außerdem besitzt er zwei weitere Funktionen. Für den Fall, daß sich zwischen Web-Browser mit dem Management-Applet und Web-Server ein Firewall befindet, der nur HTTP-Requests passieren läßt, unterstützt der *Gatekeeper* IIOP-Tunneling über HTTP. Darüber hinaus bietet der *Gatekeeper* auf dem Default-Port 15000 die Funktionalität eines Web-Servers. Da er aber komplett in Java implementiert ist, fällt die Leistung im Vergleich zu normalen Web-Servern schwach aus. Voraussetzung für den Betrieb des *Gatekeepers* ist das JDK 1.1.x und die Zugriffsmöglichkeit auf die VisiBroker-Klassen. Soll der *Gatekeeper* auch die Aufgabe des Web-Servers übernehmen, sollte er im gleichen Verzeichnis gestartet werden, das die Klassen des Applet enthält (CODEBASE). Er wird über das Kommando `gatekeeper [-ORBdebug 1]` aufgerufen. Auf allen drei bisher beschriebenen Rechnern müssen einige Umgebungsvariablen gesetzt sein (siehe Abschnitt 6.3.1 und Shellskript in Anhang B). Ebenfalls im Anhang B findet sich die HTML-Datei Manager.html, die der Rahmen für das Management-Applet ist. Die dort gesetzten Parameter sind zwingend für die Zusammenarbeit des Applet mit dem *Gatekeeper* erforderlich.

Das Management-Applet kann in jedem Java-fähigen (JDK 1.0.2) Web-Browser auf einem beliebigen Rechner aufgerufen werden. Der URL bei Benutzung des *Gatekeepers* als Web-Server lautet `http://<gatekeeper-host>:15000/Management.html`. Tabelle 6.2 enthält die Web-Browser, in denen das Applet erfolgreich getestet wurde. Hierbei wurde die Zusammenarbeit des Applet sowohl mit dem *Gatekeeper* als Web-Server als auch mit der Kombination Standard-Web-Server (httpd von IBM) und *Gatekeeper* überprüft.

Das Management-Applet verursachte zunächst zahlreiche Probleme (Sicherheitsverletzungen, Java-Runtime-Exceptions, etc.) innerhalb des *Netscape Communicator*, die im Appletviewer des JDK nicht auftraten. An dieser Stelle half auch die ansonsten recht gute Dokumentation zum VisiBroker nicht weiter. Zahlreiche Tests[7] führten schließlich zur Lösung des Problems.

[7] Hilfe boten oft Diskussionen in der Newsgroup comp.object.corba sowie eine FAQ auf dem Web-Server von Visigenic (`http://www.visigenic.com`).

Produktname	Version	Betriebssystem
Netscape Communicator	4.03[en]	Solaris 2.6
Netscape Communicator	4.03[en]	HP-UX 10
Netscape Communicator	4.03[en]	AIX 4.2
Netscape Communicator	4.03[en]	Windows 95
Internet Explorer	4.0	Windows 95
Appletviewer	JDK 1.1.x	alle

Tabelle 6.2: Getestete Web-Browser

Zum ersten sind die Parameter in der oben erwähnten HTML-Datei Manager.html äußerst wichtig. Zweitens darf die Umgebungsvariable CLASSPATH auf der Managementstation die VisiBroker-Klassen *nicht* enthalten, da der Browser sonst versucht, die ORB-Klassen lokal von der Festplatte zu laden, was wiederum zum Abbruch des Applet (*security exception*) führt. Drittens muß der Konfigurationsdatei des *Communicators* ($(HOME)/.netscape/preferences.js) folgender Eintrag hinzugefügt werden:

```
user_pref("security.lower_java_network_security_by_trusting_proxies", true);
```

Prinzipiell kann der ORB von *Visigenic*, den *Netscape* in seine Browser ab der Version 4.0 integriert hat, vom Management-Applet genutzt werden. Hierzu sind einige kleine Änderungen an der HTML-Datei nötig. Außerdem muß der *Gatekeeper* in einem speziellen Kompatibilitätsmodus gestartet werden (siehe [Vis97a]). In diesem Fall müssen die ORB-Klassen, die als Jar-Archiv immerhin eine Größe von über zwei Megabyte besitzen, nicht vom Web-Server über das Netz geladen werden. Das Problem hierbei ist, daß der ORB im *Communicator* in der „alten" Version 2.5 vorliegt. Der CORBA Event-Service von *Visigenic* funktioniert aber erst ab Version 3.0 des ORB. Weiterhin plante *Netscape* sowohl den ORB als auch die Technologie des *Gatekeeper* in ihre Web-Server zu integrieren. Es ist zu hoffen, daß die Produkte in Zukunft besser aufeinander abgestimmt werden. Die Integrationen könnten in diesem Fall die Ausführungsgeschwindigkeit des Management-Applet erheblich verbessern.

6.6 Erfahrungsbericht zur Implementierung

Zuletzt soll stichpunktartig über einige Erfahrungen berichtet werden, die bei der Implementierung des Prototypen gewonnen wurden.

- Generierung von IDL-Dateien aus StP
  - Um den etwas umständlichen Anmerkungsmechanismus zur Definition von eigenen IDL-Datentypen für Attribute einer Objektklasse in StP zu umgehen, kann folgendes Vorgehen gewählt werden: Alle Datentypen werden außerhalb von StP in einer globalen Datei types.idl definiert. Den von StP generierten IDL-Dateien wird ein entsprechendes Include-Statement hinzugefügt.
  - StP sollte Objektklassen innerhalb sog. OMT-Subsysteme auf IDL-Module abbilden. Dies würde zu einer besseren Gliederung der Klassen führen, da der IDL-Compiler die Objekte eines IDL-Moduls in einem Java-Package zusammenfaßt.

Diese Abbildung funktioniert in der verwendeten Version von StP/OMT laut Aonix-Support nicht. Hier war ebenfalls ein manueller Workaround erforderlich.

- Implementierung des CORBA-Agenten
  - Aufgrund der Spezifika der Systeme ist eine Vererbung von Code und somit die Wiederverwendung für die Klassen des Objektmodells nur sehr eingeschränkt möglich. Im wesentlichen müssen die Klassen in den Blättern der Vererbungshierarchie jeweils neu implementiert werden, da die Ermittlung der gleichen Information für unterschiedliche Systeme auch unterschiedlich realisiert werden muß. Diese Probleme, die bereits von der Implementierung der SNMP-Subagenten für diverse UNIX-Derivate bekannt sind, übertragen sich somit auch auf die Klassen für den CORBA-Agenten. Keinesfalls darf aufgrund der plattformunabhängigen Programmiersprache Java auf die Plattformunabhängigkeit des Agenten geschlossen werden.
  - Die Wiederverwendung von C-Funktionen zur Ermittlung von Managementinformation durch das JNI funktioniert prinzipiell gut. Die Implementierung der Wrapper ist weit weniger aufwendig als eine Neuimplementierung der übernommenen Funktionalität. Als problematisch hat sich jedoch herausgestellt, daß die Modularität der C-Funktionen des bestehenden SNMP-Agenten nicht konsequent durchgehalten war. Die Verwendung der Funktionen über das JNI erfordert dann aber die Einbindung aller abhängigen Objekt-Files in die Bibliothek.
  - Relativ gut eignet sich die Methode, Managementinformation durch Ausführen von UNIX-Kommandos aus dem Agenten heraus zu ermitteln. Gerade AIX stellt eine Vielzahl von Administrationskommandos bereit. Dies vermindert auch den Wartungsaufwand des Agenten bei neuen Versionen des Betriebssystems, da die Systemkommandos meistens zumindest abwärtskompatibel sind.
  - Das Polling von Attributen oder Zuständen einer Ressource kann in Java sehr einfach über Threads innerhalb eines Agentenobjekts realisiert werden. Hierzu muß lediglich die Methode `run()` für das Java-Interface *Runnable* implementiert werden.
  - Das Push-Modell des CORBA Event-Services bietet eine sehr gute Basis für asynchrone Meldungen eines Agenten. Da sich Ereigniskanäle leicht hierarchisch anordnen lassen, sind CORBA-Objekte denkbar, die Funktionen zur Ereignisfilterung und -korrelation bereitstellen.
  - Die Code-Größe der Java-Objektklassen ist sehr gering. Sie liegt durchwegs unter 10 kB pro Klasse. Die Klassen des ORBs einschließlich der Services besitzen eine Gesamtgröße von ca. 2.5 MB.
  - Aussagen über die Performance des Agenten sind nur schwer zu treffen. Die Antwortzeit bei Abfragen über das Management-Applet war subjektiv als gut einzuschätzen. Genaue Messungen müßten zeigen, wieviel Zeit die Ausführung einer Anfrage durch den Agenten benötigt und wieviel Zeit auf den Kommunikationsmechanismus entfällt. Auch bei der Leistung der Java-VM sind in Zukunft noch erhebliche Verbesserungen zu erwarten.

