INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Eignung von JMAPI als Basis integrierter Managementanwendungen

Bearbeiter:	Christian Schiller
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Boris Gruschke Stephen Heilbronner

INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Eignung von JMAPI als Basis integrierter Managementanwendungen

Bearbeiter:	Christian Schiller
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Boris Gruschke Stephen Heilbronner
Abgabetermin:	15. August 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. August 1998

....

(Unterschrift des Kandidaten)

Zusammenfassung

Web-Browser werden in letzter Zeit nicht nur für ihre ursprüngliche Aufgabe der Informationspräsentation im Rahmen des World Wide Web verwendet, sondern gewinnen zunehmend als Benutzeroberfläche für verteilte Anwendungen an Bedeutung. Auch im Bereich des Netz- und Systemmanagements sind Trends zum Einsatz Web-basierter Managementanwendungen erkennbar. Viele Hersteller von Hardware-Komponenten sind deswegen inzwischen dazu übergegangen, Web-Server auf den Komponenten zu integrieren um ein Management der Komponenten über eine Browseroberfläche zu ermöglichen. Ein Rahmen für die Entwicklung integrierter Web-basierter Managementanwendungen, die über das Komponentenmanagement hinausgehen, war bisher jedoch nicht vorhanden.

Sun Microsystems und JavaSoft entwickeln zur Zeit eine Reihe von Programmierschnittstellen (Application Programming Interfaces), welche die Erstellung Java-basierter Netz- und Systemmanagementanwendungen erleichtern und standardisieren sollen. Diese Schnittstellen werden unter dem Namen Java Management API (JMAPI) zusammengefaßt. In der vorliegenden Arbeit wird untersucht, inwieweit sich JMAPI als Basis integrierter Managementanwendungen eignet.

Das Common Information Model (CIM) der Desktop Management Task Force (DMTF) stellt ein herstellerübergreifendes Informationsmodell dar, welches als Grundlage für Web-Based Management-Lösungen entwickelt wurde. In der vorliegenden Arbeit ist mit Hilfe einer Reihe von `Perl`-Skripten eine Möglichkeit erarbeitet worden, eine CIM-Beschreibung von Objektklassen im Managed Object Format (MOF) in eine JMAPI-konforme Beschreibung zu transformieren und so für JMAPI nutzbar zu machen. Damit wird der Nachweis der Möglichkeit einer Integration anderer Managementarchitekturen, die ebenfalls dieses Informationsmodell verwenden, erbracht.

Für Fragen der Gateway-Konstruktion wird gezeigt, wie sich CIM-Assoziationen, die zur Modellierung der Beziehungen zwischen Managed Objects eingesetzt werden, auf den Topology Service der Object Management Group (OMG) abbilden lassen. Ferner wird ein Vergleich des in JMAPI integrierten Event Service mit dem Topology Service der OMG gezogen.

Hinsichtlich der Integration von Agenten anderer Managementarchitekturen wird speziell auf CORBA-Agenten und auf Agenten, die mit dem ebenfalls von Sun Microsystems entwickelten Java Dynamic Management Kit (JDMK) erstellt wurden, eingegangen. Für einen am Lehrstuhl vorhandenen CORBA-Agenten wurde im Rahmen dieser Arbeit eine JMAPI-konforme Oberfläche erstellt und so der Einsatz von JMAPI zur Oberflächenintegration vorgeführt.

Schließlich wird eine allgemeine Bewertung von JMAPI auf Basis des Informationsmodells, Organisationsmodells, Kommunikationsmodells und Funktionsmodells gegeben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

1.1 Web-Based Management

Während das World Wide Web in der ersten Zeit seiner Entwicklung nur der Präsentation mehr oder weniger statischer Information diente, rückt mittlerweile immer mehr eine dynamisch-interaktive Komponente in den Vordergrund. Web-Seiten können durch Einsatz von CGI-Skripten dynamisch generiert werden und gestatten somit die Darstellung individuell angepaßter Teilinformation entsprechend den Wünschen des Benutzers, wie dies zum Beispiel bei Anfragen an eine Datenbank über eine Web-Schnittstelle der Fall ist. Darüberhinaus erlaubt der Einsatz von Java-Applets noch weitergehende Möglichkeiten bei der dynamischen Gestaltung von Web-Seiten, da für eine Aktualisierung der dargestellten Information nun ein Neuladen der Seite nicht mehr nötig ist. Zudem stellt der Browser im Zusammenhang mit Java-Applets eine Plattform für verteilte Anwendungen dar, die von der zugrundeliegenden Betriebssystem- und Hardware-Plattform unabhängig ist. Da mittlerweile nahezu jeder Benutzer mit den Navigationstechniken der Browseroberfläche vertraut ist, hat eine Applikation, deren Benutzeroberfläche auf der Web Browsing Metapher basiert, den Vorteil der leichten Erlern- und intuitiven Durchschaubarkeit.

Seit einiger Zeit gibt es Bestrebungen, Web-basierte Applikationen auch für das Netz- und Systemmanagement zu entwickeln und einzusetzen *(Web-Based Management)* [Lor98].

Der Browser kann zum einen dazu verwendet werden, mittels dynamisch erzeugter HTML-Seiten managementrelevante Information zu präsentieren. Bei klassischem Vorgehen ohne Einsatz von Java Applets sind hier zwei verschiedene Ansätze denkbar. Zum einen können Daten über einen gewissen Zeitraum gesammelt und daraus periodisch HTML-Seiten generiert werden, die dann von einem Client abrufbar sind. Alternativ hierzu kann die HTML-Seite erst auf Anfrage hin erstellt werden. Die hierbei gelieferte Information ist in diesem Fall aktueller, allerdings i.d.R. auf Kosten der Antwortzeit, da die Seite im Gegensatz zum vorhergehenden Fall noch nicht abrufbereit auf dem Server vorliegt. Die erste Methode bietet sich für die Darstellung von Statistikinformation an, während die zweite Methode eher für die Präsentation von Statusinformation geeignet ist.

Durch Nutzung zusätzlicher Browser-Funktionalität kann über die oben erwähnten Möglichkeiten der *Überwachung* von Ressourcen hinaus die Fähigkeit zur *Steuerung* realisiert werden. Beispielsweise bietet sich das Laden spezieller Java-Applets von einem Management Server an, die es dem Benutzer/Administrator gestatten, über die von den Applets bereitgestellten Schnittstellen interaktive Maßnahmen zu ergreifen. Auch die Anzeige von Statusdaten kann mittels Applets in Echtzeit umgesetzt werden. Weil mehrere Browser gleichzeitig Daten und Applets von einem Web Server laden können, resultiert aus diesem Modell automatisch eine Multi-User-Fähigkeit. Gleichzeitig ergeben sich hieraus jedoch Fragen der Mehrbenutzersynchronisation und des Zugriffsschutzes.

Da es sich beim Web-Based Management um ein verhältnismäßig neues Gebiet handelt, existieren noch keine etablierten Standards. Hinsichtlich des Informationsmodells hat die von Microsoft mitbegründetete Web-Based Enterprise Management Initiative (WBEM)[1], deren organisatorische Leitung mittlerweile von der Desktop Management Task Force (DMTF) übernommen wurde, ein objektorientiertes Informationsmodell, das Common Information Model (CIM) [Des98], erarbeitet, welches in einer Web-basierten Management-Umgebung einsetzbar ist. Die Weiterentwicklung von CIM wird z. Z. von der DMTF betrieben. Was das Kommunikationsmodell betrifft, konnte sich das von WBEM ebenfalls vorgeschlagene Hyper Media Management Protokoll (HMMP) in der IETF nicht durchsetzen [Gio98a].

1.2 Aufgabenstellung

Parallel zu den Bestrebungen im Rahmen von WBEM gibt es mit Sun Microsystems und JavaSoft eine weitere Gruppe, die es sich zur Aufgabe gemacht hat, Standards auf dem Gebiet des Web-Based Managements zu erarbeiten. Mit dem Java Management API (JMAPI) verfolgt JavaSoft gegenwärtig das Ziel, die Erstellung von Web-basierten Management-Lösungen auf Basis von Java-Applets zu standardisieren und zu vereinfachen. Ausgehend von einem speziellen Architekturmodell werden dem Anwendungsentwickler Schnittstellen bereitgestellt, die einerseits die Erstellung einer einheitlichen Benutzeroberfläche auf der Basis von Java-Applets erleichtern sollen. Zum anderen sollen auch die oben angesprochenen Probleme der Mehrbenutzersynchronisation mittels eines integrierten Transaktionsmechanismus und des Zugriffsschutzes adressiert werden. Als Informationsmodell wurde ebenfalls CIM zugrundegelegt. Die Kommunikation der Komponenten basiert auf Java Remote Method Invocation (RMI) [Sun98d]. Obwohl die Entwicklung von JMAPI noch nicht abgeschlossen ist, läßt sich dennoch der Aufbau in seiner Grobstruktur schon jetzt erkennen, so daß eine vorläufige Bewertung vorgenommen werden kann.

In der vorliegenden Arbeit soll untersucht werden, inwieweit sich JMAPI als Basis für integrierte Managementanwendungen eignet, und ob die obengenannten Ziele, die bei der Entwicklung von JMAPI verfolgt wurden, erreicht werden konnten.

[1] `http://wbem.freerange.com`

1.3 Aufbau der Arbeit

Kapitel 2 gibt eine kurzen Überblick über JMAPI, stellt das Organisationsmodell, Informationsmodell und Kommunikationsmodell vor und erläutert die einzelnen Komponenten, aus denen sich JMAPI zusammensetzt. Ferner wird darauf eingegangen, inwiefern sich JMAPI von einer Management-Plattform unterscheidet.

Kapitel 3 geht näher auf die zentrale Komponente von JMAPI, den *Managed Object Server*, ein und führt die von ihm zur Verfügung gestellten Dienste auf. JMAPI Managed Objects werden vorgestellt, und es wird gezeigt, wie man neue Managed-Objekt-Klassen anlegen kann. Dieses Kapitel schafft die Grundlagen für das Verständnis der im folgenden Kapitel vorgestellten Skripten zur Integration der ebenfalls im nächsten Kapitel beschriebenen CIM-Klassen in das JMAPI-Framework.

Kapitel 4 widmet sich im ersten Teil der Beziehung zwischen JMAPI und dem Common Information Model (CIM) der Desktop Management Task Force (DMTF). Aufbauend auf den Erläuterungen von Kapitel 3 wird aufgezeigt, wie eine im Managed Object Format (MOF) vorliegende CIM-Objektklassenmodellierung in ein für JMAPI geeignetes Format transformiert werden kann. Hierzu wird eine Reihe von im Rahmen der vorliegenden Arbeit entwickelten `Perl`-Skripten vorgestellt, die diese Aufgabe erledigen. Der zweite Teil dieses Kapitels untersucht, wie JMAPI Beziehungen zwischen MOs modelliert. Es wird gezeigt, wie sich die hierbei relevanten CIM-Associations auf den Topology Service der Object Management Group (OMG) abbilden lassen. Eine Zusammenfassung am Ende des Kapitels enthält die hierbei gewonnenen Ergebnisse.

Kapitel 5 stellt das JMAPI Event Model vor und zieht einen Vergleich zum Notification Service der OMG.

Kapitel 6 geht auf die Rolle der Agenten ein, und zeigt, wie JMAPI mit Agenten anderer Managementarchitekturen (SNMP, CORBA, JDMK) zusammenarbeiten kann.

Kapitel 7 ist der JMAPI-Benutzeroberfläche gewidmet. Für einen am Lehrstuhl vorhandenen CORBA-Agenten wurde im Rahmen der Diplomarbeit eine JMAPI-konforme Benutzeroberfläche erstellt um die Möglichkeiten zur Oberflächenintegration zu untersuchen und an einem Beispiel zu demonstrieren.

Kapitel 8 untersucht, inwieweit sich JMAPI für das integrierte Management eignet. Nach einer Aufzählung der Kriterien, die an das integrierte Management zu stellen sind, folgt eine Bewertung anhand des Informationsmodells, Organisationsmodells, Kommunikationsmodells und des Funktionsmodells.

In Kapitel 9 sind einige Erfahrungen zusammengestellt, die bei der Arbeit mit der JMAPI-Referenzimplementierung gemacht wurden.

Kapitel 10 schließlich liefert eine Zusammenfassung der Arbeit und gibt einen Ausblick auf zu erwartende Entwicklungen und auf wünschenswerte Neuerungen.

Kapitel 2

JMAPI-Überblick

Die vorliegende Arbeit befaßt sich mit dem Java Management API (JMAPI). Hierbei handelt es sich um eine von JavaSoft geplante Standarderweiterung der JDK Klassenbibliothek, die ein Framework für die Entwicklung von Applikationen im Bereich des Netz-, System- und Anwendungsmanagements bereitstellen soll.

2.1 Entwicklungssstand

Zum jetzigen Zeitpunkt ist JMAPI noch in der Entwicklungsphase. Im wesentlichen stützt sich der folgende Text auf den *JMAPI Developer's Release 0.5* [JMA97], welcher im Oktober 1997 veröffentlicht wurde. Neuere JMAPI-Versionen sowie eine endgültige Spezifikation waren für Ende 1997 angekündigt, lagen jedoch bei Fertigstellung dieser Arbeit immer noch nicht vor.

2.2 Teilmodelle des Managements

Im folgenden soll ein Überblick über das Organisationsmodell, das Informationsmodell und das Kommunikationsmodell von JMAPI gegeben werden. Das Funktionsmodell findet im Falle von JMAPI keine Ausprägung.

2.2.1 Organisationsmodell

JMAPI geht von folgendem Organisationsmodell aus (vgl. Abbildung 2.1):

In der zu verwaltenden Managementdomäne existieren eine Reihe von Hardware-Komponenten **(Appliances)**, die mit Hilfe von JMAPI zu administrieren sind. Für die hierzu notwendige Überwachung und Steuerung werden **Agenten** eingesetzt, die jeweils auf den Appliances laufen. Sie realisieren die eigentliche Managementfunktionalität auf den Appliances. Die Koordination der Aktivitäten der Agenten erfolgt durch einen Server-Prozeß auf einem speziellen Host, dem **Managed Object Server**. Neben der Kommunikation mit den Agenten übernimmt der Managed Object Server die Verwaltung von in