  - Auch die Systembelastung durch den Agenten ist aus zwei Gründen schwer zu beurteilen. Einerseits konnte für den Prototypen nur ein Teil der Klassen des Objektmodells implementiert werden. Andererseits handelte es sich bei der Testmaschine für den Agenten um eine relativ leistungsfähige, zugleich aber wenig belastete Workstation.

- Implementierung des Management-Applet

  - Java-Applets stellen eine gute Möglichkeit dar, mit relativ geringen Aufwand eine plattformunabhängige Managementanwendung mit graphischer Benutzeroberfläche zu realisieren. Ein Manager kann somit praktisch von jedem Rechner im Intra- bzw. Internet auf die Agenten zugreifen. Sobald die gängigen Browser das JDK 1.1 unterstützen, können auch die erweiterten GUI-Funktionen des AWT und das neue Ereignismodell genutzt werden.

  - Für den Entwurf der Oberfläche sollte dringend ein leistungsfähiger GUI-Builder eingesetzt werden, um Zeit bei der Implementierung des Applet zu sparen. Ohne GUI-Builder entfiel ca. 50% des Zeitbedarfs pro implementierter Klasse des Modells auf das Applet.

  - Wie im letzten Abschnitt beschrieben, traten einige Probleme in Form von *Security Exceptions* innerhalb des Netscape-Browsers auf. Diese sind darauf zurückzuführen, daß das Produkt *Visibroker for Java* noch sehr jung ist. Auch der Versionskonflikt zwischen dem im *Communicator* integrierten ORB und dem der Entwicklungsumgebung trug hierzu bei.

Kapitel 7

Zusammenfassung und Ausblick

Der Trend zu verteilten, heterogenen Systemen führt nicht nur zu einer steigenden Komplexität des Managements insgesamt, sondern auch dazu, daß die Grenzen zwischen Netz-, System- und Anwendungsmanagement immer mehr verschwimmen. Ein funktionierendes Kommunikationsnetz ist Voraussetzung für eine verteilte Umgebung, in der Systemdienste zu verteilten Anwendungen werden. Neben dem Management von Endsystemressourcen nimmt das Dienstmanagement einen wachsenden Stellenwert ein. In Kapitel 2 wurde festgestellt, daß für ein integriertes Management von UNIX-Workstations ein mächtiges Managementmodell erforderlich ist, welches einerseits von Spezifika der Ressourcen zur Verschattung der Heterogenität abstrahiert und generische Information und Funktionalität definiert, die ein effizientes auf die Anforderungen des Betreibers ausgerichtetes Management ermöglichen. Andererseits sollte das Modell aus Gründen der Zukunftssicherheit und der breiten Anwendbarkeit unabhängig von einer bestimmten Managementarchitektur sein.

Ein solches Modell ist im Rahmen dieser Arbeit durch Zusammenführen und Weiterentwicklung vorhandener Ansätze in mehreren Schritten erstellt worden. Bei der anschließenden prototypischen Implementierung entstand eine Managementlösung in Form eines CORBA-Agenten und zugehöriger Managementanwendung mit graphischer Oberfläche.

Für das Informationsmodell wurde als Basis die *Object Modeling Technique* festgesetzt. Neben dem mächtigen objektorientierten Ansatz bietet OMT im Objektmodell die graphische Repräsentation von Beziehungen zwischen den MOCs. Außerdem wurde gezeigt, daß das dynamische Modell nicht nur die Modellierung von asynchronen Meldungen für Ereignisse erlaubt, die innerhalb der Ressourcen auftreten können, sondern im Zustandsdiagramm auch die Zustandsübergänge graphisch darstellen kann, die zum Eintritt eines Ereignisses führen. Da die verwendeten OMT-Techniken praktisch unverändert von UML, dem kommenden State-of-the-Art für Design und Entwurf objektorientierter Software, übernommen werden, ist die Zukunftssicherheit gewährleistet. Im Hinblick auf die Realisierung des Modells auf Basis von CORBA war die Möglichkeit der Generierung von IDL-Objektbeschreibungen durch ein kommerzielles CASE-Tool ein weiteres wichtiges Argument für OMT.

Zur Gewinnung von generischen Basisklassen für das Modell wurde im ersten Schritt der Ansatz von Bernhard Neumair aufgegriffen und auf das Systemmanagement ausgedehnt. Dieser analysiert unter dem Managementgesichtspunkt die semantischen Konzepte der RM-ODP

Viewpoint Languages, um daraus geeignete generische MOCs abzuleiten. Das Referenzmodell bietet eine solide, weil standardisierte, Grundlage und gewährleistet aufgrund seiner abstrakten, systemunabhängigen Sichten die breite Anwendbarkeit der gefundenen MOCs auf Ressourcen eines heterogenen, verteilten Systems. Der dienstorientierte Blick des *Computational Viewpoint*, der die funktionale Zerlegung einer Anwendung in Software-Komponenten beschreibt, die über Schnittstellen interagieren, führt zu MOCs für das Software- und Anwendungsmanagement. Der *Engineering Viewpoint* hingegen legt fest, welche Infrastrukturobjekte eine verteilte Plattform bieten muß, um die Kooperation der verteilten Anwendungskomponenten zu unterstützen. Aus diesen Konzepten lassen sich MOCs für Endsystem- und Netzressourcen wie Prozesse und Kommunikationsverbindungen ableiten.

Um ein effizientes Management zu ermöglichen, erfolgte im zweiten Schritt die Definition von Managementinformation und Funktionalität in Form von Attributen und Methoden für die gefundenen Basisklassen. Hierzu wurde auf die Anforderungen verschiedener Managementfunktionsbereiche, auf existierende generische MIBs und auf bottom-up gewonnene Information spezieller Ressourcen zurückgegriffen. Für letztere mußte insbesondere überprüft werden, ob sie so allgemeingültig ist, daß sie generell auf Klassen dieser Ressourcen anwendbar ist und auch gewöhnlicherweise bereitgestellt werden kann. Das Gesamtmodell der Basisklassen ist in den Abbildungen A.1 und A.2 im Anhang A zu finden.

Im nächsten Schritt wurde das Modell um Klassen für das Management der Endsystemressourcen von UNIX-Workstations erweitert, in dem der Prototyp eines vorhandenen Objektmodells erfolgreich über eine Spezialisierungshierarchie integriert wurde. Da dies praktisch problemlos gelang, sollten sich die generischen Basisklassen auch auf andere Endsysteme, z.B. unter OS/2 oder Windows NT, anwenden lassen.

Ein weiterer Schwerpunkt der Arbeit war die Berücksichtigung des Managements verteilter Systemdienste. Hierzu wurden generische MOCs für beliebige Client/Server-basierte Dienste eingeführt und wiederum mit Information und Funktionalität zu verschiedenen Funktionsbereichen ausgestattet. Diese wurden anschließend für die konkreten Dienste NFS und NIS weiter spezialisiert. Eine Bottomp-Up-Analyse hat ergeben, daß die geforderte Managementinformation ohne Instrumentierung der Software momentan leider nur zum Teil von den Diensten bereitgestellt wird. Dies gilt insbesondere auch für wünschenswerte asynchrone Ereignismeldungen. Trotzdem erlaubt das entstandene Modell (siehe Abbildung A.3) die Entwicklung von Managementanwendungen, die auf unterschiedlichen Ebenen der Spezialisierungshierarchie operieren. Auf der obersten Ebene der ODP-Basisklassen ist die Überwachung des Status beliebiger Anwendungen möglich. Eine Applikation für das Dienstmanagement könnte auf den generischen MOCs **Server**, **Client** und **Request** arbeiten. Schließlich wird ein Administrationswerkzeug für NFS auf unterster Ebene die dienstspezifischen Klassen instantiieren. Letzteres wurde im Rahmen des implementierten Prototyps realisiert.

Zur Erstellung des Objektmodells und der Zustandsdiagramme wurde das CASE-Tool StP/OMT eingesetzt. Nach Festlegung der Datentypen für die Attribute und der Parameter und Rückgabewerte für die Methoden generiert das Werkzeug aus dem Objektmodell automatisch die IDL-Beschreibungen der Managementobjektklassen als Grundlage für den CORBA-Agenten. Der generierte Code konnte ohne größere Modifikationen vom IDL-Compiler übersetzt werden. Bei der Abbildung gehen allerdings Assoziationen und Aggregationsbeziehungen zwischen den Klassen verloren. Auch die in den Zustandsdiagrammen definierten asynchronen Ereignismeldungen werden bei der Code-Generierung übergangen. Schwächen zeigte das ansonsten sehr leistungsfähige Werkzeug darüber hinaus bei der Formatierung und beim Ausdruck der Modelle.