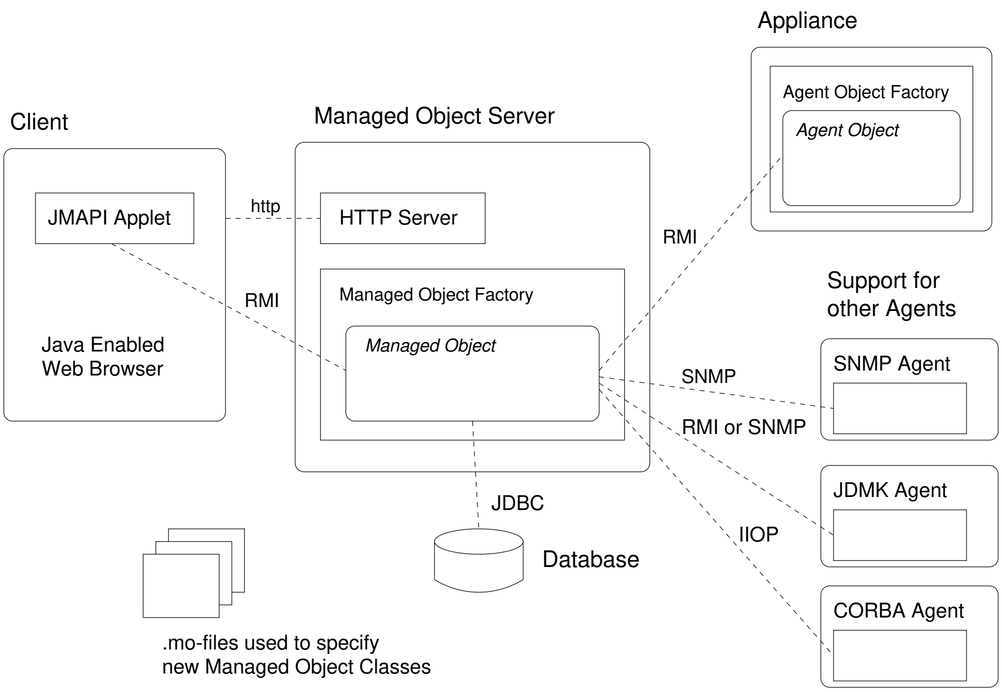


Abbildung 2.1: JMAPI Organisations- und Informationsmodell

Java implementierten Managed Objects[1], welche die abstrahierte Managementinformation beinhalten. Managementaktionen, die sich in der Manipulation von Managed Objects äußern, werden von diesen auf entsprechende Aktionen der Agenten abgebildet. Die Managed Objects werden in einer auf dem Server vorhandenen Datenbank persistent gespeichert.

Das JMAPI Organisationsmodell macht die Annahme, daß es sich bei den Agenten i. d. R. um in Java implementierte *Agent Objects* handelt, die von einer Agent Object Factory verwaltet werden. Der Einsatz anderer Agenten, wie z. B. SNMP- oder CORBA-Agenten ist jedoch auch möglich, und wird im Falle der SNMP-Agenten durch eine mitgelieferte SNMP-Klassenbibliothek unterstützt.

Die Interaktion mit dem System erfolgt über spezielle **Clients**. Hierbei handelt es sich um Java Applets oder Applications, welche die Möglichkeit bieten, relevante Managementinformation in einer für den Benutzer konsistenten Weise darzustellen, und es zudem gestatten, Managed Objects zu modifizieren, zu löschen oder neue Managed Objects anzulegen.

2.2.2 Informationsmodell

Das Informationsmodell von JMAPI stützt sich auf das Common Information Model (CIM) der Desktop Management Task Force (DMTF) [Des98]. Die mit

[1] Während in der Literatur der Begriff *Managed Object* meist allgemein die managementrelevante Information zur Beschreibung eines zu verwaltenden, zu überwachenden oder zu steuernden Betriebsmittels bezeichnet (vgl. [HA93]), sei in der vorliegenden Arbeit hiermit immer ein auf dem MO Server vorhandenes Java-Objekt gemeint.

JMAPI ausgelieferten MO-Basisklassen (vgl. Abschnitt 3.2) stellen eine Implementierung einer inzwischen nicht mehr aktuellen Version von CIM dar. Nach Aussage von Sun/JavaSoft ist es geplant, in die endgültige JMAPI Version wieder MO-Basisklassen zu integrieren, die mit der zur Zeit gültigen CIM Version 2.0 in Einklang stehen [Gof98].

Ziel von CIM ist die Bereitstellung eines auf einem objektorientierten Ansatz basierenden Management Schemas, das als Ausgangspunkt für eine verfeinerte Modellierung dienen kann.

Das CIM Management Schema unterteilt sich in drei Teilbereiche:

- Das **Core Model** modelliert Begriffe, die für alle Bereiche des Managementes relevant sind.
- Das **Common Model** ist ein Informationsmodell, das spezifische Managementbereiche umfaßt, die im Gegensatz zu den im folgenden genannten Extension Schemas aber implementierungs- und technologieunabhängig sind. Folgende Schemata wurden definiert: *Systems Schema*, *Applications Schema*, *Network Schema* und *Device Schema*.
- Die **Extension Schemas** bieten technologiespezifische Erweiterungen des Common Models und gehen auf plattformabhängige Besonderheiten ein.

Core Model und Common Model werden auch unter dem Begriff CIM Schema zusammengefaßt.

Zusätzlich zu den obengenannten Schemata existiert das CIM Meta Schema, welches die Begriffe, einschließlich ihrer Verwendung und Semantik, die zur Formulierung der Teilmodelle verwendet werden, genau definiert.

Die Beschreibung der Management Information erfolgt in einer an IDL angelehnten Sprache, die als *Managed Object Format* (MOF) bezeichnet wird.

2.2.3 Kommunikationsmodell

Gemäß Abbildung 2.1 wird Java Remote Method Invocation (RMI) als Protokoll für die Kommunikation der Clients mit dem Managed Object Server verwendet. Für die Kommunikation der MOs mit den Agenten sind sowohl RMI als auch SNMP vorgesehen, jedoch können durchaus andere Kommunikationsprotokolle, wie beispielsweise CORBA IIOP zwischen den MOs und den Agenten eingesetzt werden. Allerdings hat die Verwendung von RMI als alleinigem Kommunikationsprotokoll den Vorteil, auf diese Weise den in JMAPI integrierten Transaktionsmechanismus nutzen zu können (vgl. Abschnitt 6.6).

2.3 Einzelheiten zur Referenzimplementierung

In der endgültigen Form soll JMAPI die Schnittstellen zwischen den Komponenten des oben dargestellten Managementmodells definieren, sowie eine Referenzimplementierung liefern. Die Schnittstellenspezifikation ist noch in der Entwicklung begriffen. Die Bestandteile der Referenzimplementierung von [JMA97] werden im folgenden näher erläutert.

2.3.1 Java-Klassen und Style Guides

Die z. Z. aktuelle Version [JMA97] umfaßt folgende Komponenten:

- Eine Reihe von Java-Packages, die eine Vorversion der Referenzimplementierung darstellen (vgl. Tabelle 2.1).
- Dokumentation zu den Packages im `javadoc`-Format [Sun97g].
- Ein *Programmer's Guide* [Sun97c], welcher Hinweise zur Verwendung der mitgelieferten Software gibt.
- Zwei Style-Guides ([Sun97e] und [Sun97d]), die Vorgaben für eine Benutzeroberfläche mit einheitlichem *Look and Feel* geben.

Ursprünglich war außerdem geplant, die Erstellung einer integrierten Online-Hilfe mittels spezieller Klassen zu unterstützen [Sun97a]. Die entsprechenden Klassen sind jedoch in [JMA97] als `deprecated` markiert, und sollen dementsprechend nicht mehr verwendet werden. In der angekündigten Version 1.0 werden sie nicht mehr enthalten sein, da JavaSoft inzwischen eine eigene JavaHelp API erarbeitet [Sun98c]. Im folgenden wird daher dieser Teilaspekt von JMAPI nicht näher untersucht.

Package	**Kurzbeschreibung**
sunw.admin.arm.agents	Dient der Kommunikation der MOs mit den Agents
sunw.admin.arm.common	Datentypen, die von allgemeinem Interesse sind
sunw.admin.arm.event	Klassen und Interfaces für JMAPI Events
sunw.admin.arm.manager	Managed Object Server Klassen und Interfaces
sunw.admin.arm.mo.base	JMAPI Managed Object Basisklassen
sunw.admin.arm.mo.enum	Klassen für persistente Aufzählungen
sunw.admin.arm.mo.event	Weitere Klassen und Interfaces für JMAPI Events
sunw.admin.arm.rmi	Enthält als wichtigste Klasse die `MOFactory`
sunw.admin.avm.base	GUI Klassen
sunw.admin.avm.help	ehemalige Klassen für Benutzer-Help-Funktion, wird nicht mehr unterstützt
sunw.admin.snmp	SNMP Klassen

Tabelle 2.1: JMAPI Packages

2.3.2 Clients

Clients stellen eine graphische Oberfläche zur Verfügung, die es gestattet, managementrelevante Information auszuwählen, darzustellen und interaktiv Managementaktionen auszuführen. Bei einem Client handelt es sich entweder um eine eigenständige Java Application oder aber um einen Java-fähigen Web Browser[2], der geeignete JMAPI Applets laden und ausführen kann.

2.3.3 Managed Object Server

Der Managed Object Server stellt die zentrale Komponente des Systems dar. Er stellt die in Abschnitt 3.1 beschriebenen Dienste zur Verwaltung der Managed Objects bereit.

Ein Web Server dient den Clients zum Laden der JMAPI Applets. Da Applets in der Regel nur Netzverbindungen zu dem Host herstellen dürfen, von dem sie geladen wurden, ist es nötig, daß der Web Server auf demselben Host wie der Managed Object Server läuft. Beim Start des Systems lädt der Client über eine bekannte URL auf dem Web Server ein Start-Applet. Von diesem Applet aus können dann andere Applets geladen werden, mittels derer die eigentlichen Managementaufgaben durchgeführt werden können.

Als Host für den Managed Object Server kommt entweder ein PC mit Windows NT 4.0 oder eine SPARC-Workstation unter Solaris 2.5 (oder neuer) in Frage. Andere Betriebssysteme werden zur Zeit nicht unterstützt [Sun97b].

2.3.4 Datenbank

Damit die durch die Managed Objects repräsentierte Information bei Ausfällen oder dem beabsichtigten Herunterfahren des Managed Object Servers nicht verloren geht, benötigt dieser eine Möglichkeit zur dauerhaften Speicherung von Managed Objects. Hierzu ist bei [JMA97] eine Datenbank nötig. Folgende Datenbanken werden hierbei unterstützt:

- Sybase 10.01 und 11
- Oracle 7.2.3 und 8
- andere Datenbanken, sofern sie „JDBC compliant“[3] sind.

Die Anbindung an die Datenbank kann über einen proprietären Treiber oder über eine JDBC-Schnittstelle mit entsprechendem JDBC-Treiber erfolgen. Nach Auskünften von der JMAPI-Mailing-List [Mai] umfassen die Anforderungen, die an die Datenbank und den Treiber gestellt werden, neben der Unterstützung des Transaktionskonzeptes auch die Abfragbarkeit von Metadaten. Die Verwendung einer Reihe einfacher, frei verfügbarer Datenbanken, die auf dem UNIX Dateisystem aufsetzen, schließt sich hierdurch aus. Bei der am Lehrstuhl vorliegenden Installation wurde eine Sybase 11 Datenbank mit dem FastForward JDBC Driver der Version 3.0 von Connect Software[4] verwendet.

[2] Bei der Auswahl eines geeigneten Browsers ist darauf zu achten, daß JMAPI auf Java 1.1 aufbaut. Browser, die diese Java-Version voll unterstützen, sind z. B. der HotJava Browser von Sun, oder der Netscape Communicator ab der Version 4.5.

[3] Die genaue Definition dieses Begriffs ist in der Dokumentation nicht auffindbar. Vgl. jedoch die Ausführungen im nächsten Absatz.

[4] `http://www.connectsw.com`

Wie in Kapitel 9 geschildert wird, gestaltet sich die Auswahl einer geeigneten Datenbank, zusammen mit deren Installation, Konfiguration und der Wahl und Installation eines passenden Treibers, oft als sehr aufwendig und zeitraubend. Für die kommenden Versionen wurde daher von JavaSoft angekündigt, daß eine Datenbank nicht mehr zwingend vorgeschrieben werden soll, sondern auch einfache Dateien für die Speicherung von Managementinformation dienen können [Gio98a].

2.3.5 Agenten

Die Agenten führen die eigentlichen Managementaktionen auf den Appliances aus. Das JMAPI Framework ermöglicht die Integration einer Vielfalt von Agenten (siehe Kapitel 6): Zum einen können in Java implementierte **Agent Objects** eingesetzt werden, die von einer **Agent Object Factory** verwaltet werden. Auch **SNMP-Agenten** werden über eine mitgelieferte SNMP Klassenbibliothek unterstützt. Als weitere Möglichkeiten bieten sich die Verwendung von **CORBA-Agenten** und **Java-Dynamic-Management-Agenten** (vgl. Abschnitt 6.4) an.

Die Agentensoftware, die für die `AgentObjectFactory` benötigt wird, kann zur Zeit in folgenden Umgebungen installiert werden[5]:

- PCs unter Windows NT 4.0 oder Windows 95
- SPARC-Workstations unter Solaris 2.5 (oder neuer)

2.4 Zusammenhang mit Management-Plattformen

Abbildung 2.2 zeigt den schematischen Aufbau einer Management-Plattform nach [HA93]. Man kann die Plattform grob in drei Teile untergliedern: Die Interaktion mit dem Benutzer erfolgt über den Oberflächenbaustein. Im mittleren Teil befinden sich die eigentlichen Managementapplikationen, zusammen mit einer Reihe von oft mitgelieferten Basisanwendungen, sowie Entwicklungswerkzeuge zur Erweiterung der Plattform. Der untere Teil umfaßt die Infrastruktur der Plattform, welche das Kernsystem, den Kommunkiationsbaustein und die Datenbank enthält.

JMAPI bildet keine vollständige Management-Plattform. Nur die in Abbildung 2.2 schraffiert gezeichneten Teile haben eine Entsprechung in JMAPI. Hierbei handelt es sich um den Oberflächenbaustein und Bausteine der Infrastruktur. JMAPI enthält insbesondere keine mitgelieferten Managementapplikationen. Diese sind vom Anwendungsentwickler selbst zu erstellen.

[5]SNMP-, CORBA- oder JDMK-Agenten benötigen diese Software nicht und unterliegen somite auch nicht dieser Einschränkung.