Die CORBA-Entwicklungs- und Laufzeitumgebung *VisiBroker for Java* war insgesamt betrachtet sehr gut für die Realisierung des Prototyps geeignet. Die Java-Agentenobjekte sind allerdings aufgrund der Heterogenität der zu managenden Endsysteme nicht plattformunabhänging. Das JNI ermöglicht sowohl die Nutzung von Betriebssystemfunktionen innerhalb eines Agentenobjekts als auch die Wiederverwendung von C-Code eines bestehenden SNMP-Agenten unter der Voraussetzung, daß die nativen Funktionen in einzelnen Modulen vorliegen. Dynamische Managementobjekte des Agenten wurden mit Hilfe von Factories bereitgestellt. Asynchrone Ereignismeldungen wurden auf Basis des CORBA Event-Service realisiert. Der enstandene CORBA-Agent implementiert einen repräsentativen Teil des entwickelten Objektmodells für das Betriebssystem AIX 4.2. Zum Zugriff auf die Information und Funktionalität des Agenten wurde eine plattformunabhängige Managementanwendung in Form eines Java-Applet erstellt, die innerhalb aller gängigen Web-Browser ausgeführt werden kann. Der ohnehin schon relativ kleine Aufwand für die graphische Oberfläche der Anwendung könnte durch den Einsatz eines GUI-Builder noch erheblich gesenkt werden. Die Kommunikation zwischen Applet und Agentenobjekten über CORBA funktioniert bis auf kleinere Probleme, die auf den mangelnden Reifegrad der verwendeten Produkte zurückzuführen sind, reibungslos. Die Ergebnisse der Arbeit zeigen, daß der CORBA-basierte Ansatz für das System- und Anwendungsmanagement tragfähig ist. Voraussetzungen für einen Durchbruch sind allerdings die Integration eines ORB in die unterschiedlichen Betriebssysteme der Workstations und die weitgehende Verwendung von CORBA durch die Software-Hersteller bei der Entwicklung verteilter Anwendungen.

Zuletzt soll noch ein Ausblick auf offene Fragestellungen präsentiert werden, die sich aus dieser Arbeit ergeben. Ebenso sollen Punkte genannt werden, an denen aufbauende Arbeiten anknüpfen könnten.

Eine zentrale Fragestellung ist natürlich, ob die Information und Funktionalität der generischen Basisklassen des Objektmodells tatsächlich allen Anforderungen des System- und Anwendungsmanagements gerecht wird. Aufgrund der Breite dieser beiden Disziplinen konnten einige Bereiche nur oberflächlich betrachtet werden. Ebenso steht der Beweis aus, daß die Information der Oberklassen auch für das Management von Workstations, die nicht unter UNIX betrieben werden, angewendet werden kann. Hierzu wäre eine Bottom-Up-Analyse der Betriebssysteme OS/2 und Windows NT einschließlich der Betrachtung der Systemdienste erforderlich.

Weiterhin fehlt es an ODP-konformen Anwendungen, die Managementinformation an einer genormten Schnittstelle bereitstellen. Bei kommerziellen Werkzeugen zum Anwendungsmanagement handelt es sich praktisch ausschließlich um spezielle Tools, die mit einem Software-Produkt ausgeliefert werden oder in dieses integriert sind. Ein Beispiel hierfür ist das *Computing Center Management System* zur Status- und Performance-Überwachung von SAP R/3. Hier wären die Software-Hersteller gefordert, ihre Anwendungen entsprechend zu instrumentieren, damit Agenten entsprechende Klassen des Objektmodells wie **compObject**, **comp Interface** und **interactionInfo** instantiieren können und die Information somit einer integrierten Managementlösung zugänglich machen können. Zumindest bei der Neuentwicklung von Systemdiensten und Anwendungen wäre es wünschenswert, daß diese auf dem RM-ODP basierten, damit RM-ODP-konforme Beschreibungen vorliegen würden, um daraus spezifische

MOCs ableiten zu können, die von einem Agenten implementiert werden könnten.

Das Objektmodell bietet Spielraum für Erweiterungen und Weiterentwicklungen:

- Neben der bereits angesprochenen Spezialisierung der Basisklassen für andere Systemplattformen und Dienste können dem Modell MOCs hinzugefügt werden, die Agenten bzw. Applikationen generische, wiederverwendbare Funktionalität anbieten. Beispiele hierfür wären Klassen für Logging, Schwellwertüberwachung, statistische Auswertung von Meßdaten und Filterung von Ereignissen. Als Grundlage könnten hierfür die *Systems Management Functions* der ISO dienen. Diese Klassen könnten auf einer CORBA-Managementplattform durch verteilte Unterstützungsobjekte realisiert werden.

- Interessant wäre auch der Aufbau einer Generalisierungshierarchie für Ereignisse für die Managementobjektklassen. OMT unterstützt eine solche Hierarchie innerhalb des dynamischen Modells. Hierzu müßte untersucht werden, ob den Basisklassen generische Ereignisse und entsprechende Meldungen zugeordnet werden können, die für Unterklassen entsprechend verfeinert werden. Beispielsweise könnte das Computational Object eines Servers eine «*Security Violation*» erzeugen, wenn ein Unberechtigter den Dienst benutzen möchte. Konkrete Dienste könnten den „Angriffsversuch“ dann genauer spezifizieren.

- Eine Analyse der RM-ODP *Viewpoint Languages* des *Enterprise* und *Information Viewpoint* könnte die Grundlage für zusätzliche generische Managementinformation oder Klassen für das Management verteilter, heterogener Systeme sein. Der *Enterprise Viewpoint* könnte unternehmensspezifische Vorgaben definieren, dessen Einhaltung das Management überwachen muß.

Ein weites Feld für weiterführende Arbeiten stellt auch der entstandene Prototyp dar. Die folgende Aufzählung nennt die entsprechenden Ansatzpunkte.

- Die erste Aufgabe besteht darin, den Agenten generell zu erweitern und die noch fehlenden Klassen des Modells zu implementieren. Dies zieht natürlich auch den Ausbau des Management-Applet nach sich.

- Der nächste Schritt wäre die Portierung des Agenten für andere UNIX-Derivate wie z.B. Solaris und HP-UX oder – soweit möglich – auf OS/2 und Windows NT.

- Der Mechanismus zur Lokalisierung der CORBA-Agentenobjekte sollte von der proprietären „bind“-Methode auf den standardisierten Naming-Service umgestellt werden. Hierzu ist es erforderlich, eine Namenshierarchie für die Objekte einzuführen. Bei dem bisherigen Prototypen muß dem Manager der Ort (Hostname) des Agentenobjekts bekannt sein. Auf der Basis des Naming-Service könnten dann mächtigere Lokalisierungsfunktionen für die Managementanwendung implementiert werden. Ein Beispiel wäre eine Funktion, die eine Liste aller Agentenobjekte liefert, die den Status von Systemdiensten überwachen.

- Eine CORBA-Managementplattform benötigt auch gewisse Basiswerkzeuge. In Anlehnung an die SNMP-MIB-Browser könnte ein Tool implementiert werden, welches die Auswertung der Informationen über die Agentenobjekte erlaubt, die im *Interface Repository* abgelegt sind. Dies könnte zu einer generischen Anwendung führen, die mittels dieser Informationen über das *Dynamic Invokation Interface* (DII) des ORB auf beliebige Agentenobjekte zugreifen könnte.

Der Einsatz weiterer CORBA-Services würde die Leistungsfähigkeit der Managementlösung erhöhen. Die folgenden Anregungen hängen natürlich von der Verfügbarkeit des entsprechenden Service ab.

Event-Service: Der Event-Service des *VisiBroker* unterstützt derzeit nur untypisierte Events. Typisierte Events würden es einer Managementapplikation erlauben, nur bestimmte Klassen von Ereignismeldungen zu abonnieren, ohne daß dafür separate Kanäle (*event channels*) erforderlich wären. Hiermit könnte die Funktionalität des *Event Forwarding Discrimator* des OSI-Managements nachgebildet werden.

Relationship-Service: Assoziationen und Aggregationen des Objektmodells könnten auf den Relationship-Service abgebildet werden, der diese speziellen Beziehungen bereits im Standard (*Containment and Reference*) vorsieht. Im Idealfall wird diese Abbildung in einer späteren Version von StP direkt unterstützt. Ohne auf die Details einzugehen, stellt der Dienst die Relationen selbst und die Rolle, die ein Objekt in der Beziehung eingeht (z.B. *Owner*) als eigene Objekte dar. Es wird Funktionalität bereitgestellt, um ausgehend von einer Beziehung die in Relation stehenden Objekte zu ermitteln. Ebenso wird sichergestellt, daß Typrestriktionen und Kardinalitätsbedingungen eingehalten werden. Beispielsweise könnte über den Relationship-Service eine Manageranfrage wie „*Liefere alle Prozesse auf allen Systemen, die zu einer verteilten Anwendung gehören.*" realisiert werden, ohne daß diese spezielle Anfrage bereits bei der Implementierung der Agentenobjekte berücksichtigt werden muß.

Lifecycle-Service: Es wäre für die Wartung und Weiterentwicklung der Agenten wünschenswert, den Code aller Agentenobjekte (auch für unterschiedliche Architekturen und Betriebssysteme) im *Implementation Repository* auf einem zentralen Rechner abzulegen. Daher könnte versucht werden, folgendes Szenario zu realisieren: Eine Anfrage eines Managers über das Management-Applet veranlaßt den ORB, das entsprechende Server-Objekt aus dem zentralen *Implementation Repository* zu holen und über den Lifecycle-Service auf das Zielsystem zu transportieren. Dort wird das Objekt aktiviert, um die Anfrage des Clients zu bearbeiten. Der ORB liefert schließlich das Ergebnis zurück. Ein Problem hierbei ist die erforderliche architekturabhängige Benennung der Agentenobjekte.

Schließlich sollte die Tragfähigkeit der CORBA-Managementlösung auch in produktiven Systemumgebungen durch Performance-Messungen und Überprüfung der Skalierbarkeit unter Beweis gestellt werden. Dies könnte auch den Vergleich der Leistungsfähigkeit mit einer herkömmlichen, SNMP-basierten Lösung einschließen. Die CORBA-Lösung wird überzeugen, sobald die angesprochene Funktionalität der CORBA-Services zur Verfügung steht und von Agenten und Managern genutzt wird. Die prinzipielle Eignung des Ansatzes hat diese Arbeit gezeigt.

Anhang A

OMT-Objektmodelle

Objektmodell für das Management von UNIX-Workstations

Die folgenden Klappseiten im A3-Format enthalten das komplette Modell für das Management von UNIX-Workstations, das in dieser Arbeit entwickelt wurde.

- Abbildung A.1, Seite 167: Generische MOCs zum Engineering Viewpoint
- Abbildung A.2, Seite 168: Generische MOCs zum Computational Viewpoint, Software- und Dienst-Management
- Abbildung A.3, Seite 169: Spezielle MOCs für das Management der Endsystemressourcen einer UNIX-Workstation, NFS- und NIS-Management