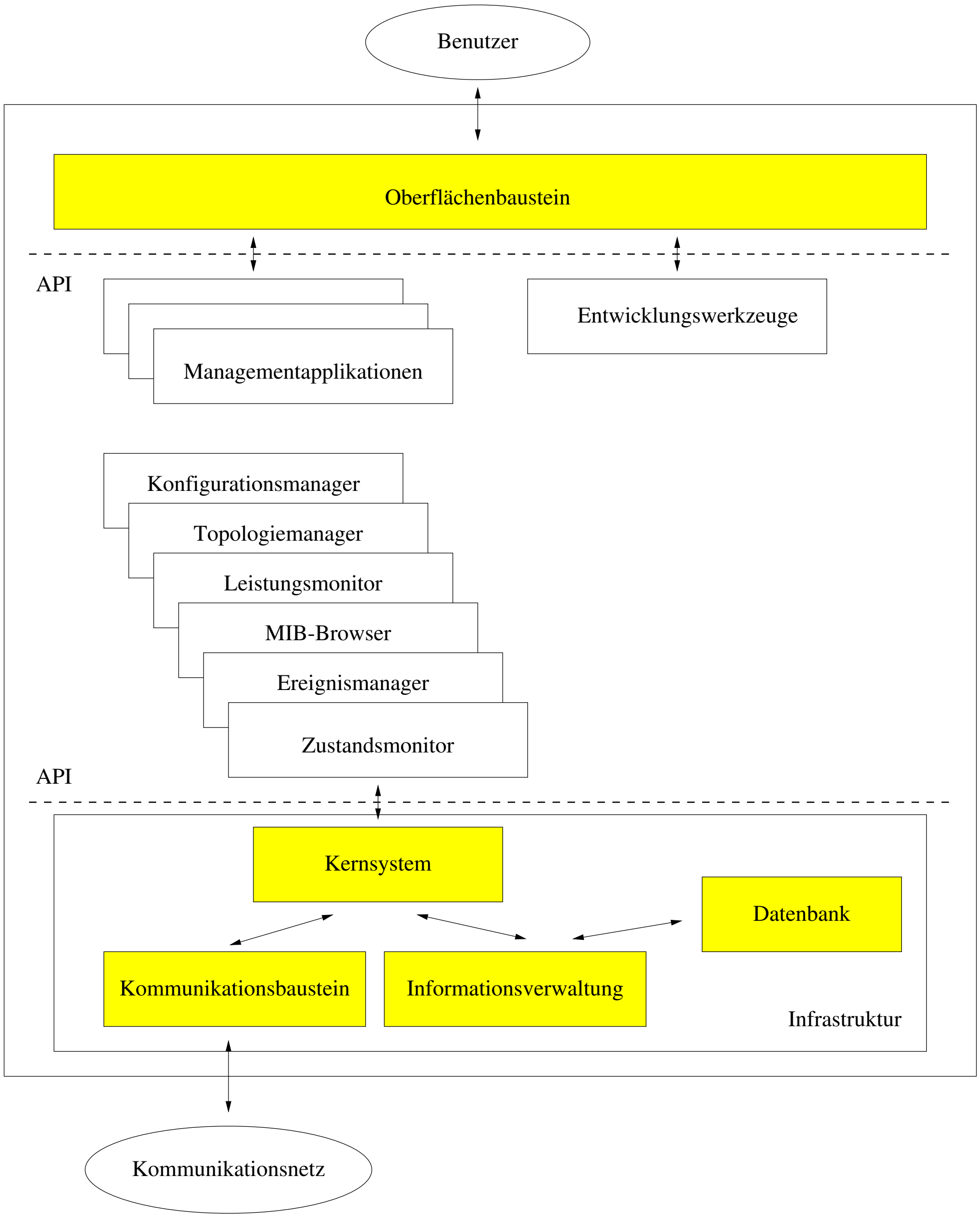


Abbildung 2.2: Schematischer Aufbau einer Management-Plattform

Kapitel 3

Der JMAPI Managed Object Server

Der Managed Object Server stellt das zentrale Kernstück der JMAPI-Architektur dar. Er bildet das Bindeglied zwischen den Web-basierten Clients und den Agenten. Die Clients greifen auf spezielle JMAPI Managed Objects zu, welche vom Managed Object Server verwaltet werden. Diese MOs stoßen ihrerseits Aktionen der Agenten an. Die Managed Objects sind Instanzen von Managed Object Klassen, die entweder Teil der Referenzimplementierung von JMAPI sind, oder aber mittels Verfeinerungen dieser Klassen vom Anwendungsprogrammierer selbst definiert werden können.

Im vorliegenden Kapitel werden zunächst die vom Managed Object Server bereitgestellten Dienste vorgestellt. Im Anschluß daran wird auf die Aufgaben und die Bedeutung der MOs näher eingegangen. Schließlich wird gezeigt, wie man vorzugehen hat, um neue MO-Klassen anzulegen.

3.1 Verfügbare Dienste

Der Managed Object Server stellt eine Reihe von Diensten zur Verfügung, die von den Clients genutzt werden können. Das [JMA97] zugrunde liegende Managed Object Server Modell hat sich als ungenügend hinsichtlich zukünftiger Erweiterungen der vom Managed Object Server bereitgestellten Dienste erwiesen. Daher wurde für zukünftige Versionen ein modularerer Aufbau angekündigt [Gio98b] (vgl. Abbildung 3.1)[1].

Es wird zwischen *Management Services* und *Framework Services* unterschieden. Zu den Management Services zählen unter anderem

[1] In [JMA97] ist die Dienststruktur nicht ganz so modular gestaltet, wie oben beschrieben. Zentrale Klasse ist hier die Klasse `ManagedObjectImpl` mit ihrem RMI Remote Interface `ManagedObject`. Diese Klasse realisiert durch die von ihr zur Verfügung gestellten Methoden einige der obigen Dienste, verwischt dadurch aber auch deren saubere Trennung. So ist beispielsweise die Verwaltung von Assoziationen, auf die in Abschnitt 4.2 eingegangen wird, ganz in die Klasse `ManagedObjectImpl` integriert. Auch der Update Service ist über die im Abschnitt 3.2 beschriebenen Methoden `performNewActions`, `performModifyActions` und `performDeleteActions` eng mit der Klasse `ManagedObjectImpl` verknüpft.

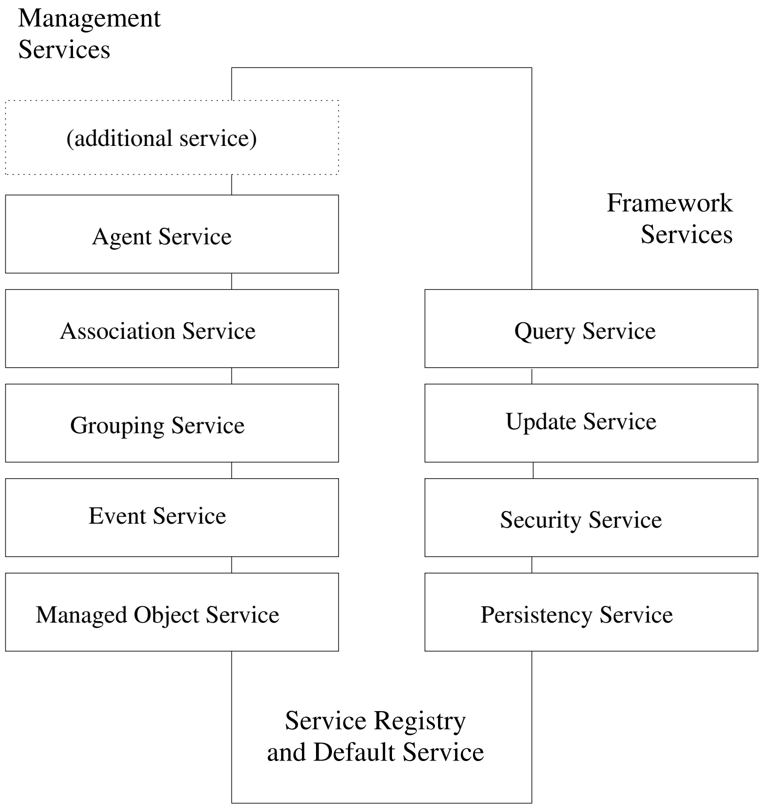


Abbildung 3.1: Aufbau des Managed Object Servers

- der *Managed Object Service*, welcher das Anlegen, Modifizieren und Löschen von Managed Objects ermöglicht.
- der *Event Service*, der asynchrone Ereignismeldungen an die Clients oder an andere interessierte Komponenten des Systems unterstützt.
- der *Grouping Service*, mittels dessen Managed Objects zu Gruppen zusammengefaßt werden können.
- der *Association Service*, der es gestattet, Associations, welche Beziehungen zwischen Managed Objects modellieren, aufzubauen und zu verwalten. In der endgültigen Version soll es sich hierbei um eine Implementierung der Associations des Common Information Model (CIM) [Des98] der Desktop Management Task Force (DMTF) handeln.
- der *Agent Service*, der die Verwaltung von und Kommunikation mit den *Agent Objects* auf den zu managenden *Appliances* übernimmt.

Bei den Framework Services gibt es

- den *Query Service*, welcher es gestattet, mittels einfacher Anfrageausdrücke *(Query Expressions)* gewünschte Managed Objects für die weitere

Bearbeitung auszuwählen. Es ist geplant, als Anfragesprache eine Untermenge von SQL92 zu verwenden [Gio98a].