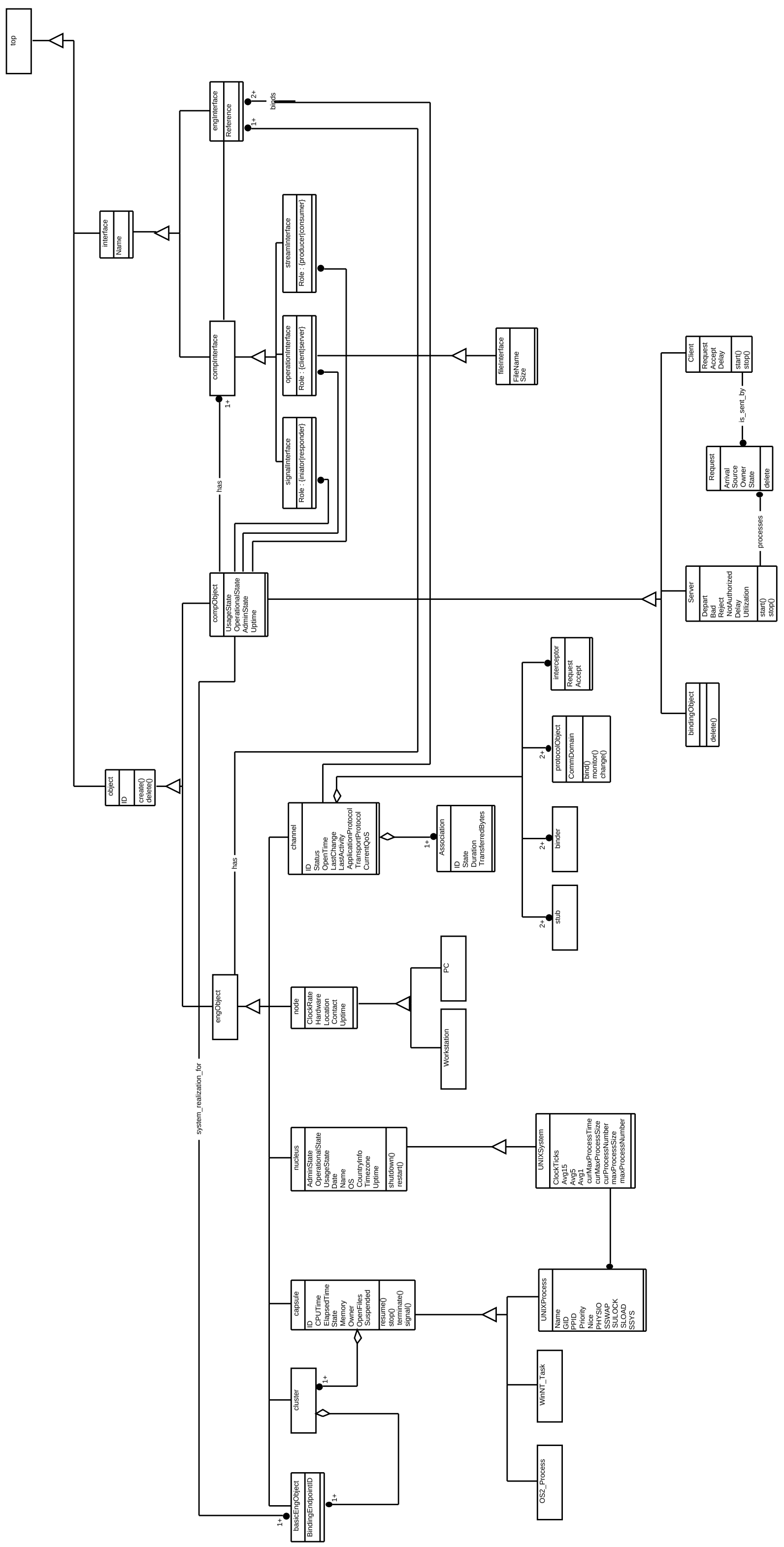


Abbildung A.1: Generische MOCs zum Engineering Viewpoint

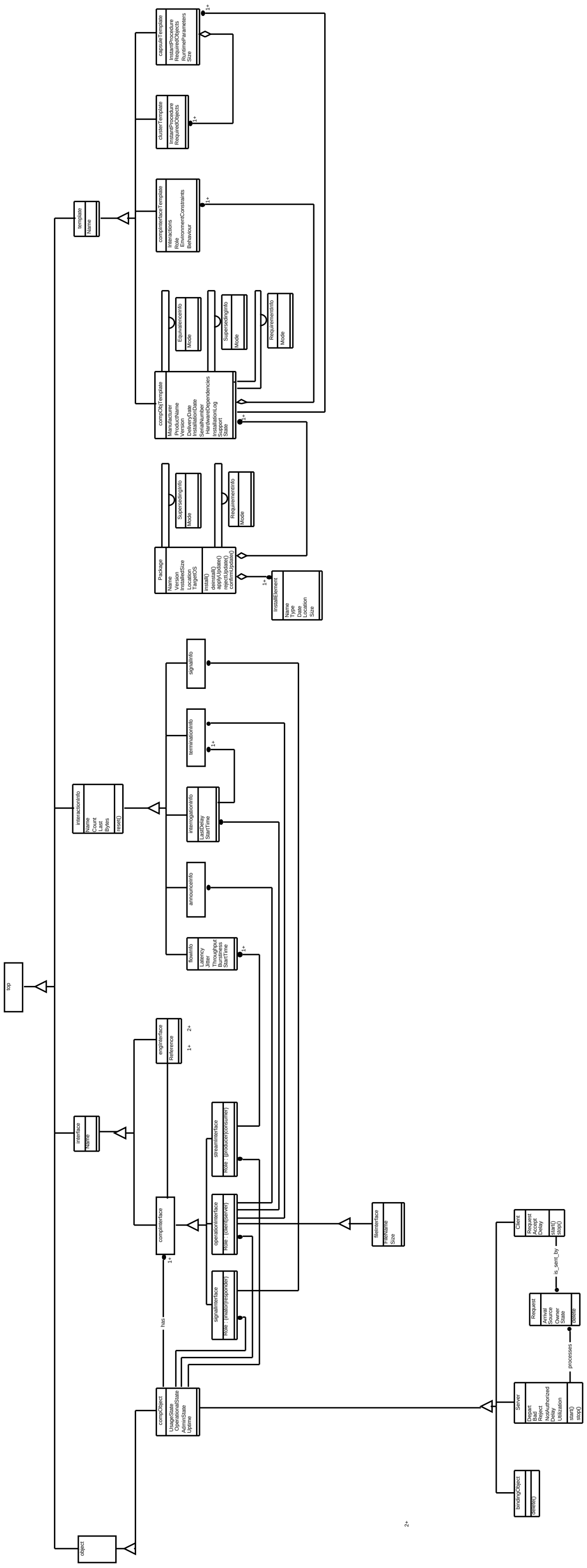


Abbildung A.2: Generische MOCs zum Computational Viewpoint

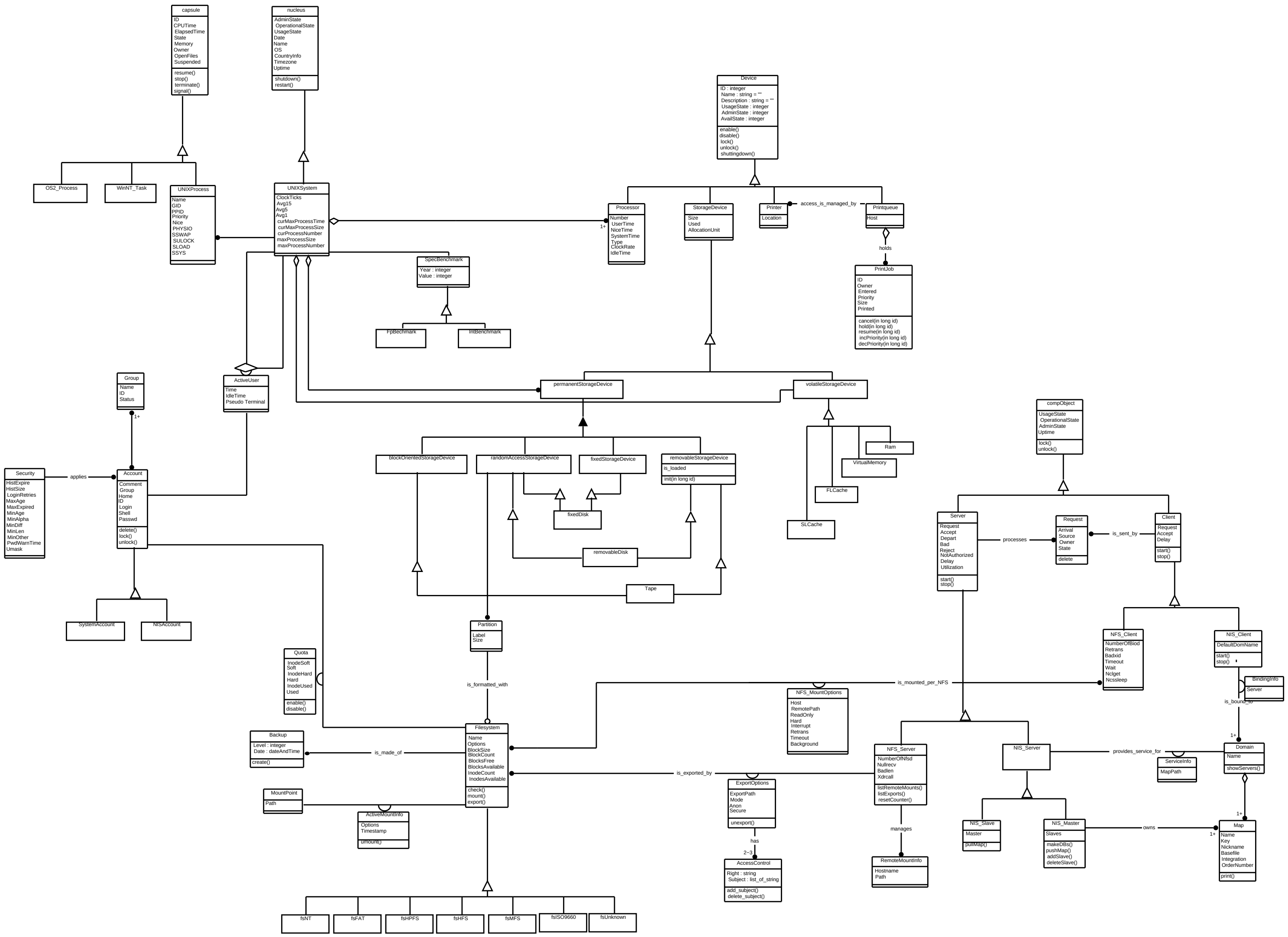


Abbildung A.3: Spezielle MOCs für UNIX-Workstations

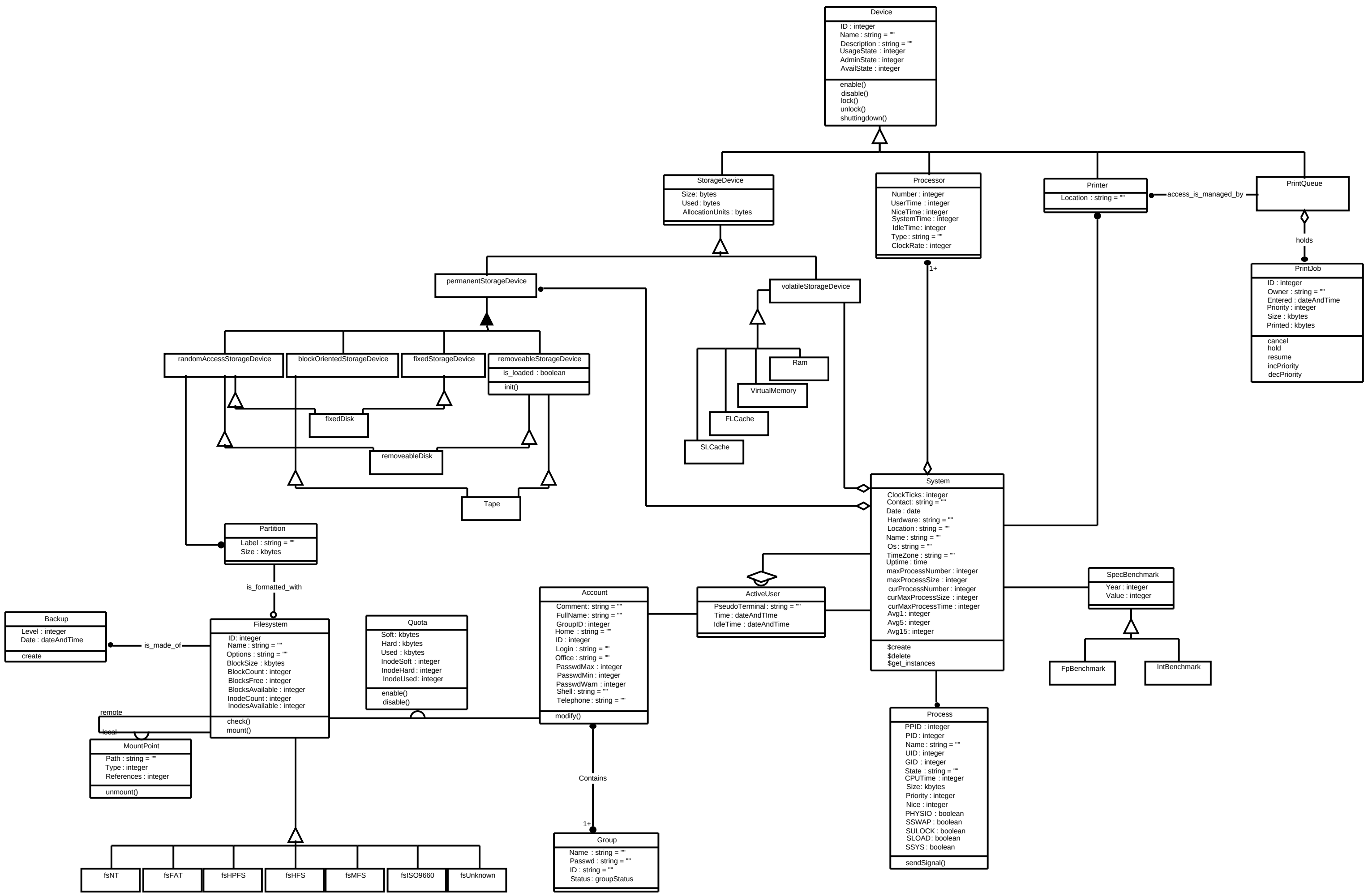


Abbildung A.4: Originalmodell aus [Sir96]