- den *Update Service*, der eine Transaktionssemantik für Änderungsaktionen auf Managed Objects zur Verfügung stellt.

- den *Persistency Service*, der die dauerhafte Speicherung von Managed Objects in der Datenbank ermöglicht.

- den *Security Service*, mittels dessen ein Autorisierungsmechanismus implementiert werden kann.

Im Gegensatz zu den Framework Services, die in ihrer Anzahl und ihrem Funktionsumfang festgelegt sind, können die Management Services um einzelne Dienste erweitert werden. Hierzu soll in der nächsten JMAPI-Version ein Rahmen vorgelegt werden, der festlegt, wie neue Dienste in das Server Modell integriert werden. Ein *Service Registry* übernimmt hierbei die Verwaltung aller angemeldeten Dienste.

Abbildung 3.2 zeigt das Zusammenspiel der Management Services des Managed Object Servers.

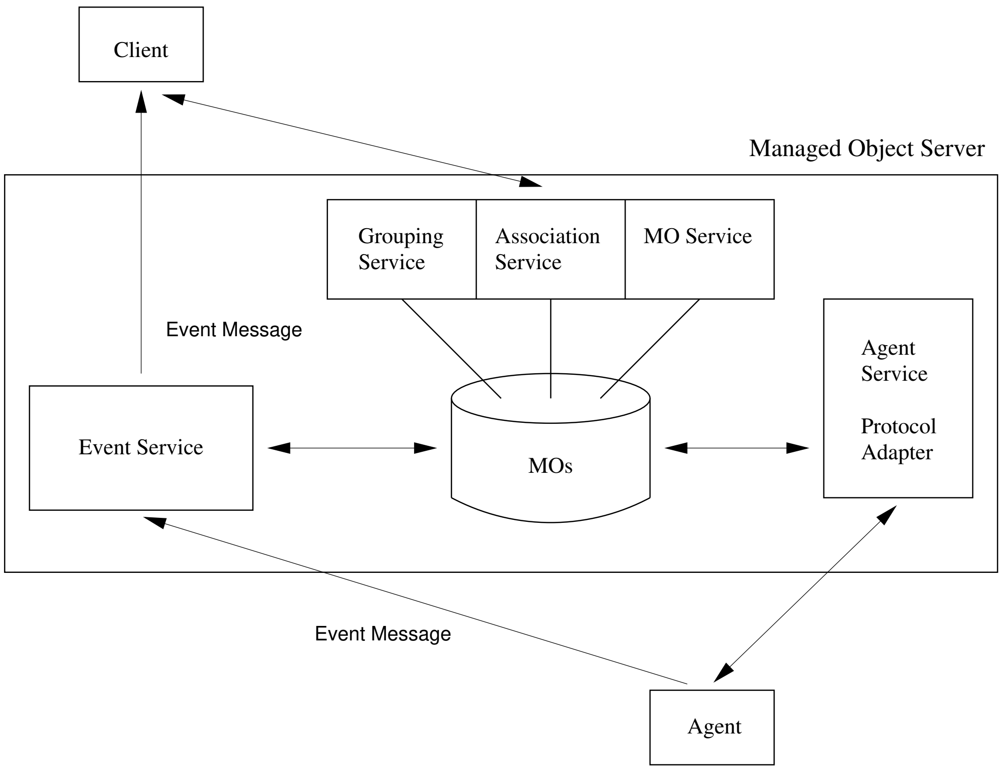


Abbildung 3.2: Management Services des MO Servers

3.2 JMAPI Managed Objects

JMAPI Managed Objects dienen der Modellierung des managementrelevanten Verhaltens und Zustands realer Ressourcen. Sie haben die folgenden zwei Aufgaben:

- Sie stellen Attribute bereit, die den Zustand der zu managenden Ressource charakterisieren. Die Ermittlung der jeweils aktuellen Attributwerte erfolgt auf zwei unterschiedliche Arten: Der Attributwert wird entweder in der Datenbank auf dem Managed Object Server gespeichert und bei einer Anfrage entsprechend aus der Datenbank abgefragt, oder er wird auf der Basis einer Operation auf einem Agenten berechnet. Bei Attributen letzteren Typs handelt es sich zumeist um Größen, die einer großen Änderungsdynamik unterworfen sind und deren Speicherung in der Datenbank deswegen nicht sinnvoll ist. Als Beispiel hierfür sei die Anzahl der gerade laufenden Prozesse auf einer Workstation genannt. Die Attribute von Managed Objects werden in der Regel in der JMAPI Datenbank gespeichert, es sei denn sie werden extra durch das Schlüsselwort `transient` gekennzeichnet. Änderungen von Attributen erfolgen im Rahmen einer *Session*, welche einen Transaktions- und Security-Kontext definiert. Für die Mehrbenutzersynchronisation wird ein Algorithmus der Optimistic Concurrency Control verwendet [Gio98a]. Wenn sich der Zustand eines Objekts der Klasse `ManagedObject` ändert, werden asynchrone Ereignismeldungen *(Notifications)* erzeugt, für die sich interessierte Observer mittels des JMAPI Event-Mechanismus registrieren können (vgl. Kapitel 5).

- Managed Objects stellen mittels der von ihnen definierten Methoden Operationen zur Verfügung, über die Client-Applikationen Steuerungsaktionen ausführen können. Zum einen gibt es die Möglichkeit, Methoden zu definieren, die von Clients explizit aufgerufen werden. Zum anderen existieren drei besondere vordefinierte Methoden (`performAddActions`, `performModifyActions` und `performDeleteActions`), die beim Ändern, Löschen und Anlegen neuer Managed Objects selbsttätig zur Ausführung gelangen (vgl. Abschnitt 6.6). Neben der Möglichkeit, diese speziellen Methoden zum Ausführen von Aktionen auf den Agenten zu nutzen, können sie beispielsweise auch dazu verwendet werden, im Falle des Hinzufügens oder Änderns von Managed Objects Konsistenzprüfungen der Attributwerte vor der Datenbanktransaktion vorzunehmen.

Das Anlegen neuer MO Instanzen erfolgt über die Methode `newObj` der Klasse `sunw.admin.arm.MOFactory`. Modifikationen von Attributwerten werden durch Aufruf der für jedes Attribut vorhandenen `set`-Methode des entsprechenden Managed Objects erzielt. Durch den Aufruf der Methode `deleteObject` kann ein MO zum Löschen markiert werden. Es ist zu beachten, daß sich sämtliche der hier aufgezählten Operationen stets innerhalb eines Update-Kontexts vollziehen müssen. Ob die Operation dann tatsächlich ausgeführt wird, hängt vom erfolgreichen Commit der Transaktion ab (vgl. Abschnitt 6.6).

JMAPI Managed Objects werden auf dem Managed Object Server instantiiert und von Clienten über die Java RMI Schnittstelle angesprochen. Zum Lieferumfang von JMAPI gehört eine Reihe von MO-Basisklassen *(Base Managed Object Classes)*, die als Ausgangspunkt für die Entwicklung eigener Managementapplikationen dienen sollen. In der endgültigen Version sollen diese Klassen eine Implementierung der CIM Klassen sein. Wurzel der Klassenhierarchie ist die Klasse `ManagedObjectImpl` mit dem Remote Interface `ManagedObject`.

Da die MO-Basisklassen sehr allgemeiner Natur sind, wird es für die Anwendungsentwicklung nötig sein, sie zu spezialisieren. Bei der Arbeit mit MO Klassen sollten Applikationen soweit oben wie möglich in der Vererbungshierarchie ansetzen, um zu gewährleisten, daß die Anwendung für eine Vielzahl von Managed Objects einsetzbar ist.