Anhang B

Implementierungsbeispiele

Shellskript zum Setzen der VisiBroker-Umgebung, Kapitel 6.3.1

```
setenv VBROKER_ADM /opt/vbroker_adm
setenv CLASSPATH /usr/local/mnmcommon/vbroker-3.0/lib:${CLASSPATH}
setenv CLASSPATH /usr/local/mnmcommon/vbroker-3.0/lib/vbj30.jar:${CLASSPATH}
setenv CLASSPATH /usr/local/mnmcommon/vbroker-3.0/lib/vbjcosev.jar:${CLASSPATH}
setenv PATH /usr/lpp/Java/bin:/usr/local/mnmcommon/vbroker-3.0/bin:${PATH}

setenv OSAGENT_ADDR 129.187.214.2

setenv LD_LIBRARY_PATH /users/stud/muelleto/proj/da/lib
```

Funktion get_sysUptime.c, Kapitel 6.3.4

```
/* $Id: get_sysUptime.c,v 1.1 1996/10/15 10:42:53 haslbecs Exp $ */

/*
** "The time since last boot in seconds."
*/

#include <unistd.h>
#include <sys/types.h>
#include <sys/times.h>

#include "get_sysUptime.h"

/* uptime des systems. */

const int *get_sysUptime( int *param_list )
{
    static int uptime;
    struct tms tbuf;
```

```
    uptime = times( &tbuf ) / sysconf( _SC_CLK_TCK );;

    return &uptime;
}
```

Shellskript zur Erstellung von shared libraries für AIX, Kapitel 6.3.4

```
#!/bin/sh
#
# Shellskript create_lib_aix:
#
# 1. Parameter: Name der C-Datei mit JNI-Funktionen
# 2. Parameter: Name der zu erstellenden Bibliothek
#
# Beispiel: create_lib_aix UNIXSystemImpl UNIXSystem
#
# Ergebnis: libUNIXSystem.AIX.so
#
# Die .o-Dateien der einzubindenden Agentenfunktionen muessen sich im
# gleichen Verzeichnis wie die zu kompilierende C-Datei befinden
#
# Vergleiche: /usr/lpp/Java/jni_example/c/Makefile
#
xlc_r -c -M -I.  -I/usr/lpp/Java/include -I/usr/lpp/Java/include/aix \
            $1.c -o $1.o
grep " JNICALL " $1.h  | sed "s/.* JNICALL //g">"lib"${2}.so.exp
ld  -bnoquiet -bnoentry -bM:SRE -blibpath:/lib:/usr/lib -lc_r \
    -blibpath:/usr/lib/threads:/usr/lib:/lib -bE:"lib"${2}.so.exp
    -L /usr/lpp/Java/lib/aix/native_threads -ljava
    -o /users/stud/muelleto/proj/da/lib/"lib"${2}.AIX.so *.o
rm "lib"${2}.so.exp *.u
```

Objektaktivierung für den Tie-Ansatz, Kapitel 6.3.5

```
[...]
// create server object NFS_Server as a TIE-Server
// create implementation object
NFS_ServerImpl ref_nfsserv= new NFS_ServerImpl();
// create proxy-object and connect it with implementation
nfs.NFS_Server server = new nfs._tie_NFS_Server(ref_nfsserv, "NFS_Server");
// activate proxy-object
boa.obj_is_ready(server);
System.out.println(server + " is ready.");
[...]
```

Klasse AccountFactoryImpl.java, Kapitel 6.3.6

```
// AccountFactoryImpl.java
// last modified 30.10.97
// Factory for normal system accounts and NIS Accounts

import java.util.*;
import java.io.*;
import mytools.*;

public class AccountFactoryImpl extends sysagent._AccountFactoryImplBase
{
  public final String passwdFile = "/etc/passwd";
  public final String ypcatCmd ="/usr/bin/ypcat ";

  private org.omg.CORBA.ORB orb;
  private org.omg.CORBA.BOA boa;

  private Dictionary accDic = new Hashtable();  // Hashtable for IORs

  public AccountFactoryImpl(String name)
  {
    super(name);

    orb = org.omg.CORBA.ORB.init();
    boa = orb.BOA_init();

    FileInputStream source = null;
    BufferedReader in;
    String line;
    parseLineColon plc;

    Runtime shell = Runtime.getRuntime();
    Process process;
    InputStream input;
    BufferedReader reader;
    String out = new String("");

    SystemAccountImpl sysaccount_ref;
    sysagent.SystemAccount account;
    NISAccountImpl nisaccount_ref;
    sysagent.NISAccount nisaccount;

    // factory for normal system accounts
    // reads from /etc/passwd
    try {
      source = new FileInputStream(new File(passwdFile));
      in = new BufferedReader(new InputStreamReader(source));

      while(true) {
        line = in.readLine();
        if (line == null) break;
```

```
      plc = new parseLineColon(line);
      if (plc.field(1).equals("+"))
        continue;
      // login:passwd:id:gid:gecos:home:shell
      System.out.println("Creating account: " + plc.field(1) + "-"
                         + plc.field(2) + "-" + plc.field(3) + "-"
                         + plc.field(4) + "-" + plc.field(5) + "-"
                         + plc.field(6) + "-" + plc.field(7));
      // create new Account server object
      // tie model
      sysaccount_ref = new SystemAccountImpl(plc.field(1), plc.field(2),
                            (new Long(plc.field(3))).longValue(),
                            (new Long(plc.field(4))).longValue(),
                            plc.field(5), plc.field(6), plc.field(7));
      account = new sysagent._tie_SystemAccount(sysaccount_ref);
      // Make the object available to the ORB.
      boa.obj_is_ready(account);
      // Save object in dictionary
      accDic.put(plc.field(1), account);
    }
  }
  catch (IOException e) {
    e.printStackTrace();
  }
  finally {
    if(source != null)
      try { source.close(); }
    catch (IOException e) { e.printStackTrace(); }
  }

  // factory for NISAccounts
  // output from ypcat passwd
  [...]

public sysagent.Account open(String name)
{
  // Lookup object in dictionary
  sysagent.Account acc = (sysagent.Account) accDic.get(name);

  // return the IOR
  return acc;
}

public sysagent.Account[] list()
{
  Enumeration enumAccounts = accDic.elements();
  sysagent.Account[] acclist = new sysagent.Account[accDic.size()];
  int count = 0;

  while (enumAccounts.hasMoreElements()) {
    acclist[count] = (sysagent.Account) enumAccounts.nextElement();
    count++;
  }
```

```
    return acclist;
  }

  public void delete(String name)
  {
    // get IOR
    sysagent.Account acc = (sysagent.Account) accDic.get(name);
    // removes object with key <name> from dictionary
    accDic.remove(name);
    System.out.println("Account " + name + " deleted.");
    // and deactivate transient object
    boa.deactivate_obj(acc);
    System.out.println("Deleted object with IOR: " + acc);
  }

  // public void create()
  // not yet implemented
}
```

Erzeugen eines Event Channel, Kapitel 6.3.7

```
import org.omg.CosEventComm.*;
import org.omg.CosEventChannelAdmin.*;
import com.visigenic.vbroker.services.CosEvent.*;

[...]
      // create event channel for system agent
      EventChannel channel = null;
      channel = EventLibrary.create_channel(boa, "sysagent_channel", false);
      boa.obj_is_ready(channel);
      System.out.println(channel + " is ready.");
[...]
```