3.3 Anlegen neuer JMAPI Managed Object Klassen

3.3.1 Allgemeine Vorgehensweise

Für das Anlegen neuer Managed Object Klassen muß zunächst eine Datei erstellt werden, die Informationen über die Attribute und die Methoden des Managed Objects enthält. Diese Datei trägt den Namen der neu zu schaffenden Managed Object Klasse mit der Endung `.mo`. Soll beispielsweise eine neue Managed Object Klasse `ExampleMO` definiert werden, ist eine Datei `ExampleMO.mo` nötig. Jede neue MO Klasse muß entweder eine (direkte oder indirekte) Unterklasse von `ManagedObjectImpl` sein. Eine derartige `.mo`-Datei hat Java-Syntax (vgl. Abschnitt 3.3.2).

Nach der Erstellung der `.mo`-Datei wird diese mit dem mitgelieferten Managed Object Compiler `Moco` übersetzt. Dieser erzeugt die für die Verwendung von RMI nötigen Interfaces, Client-Stubs und Server-Skeletons, sowie eine Klasse, die der Speicherung der MO Attribute in der Datenbank dient. Durch Angabe der Option `-autoimport` kann `Moco` nach dem Übersetzungsvorgang das generierte Datenbankschema auch gleich in die Datenbank importieren. Ohne Angabe dieser Option ist es möglich, diesen Schritt später mittels des Kommandos `jrb_import` durchzuführen.

3.3.2 Aufbau der `.mo`-Datei

Die Erläuterung des Aufbaus einer `.mo`-Datei erfolgt am einfachsten an einem Beispiel. Es sei eine neue Managed Object Klasse `ExampleMO` anzulegen, die die Wurzelklasse `ManagedObjectImpl` der MO Basisklassenhierarchie spezialisiert. Wie bereits oben erwähnt ist hierzu eine Datei `ExampleMO.mo` anzulegen. Diese Datei muß die Java-Klasse `ExampleMOImpl` definieren:

```
public class ExampleMOImpl
    extends ManagedObjectImpl implements ExampleMO {
        // Attribute und Methoden
        // ...
}
```

Hierbei ist `ExampleMO` das Remote Interface, das später von JMAPI Clients benutzt wird, um mit den entsprechenden MO Instanzen auf dem MO Server zu kommunizieren. Die Klasse `ExampleMO.class` wird vom Managed Object Compiler `Moco` erzeugt. `Moco` erzeugt ebenfalls den Client-Stub und das Server-Skeleton `ExampleMOImpl_Stub.class` und `ExampleMOImpl_Skel.class`, die für die RMI Kommunikation zwischen Client und MO Server benötigt werden. `ExampleMOImpl` erbt von `ManagedObjectImpl` die Methoden für die Interaktion mit der Datenbank. Tabelle 3.1 führt die von `Moco` erzeugten Dateien tabellarisch auf.

Datei	**Bedeutung**
`ExampleMO.class`	RMI Interface für das MO
`ExampleMOImpl.class`	Implementation des Interface
`ExampleMOImpl_Stub.class`	RMI Client Stub
`ExampleMOImpl_Skel.class`	RMI Server Skeleton
`DBExampleMO.class`	Dient der persistenten Speicherung in der Datenbank
`DBExampleMO.import`	Hilfsdatei zum Erzeugen des Datenbankschemas

Tabelle 3.1: Von `Moco` erzeugte Dateien

Der Body der Funktionsdefinition von `ExampleMOImpl` enthält nun zunächst eine Auflistung aller Attribute der neuen Managed Object Klasse mit zugehörigem Datentyp. In der Regel werden alle Attribute in der Datenbank gespeichert. Ist dies für einige Attribute nicht erwünscht, etwa weil sich diese sehr oft ändern, so können diese, wie bereits erwähnt, durch Verwendung des Schlüsselwortes `transient` speziell gekennzeichnet werden.

Als Beispiel seien zwei Attribute anzulegen, eines vom Typ String, das andere vom Typ Integer, wobei jedoch nur das erste in der Datenbank zu speichern sei:

```
String          attr1;
transient int   attr2;
```

Als nächstes sind für alle Attribute `get`- und `set`-Methoden zu definieren. Operationen auf Managed Objects erfolgen immer innerhalb einer `Session`, welche einen Transaktions- und Security-Kontext verwaltet.

Für das erste Attribut in obigem Beispiel würden die `get`- und `set`-Methoden wie folgt aussehen:

```
public String getAttr1(Session session)
  throws AuthorizationException, RemoteException {
    return (String)getPropertyByName(session, "attr1");
}

public void setAttr1(Session session, String _attr1)
  throws NotInUpdateException, AuthorizationException,
```

```
            RemoteException
    {
        setPropertyByName(session, "attr1", _attr1);
    }
```

Die Methoden `getPropertyByName` und `setPropertyByName` sind vordefinierte Methoden des Interfaces `ManagedObject`, die den entsprechenden Wert aus der Datenbank lesen bzw. in die Datenbank schreiben.

Für das als `transient` definierte Attribut, das nicht in der Datenbank gespeichert wird, kann eine andere Implementierung der `get`- und `set`-Methoden gewählt werden.

Nach der Definition der Attribute und ihrer zugehörigen `get`- und `set`-Methoden können nun beliebige weitere Methoden hinzugefügt werden. Auch die drei speziellen Methoden (vgl. Abschnitt 6.6)

```
public synchronized void performAddActions(Session session) {};
public synchronized void performModifyActions(Session session) {};
public synchronized void performDeleteActions(Session session) {};
```

können umdefiniert werden.

Kapitel 4

JMAPI, CIM und der OMG Topology Service

Wie bereits in Abschnitt 2.2.2 erwähnt, ist das Common Information Model (CIM) der DMTF als Informationsmodell von JMAPI vorgesehen. Die Beschreibung von CIM-Schemata erfolgt mittels Dateien in einem speziellen Format, dem sog. *Managed Object Format* (MOF). CIM-Schemata enthalten u.a. Klassendefinitionen, die auf JMAPI-MO-Klassen abgebildet werden müssen, falls die entsprechenden CIM-Schemata im Rahmen von JMAPI verwendet werden sollen. Zur Beschreibung neuer MO-Klassen verwendet JMAPI allerdings keine MOF-Dateien, sondern Dateien mit Java-Syntax, die die Endung `.mo` tragen, und daher im folgenden als `.mo`-Dateien bezeichnet werden sollen. [JMA97] enthält keine Werkzeuge für eine Konvertierung von MOF in `.mo`- Dateien. Das vorliegende Kapitel widmet sich daher zunächst der Frage, wie eine derartige Konvertierung vorgenommen werden kann.

Neben der isolierten Betrachtung von Managed Objects ist es oft wichtig, Beziehungen zwischen Managed Objects zu modellieren. CIM bedient sich hierbei sog. *Associations*. [JMA97] kennt ferner noch sog. *References*. Der zweite Teil des Kapitels widmet sich der näheren Beschreibung dieser beiden Modellierungskonstrukte. Ferner bietet JMAPI die Möglichkeit, MOs zu Gruppen zusammenzufassen, wozu CIM keine Entsprechung kennt. Gruppen können zur Domänenbildung im Rahmen des Organisationsmodells eingesetzt werden (vgl. Abschnitt 8.3). Auf die Möglichkeiten, die JMAPI zur Bildung von Gruppen bietet, wird ebenfalls in diesem Kapitel kurz eingegangen.

Im dritten Teil des Kapitels wird versucht, einen Bezug zwischen den CIM Associations und dem Topology Service der Object Management Group (OMG) herzustellen. Es wird gezeigt, wie sich CIM Associations auf den Topology Service abbilden lassen, was insbesondere für die Konstruktion von Gateways zu anderen Managementarchitekturen relevant ist.