Klasse FilemonitorImpl.java, Kapitel 6.3.7

```
// Implementierung der Klasse FileSystemMonitor
// enthaelt eigene Klasse fsEventPusher
// last modified 29.10.97

import org.omg.CosEventComm.*;
import org.omg.CosEventChannelAdmin.*;
import org.omg.CORBA.SystemException;
import java.util.Calendar;
import java.util.TimeZone;
import java.text.DateFormat;
import java.util.Locale;
import mytools.hostinfo;
```

```
public class FilesystemMonitorImpl extends sysagent._FilesystemMonitorImplBase
                                   implements Runnable {

  private FilesystemImpl fs;
  private float fsFullThreshold;
  private int monInterval;
  private Thread thread;
  private ProxyPushConsumer pushConsumer =null;
  private fsEventPusher pusher;

  org.omg.CORBA.ORB orb;
  org.omg.CORBA.BOA boa;

  public FilesystemMonitorImpl ( FilesystemImpl _fs )
  {
    fsFullThreshold = (float)0.9;  // default 90%
    monInterval = 60; // default 60 sec
    fs = _fs;
    thread = new Thread(this);
    thread.setDaemon(true);
    thread.start();
    try {
      orb = org.omg.CORBA.ORB.init();
      boa = orb.BOA_init();
      EventChannel channel = null;

      channel = EventChannelHelper.bind(orb, "sysagent_channel");
      pushConsumer = channel.for_suppliers().obtain_push_consumer();
      pusher = new fsEventPusher();
      boa.obj_is_ready(pusher);
          try {
        pushConsumer.connect_push_supplier(pusher);
          }
          catch (AlreadyConnected e) {
            e.printStackTrace();
          }
    }
    catch (SystemException e) {
      e.printStackTrace();
    }
  }

  public float fsFullThreshold()
  {
    return fsFullThreshold;
  }

  public void fsFullThreshold (float threshold)
  {
    fsFullThreshold = threshold;
  }

  public int monInterval()
```

```
  {
    return monInterval;
  }

  public void monInterval(int interval)
  {
    monInterval = interval;
  }

  public void run()            // periodische Abfrage als eigener thread
  {
    while (true) {
      try {
        thread.sleep(1000*monInterval);
      }
      catch (InterruptedException e) {
        e.printStackTrace();
      }
      float usage =((float) (fs.BlockCount()-fs.BlocksAvailable()))/fs.BlockCount();
      if (usage > fsFullThreshold) {
        try {
          org.omg.CORBA.Any message = orb.create_any();
          // create time info for event
          TimeZone tz = TimeZone.getDefault();
          Calendar calendar = Calendar.getInstance(tz);
          String ts = DateFormat.getTimeInstance(DateFormat.MEDIUM,
                                 Locale.GERMANY).format(calendar.getTime());
          // fetch hostname for event
          String host = (new hostinfo()).getLocalHostname();
          message.insert_string(host + ": " + ts + ": Filesystem " + fs.Name()
                                   + " " + java.lang.Math.round(100*usage)
                                   + "% voll !!");
          pushConsumer.push(message);           // push event
        }
        catch (Disconnected e) {
          e.printStackTrace();
          pusher.disconnect_push_supplier();
                  thread.stop();
        }
      }
    }
  }
}

// Eventpusher fsEventPusher

class fsEventPusher extends _sk_PushSupplier {

  fsEventPusher ()
  {
    super();
  }
```

```
  public void disconnect_push_supplier()
  {
    try {
      _boa().deactivate_obj(this);
    }
    catch (SystemException e) {
      e.printStackTrace();
    }
  }
}
```

HTML-Rahmen Manager.html, Kapitel 6.5

```
<HTML>
<HEAD>
    <TITLE>SystemAgent Client</TITLE>
</SCRIPT>
</HEAD>

<BODY>
    <H1>SystemAgent Client</H1>
    <APPLET code="TestClientApplet.class"
            width=800 height=600>
     <param name=USE_ORB_LOCATOR value=true>
     <param name=org.omg.CORBA.ORBClass
            value=com.visigenic.vbroker.orb.ORB>
     <param name=ORBgatekeeperIOR
            value=http//ibmhegering1:15000/gatekeeper.ior>
    </APPLET>
</BODY>
</HTML>
```

Abkürzungsverzeichnis

API	Application Programming Interface
AWT	Abstract Windowing Toolkit
BOA	Basic Object Adapter
CASE	Computer Aided Software Engineering
CMIP	Common Management Information Protocol
CIM	Common Management Information Model
CO	Computational Object
CORBA	Common Object Request Broker Architecture
DBM	Database Manager
DBMS	Database Management System
DCOM	Distributed Component Object Model
DFS	Distributed Filesystem
DII	Dynamic Invocation Interface
DNS	Domain Name Service
DME	Distributed Management Environment
DMI	Desktop Management Interface
DMTF	Desktop Management Task Force
DPI	Distributed Protocol Interface
FTP	File Transfer Protocol
GDMO	Guidelines for the Definition of Managed Objects (OSI)
GIOP	General Inter-ORB Protocol
GUI	Graphical User Interface
HMMP	HyperMedia Management Protocol
HTTP	HyperText Transfer Protocol
IAB	Internet Activities Board
IETF	Internet Engineering Task Force
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IOP	Inter-ORB-Protocol
IOR	Interoperable Object Reference
IR	Interface Repository
ISO	International Organization for Standardization
ITU-T	International Telecommunications Union - Telecommunications
JDK	Java Development Kit
JNI	Java Native Interface
LAN	Local Area Network

MIB	Management Information Base
MIF	Management Information Format
MHS	Message Handling System
MLM	Mid-level Manager (IBM Systems Monitor)
MO	Managed Object
MOC	Management Object Class
MTA	Message Transfer Agent
NIS	Network Information Service
NFS	Network File System
OAD	Object Activation Daemon (VisiBroker for Java)
ODMA	Open Distributed Management Architecture
ODP	Open Distributed Processing
OMA	Object Management Architecture
OMG	Object Management Group
OMT	Object Modeling Technique
OSF	Open Software Foundation
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
QoS	Quality of Service
QRL	Query and Reporting Language (StP)
RM-ODP	Reference Model of Open Distributed Processing
RMI	Remote Method Invocation
SIA	Systems Information Agent (IBM)
SMA	Systems Management Architecture
SMIT	System Management Interface Tool
SMUX	SNMP Multiplexing (Protocol)
SNMP	Simple Network Management Protocol
StP	Software through Pictures
SW	Software
TCP	Transport Control Protocol
TCP/IP	Transport Control Protocol / Internet Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
URL	Uniform Resource Locator
VFS	Virtual File System
VM	Virtual Machine
WBEM	Web Based Enterprise Management
XDR	External Data Representation

Literaturverzeichnis

[Ban96] BAN, BELA: *Open Distributed Processing: A Reference Model For Distributed Computing.* Tutorial, IBM Research Laboratory, 1996. http://www.zurich.ibm.com/~bba.

[Der95] DERR, KURT W.: *Applying OMT.* SIGS Books, New York, Erste Auflage, 1995.

[DMT95] DESKTOP MANAGEMENT TASK FORCE, SOFTWARE WORKING COMMITTEE: *Software Standard Groups Definition, Version 2.0*, November 1995.

[Far97] FAROOQUI, KAZI: *Reference Model Of Open Distributed Processing (ODP) – A Guided Tour.* Tutorial, Department Of Computer Science, University of Ottawa, 1997. ICODP'97.

[Fla97] FLANAGAN, DAVID: *JAVA IN A NUTSHELL.* O'Reilly, Sebastopol, Zweite Auflage, 1997.

[Gut95] GUTSCHMIDT, M.: *Ein Objektmodell für ein integriertes Management von Systemdiensten mit Client/Server-Struktur.* Dissertation, Ludwig-Maximilians-Universität München, September 1995.

[GW93] GRILLO, P. und S. WALDBUSSER: *Host Resources MIB.* RFC 1514, Internet Engineering Task Force, September 1993.

[HA93] HEGERING, HEINZ-GERD und SEBASTIAN ABECK: *Integriertes Netz- und Systemmanagement.* Addison-Wesley, Bonn, Erste Auflage, 1993.

[Hai95] HAINZINGER, E.: *Objektorientierte Analyse und Implementierung von Managementinformation und -funktionalität ausgewählter systemnaher Basisdienste.* Diplomarbeit, Technische Universität München, Mai 1995.

[HKN96] HEILBRONNER, S., A. KELLER und B. NEUMAIR: *Integriertes Netz- und Systemmanagement mit modularen Agenten.* In: CAP, C. (Herausgeber): *Proceedings of SIWORK'96: Workstations und ihre Anwendungen*, Seiten 275–286, Zurich, Switzerland, Mai 1996. vdf Hochschulverlag AG an der ETH Zürich. ISBN 3-7281-2342-0.

[HNW95] HEGERING, H.-G., B. NEUMAIR und R. WIES: *Integriertes Management verteilter Systeme – Ein Überblick über den State-of-the-Art –.* Bericht 9503, Ludwig-Maximilians-Universität München, Institut für Informatik, Januar 1995.

[IBM93] IBM Corporation: *LAN NetView: Operating Systems Agents, Managed-Object Catalog*, Erste Auflage, 1993.

[IBM94] *IBM Systems Monitor: Anatomy of a Smart Agent*. International Technical Support Centers, Research Triangle Park, 1994.

[Int96a] Interactive Development Environments, Inc.: *Software through Pictures Tutorial: Getting Started with StP/OMT, Release 3*, 1996.

[Int96b] Interactive Development Environments, Inc.: *Software through Pictures User Manual: StP Core – Fundamentals of StP, Release 2*, 1996. http://www.aonix.com/Products/StP/stp.html.

[Int96c] Interactive Development Environments, Inc.: *Software through Pictures User Manual: Creating OMT Models, Release 3*, 1996.

[Int96d] Interactive Development Environments, Inc.: *Software through Pictures User Manual: Object Modeling Technique – Generating Code, Release 3*, 1996.

[ISO91] ISO/IEC JTC1/SC21: *Information Technology – Open Systems Interconnection – Structure of Management Information – Part 1: Management Information Model*, November 1991. Chapter 5.2.

[ISO93] ISO/IEC: *Information Technology – Open Systems Interconnection – Systems Management – Part 2: State Management Function*, Juni 1993.

[ISO95a] ISO/IEC JTC1/SC21 WG4: *Open Distributed Management Architecture, Committee Draft*, November 1995.

[ISO95b] ISO/IEC JTC1/SC21/WG7: *10746-1 Open Distributed Processing Reference Model Part 1: Overview*, 1995.

[ISO95c] ISO/IEC JTC1/SC21/WG7: *10746-2 Open Distributed Processing Reference Model Part 2: Foundations*, 1995.

[ISO95d] ISO/IEC JTC1/SC21/WG7: *10746-3 Open Distributed Processing Reference Model Part 3: Architecture*, 1995.

[ISO95e] ISO/IEC JTC1/SC21/WG7: *Recommendation X.904: Basic Reference Model of Open Distributed Processing - Part 4: Architectural Semantic Amendment*, 1995.

[Jav97] JavaSoft: *Java Native Interface Specification, Release 1.1*, Mai 1997. ftp://ftp.javasoft.com/docs/jdk1.1/jni.ps.

[Joy] Joyner, Ian: *Open Distributed Processing: Unplugged!* http://www.dstc.edu.au/AU/research_news/odp.

[KF94] Kille, S. und N. Freed: *Network Services Monitoring MIB*. RFC 1565, Internet Engineering Task Force, Januar 1994.

[KH96] Krieger, Uwe und Stephan Haslbeck: *MNM-UNIX-MIB*. SNMP MIB, Ludwig-Maximilans-Universität München, Institut für Informatik, 1996.

[Kri94] KRIEGER, U.: *Konzeption einer Managementinformationsbasis für das Management von UNIX-Endsystemen.* Diplomarbeit, Technische Universität München, Mai 1994.

[Lin95] LININGTON, P. F.: *RM-ODP: The Architecture.* In: RAYMOND, KERRY und LIZ ARMSTRONG (Herausgeber): *Proceedings of the third IFIP International Conference on Open Distributed Processing ICODP'95*, Seiten 15–33, Brisbane, Australia, 1995.

[Neu93] NEUMAIR, B.: *Objektorientierte Modellierung von Kommunikationsressourcen für ein integriertes Performance Management.* Dissertation, Technische Universität München, Februar 1993.

[Neu97] NEUMAIR, BERNHARD: *Ein Managementmodell für verteilte Systemdienste und Anwendungen.* Internes Papier, Ludwig-Maximilians-Universität München, Institut für Informatik, 1997.

[Obj95] OBJECT MANAGEMENT GROUP: *CORBA 2.0 Specification*, 1995. http://www.omg.org.

[OH97] ORFALI, ROBERT und DAN HARKEY: *Client/Server Programming with Java and CORBA.* John Wiley & Sons, Inc., New York, Erste Auflage, 1997.

[OHE96] ORFALI, ROBERT, DAN HARKEY und JERI EDWARDS: *Distributed Objects Survival Guide.* John Wiley & Sons, Inc., New York, 1996.

[Rat97] RATIONAL SOFTWARE CORPORATION: *Unified Modeling Language Summary, Version 1.1*, September 1997. http://www.rational.com/uml/index.shtml.

[Ray95] RAYMOND, KERRY: *Reference Model of Open Distributed Processing (RM-ODP): Introduction.* Tutorial, Centre for Information Technology Research, University of Queensland, Brisbane 4072 Australia, 1995.

[RBP+91] RUMBAUGH, JAMES, MICHAEL BLAHA, WILLIAM PREMERLANI, FREDERICK EDDY und WILLIAM LORENSEN: *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, Erste Auflage, 1991.

[San91] SANTIFALLER, MICHAEL: *TCP/IP AND NFS, Internetworking in a UNIX Environment.* Addison-Wesley, Bonn, Erste Auflage, 1991.

[Sch96a] SCHMIDT, DOUGLAS C.: *An Overview of OMG CORBA Event Services.* Washington University, St. Louis, 1996. http://www.cs.wustl.edu/~schmidt.

[Sch96b] SCHMIDT, DOUGLAS C.: *An Overview of the Common Object Request Broker Architecture (CORBA).* Washington University, St. Louis, 1996. http://www.cs.wustl.edu/~schmidt.

[Sch96c] SCHMIDT, DOUGLAS C.: *Object-Oriented Network Programming: Writing CORBA Applications.* Washington University, St. Louis, 1996. http://www.cs.wustl.edu/~schmidt.

[Sch96d] SCHMIDT, S.: *Portierung eines bestehenden Systemmanagementagenten auf IBM AIX.* Fortgeschrittenenpraktikum, Technische Universität München, März 1996.

[Sch97] Schreiner, Hans-Günther: *Eine generische Managementarchitektur für offene heterogene Rechnernetze.* In: *VDI Fortschrittberichte*, Nummer 481 in *Reihe 10: Informatik, Kommunikationstechnik.* VDI, Karlsruhe, 1997.

[Sie96] Siegel, Jon (Herausgeber): *CORBA Fundamentals and Programming.* John Wiley & Sons, New York, Erste Auflage, 1996.

[Sie97] Siegert, Andreas: *The AIX Survival Guide.* Addison-Wesley Longman Inc., Harlow, Erste Auflage, 1997.

[Sir96] Sirtl, H.: *Objektorientierte Modellierung von Workstations für ein integriertes Systemmanagement.* Fortgeschrittenenpraktikum, Technische Universität München, Mai 1996.

[SK97] Saperia, Jonathan und Cheryl Krupczak: *Definitions of System-Level Managed Objects for Applications.* Internet Draft, IAB, 10 1997. gopher://ds0.internic.net:70/00/internet-drafts/draft-ietf-applmib-sysapplmib-09.txt oder http://info.internet.isi.edu:/0/in-drafts/files/draft-ietf-applmib-sysapplmib-09.txt.

[SKPK97] Saperia, Jonathan, Cheryl Krupczak, Randy Presuhn und C. Kalbfleisch: *Application Management MIB.* Internet Draft, IAB, 08 1997. gopher://ds0.internic.net:70/00/internet-drafts/draft-ietf-applmib-mib-04.txt oder http://info.internet.isi.edu:/0/in-drafts/files/draft-ietf-applmib-mib-04.txt.

[Sta96] Stallings, William: *SNMP, SNMPv2 and RMON.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1996.

[Sta97] Stal, Michael: *World Wide CORBA – verteilte Objekte im Netz.* OBJEKTspektrum, 6, 1997.

[Ste91] Stern, Hal: *Managing NFS and NIS.* O'Reilly & Associates, Sebastopol, Erste Auflage, 1991.

[Sun87] Sun Microsystems: *XDR: External Data Representation standard.* RFC 1014, Internet Engineering Task Force, Juni 1987.

[Sun88] Sun Microsystems: *RPC: Remote Procedure Call Protocol specification: Version 2.* RFC 1057, Internet Engineering Task Force, Juni 1988. (Obsoletes RFC 1050).

[Sun89] Sun Microsystems: *NFS: Network File System Protocol specification.* RFC 1094, Internet Engineering Task Force, März 1989.

[TIN94a] Telecommunications Information Networking Architecture Consortium: *Computational Modelling Concepts, Draft 2.0*, Dezember 1994. http://www.tinac.com.

[TIN94b] Telecommunications Information Networking Architecture Consortium: *Engineering Modelling Concepts (DPE Architecture), Draft 2.0*, Dezember 1994. http://www.tinac.com.

[VD97] Vogel, Andreas und Keith Duddy: *Java Programming with CORBA.* John Wiley & Sons, Inc., New York, Erste Auflage, 1997. http://www.wiley.com/compbooks/vogel.

[Vis97a] VISIGENIC SOFTWARE, INC.: *VisiBroker for Java Gatekeeper Guide, Version 3.0*, 1997. http://www.visigenic.com/techpubs/.

[Vis97b] VISIGENIC SOFTWARE, INC.: *VisiBroker for Java Programmer's Guide, Version 3.0*, 1997. http://www.visigenic.com/techpubs/.

[Vis97c] VISIGENIC SOFTWARE, INC.: *VisiBroker for Java Reference Manual, Version 3.0*, 1997. http://www.visigenic.com/techpubs.

[Vis97d] VISIGENIC SOFTWARE, INC.: *VisiBroker Naming and Event Services Programmer's Guide, Version 3.0*, 1997. http://www.visigenic.com/techpubs/.

[Vog97] VOGEL, ANDREAS: *Java, CORBA and the Web.* Tutorial, IFIP WG6.1, 1997. icodp/icdp '97.