4.1 Umwandlung von MOF in `.mo`-Dateien

Im folgenden soll demonstriert werden, wie aus den MOF-Dateien der CIM Schemata mit Hilfe spezieller Skripten `.mo`-Dateien generiert werden können, die dann unter Verwendung des mit JMAPI mitgelieferten Managed Object Compilers `Moco` zur Erzeugung entsprechender JMAPI Managed Object Klassen dienen können.

Ausgangspunkt sei eine Datei in MOF Syntax, `file.mof`. Diese Datei beschreibt u.a. die Klassen, die als Basisklassen für JMAPI Managed Objects dienen sollen, und die im folgenden zu extrahieren und in eine vom JMAPI Managed Object Compiler akzeptierte Form zu transformieren sind. Neben diesen Klassen enthält die Datei allerdings auch Klassen, die *keine* Managed Objects modellieren, wie etwa die Associations zwischen Managed Objects. Associations werden in JMAPI über spezielle Association-Objekte verwaltet. Die Klasse `ManagedObjectImpl` stellt Methoden zur Verwaltung aller mit einem gegebenen Managed Object assoziierten MOs zur Verfügung. Aus diesem Grund kann in der weiteren Vorgehensweise auf eine Behandlung von Associations verzichtet werden[1].

Die Umwandlung der MOF-Datei erfolgt in mehreren Schritten. Nach einer kurzen Auflistung dieser Schritte folgen hierzu nähere Einzelheiten:

1. Entfernen der Kommentare, Pragmas, Qualifier-Definitionen sowie fast aller Qualifier. Pragmas und Qualifier-Definitionen können für das `.mo`-Format außer acht gelassen werden. Von den Qualifiern werden nur der `Abstract`-Qualifier und der `Association`-Qualifier berücksichtigt. Der `Abstract`-Qualifier kennzeichnet abstrakte Klassen, die auch im Java-Code als solche gekennzeichnet werden müssen. Der `Association`-Qualifier wird zur Aussonderung von Associations benötigt.

2. Konvertierung in eine Datei mit Java-Syntax, die bereits alle Managed Object Klassen samt ihrer Attribute beinhaltet, aber noch keine `get`- und `set`-Methoden für die Attribute enthält.

3. Einfügen der entsprechenden `get`- und `set`-Methoden für die Attribute.

4. Aufspalten der Datei in eine Reihe von `.mo`-Dateien, von denen jede ein Managed Object enthält, und die dann vom JMAPI Managed Object Compiler bearbeitet werden können.

Abbildung 4.1 zeigt den Ablauf der Konvertierung im Überblick.

Für die Ausführung obiger Schritte wurden vier `Perl`-Skripten entwickelt (siehe Anhang A). `pass1.pl` nimmt die Datei `file.mof` als Eingabe und erzeugt die Datei `_file.mof` als Ausgabe. Diese wird in einem weiteren Schritt von `pass2.pl` weiterverarbeitet zur Datei `_file.mo`, die bereits Java-Syntax besitzt. Als nächstes fügt `pass3.pl` die `get`- und `set`-Methoden für die Attribute hinzu. Im letzten Schritt spaltet `break-it-up.pl` die Datei in eine Reihe von kleineren Dateien auf, von denen jede eine JMAPI Managed Object Klasse beschreibt.

[1] Die mitgelieferten JMAPI Base Managed Object Klassen stellen über die einfache Verwaltung von Associations hinaus Methoden zur Verfügung, die der Modellierung gemäße assoziierte Managed Objects auflisten. So besitzt beispielsweise die Klasse `PhysHostMOImpl` eine Methode `listLocalPrinters`, die eine Aufzählung aller mit einem physikalischen Host assoziierten Drucker liefert. Diese Methoden bieten dem Programmierer zwar eine Abstraktion vom Assoziationsmodell, beinhalten jedoch keine zusätzliche Funktionalität.

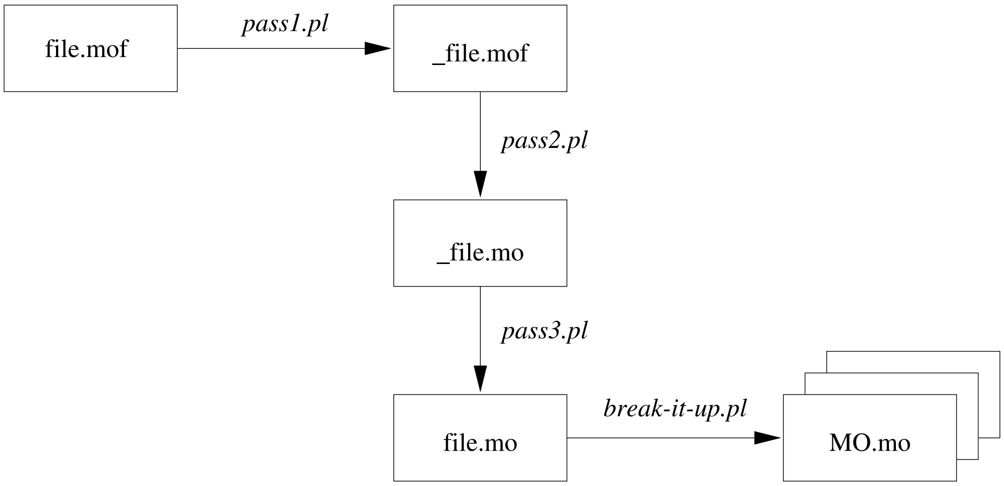


Abbildung 4.1: Schematischer Ablauf der Konvertierung von MOF nach `.mo`

Dies ist nötig, da alle Klassen in Java als `public` deklariert werden, und eine Java-Quellcode-Datei nur jeweils *eine* als `public` deklarierte Klasse enthalten darf.

Nachfolgend soll die Vorgehensweise anhand der Umwandlung der Datei `CIM_CoreSchema20.mof`, die das CIM Core Schema beschreibt, näher erläutert werden.

4.1.1 Schritt 1: Das Skript `pass1.pl`

Dieses Skript kopiert die Eingabedatei `CIM_CoreSchema20.mof` in die Ausgabedatei `_CIM_CoreSchema20.mof`, wobei allerdings folgende Zeilen nicht übernommen werden, da sie für die Umwandlung ins JMAPI-`.mo`-Format irrelevant sind:

- `pragma`-Anweisungen. Ihre Erkennung erfolgt durch das Zeichen „#“ am Zeilenanfang.

  Beispiel:

  ```
  #pragma schema ("CIM")
  ```

- Qualifier Definitionen.

  Beispiel:

  ```
  Qualifier Aggregation : boolean = false, Scope(association);
  ```

- Kommentarzeilen, die mit „`//`“ beginnen[2].

[2]Die Kommentarzeilen müßten nicht unbedingt entfernt werden, sondern könnten sogar direkt übernommen werden, da die Java-Syntax. in diesem Fall mit der MOF-Syntax über-

Beispiel:

```
// ==================================================================
//    ManagedSystemElement
// ==================================================================
```

Qualifier, die sich auf Klassen oder Attribute beziehen, stehen nach der MOF-Syntax jeweils *vor* den jeweiligen Klassen oder Attributen. Sie sind in eckigen Klammern eingeschlossen.
Beispiel:

```
    [Description (
    "The name of the organization responsible for producing the"
    "Physical Element.")]
string Manufacturer;
```

Die hierbei angegebenen Qualifier sind in der Regel nicht auf entsprechende Konstrukte in JMAPI abbildbar und werden deswegen übergangen. Ausnahmen bilden die Qualifier `Abstract` und `Association`. Auf ihre Behandlung wird unten eingegangen.

Obiges Beispiel würde demnach, unter Fortlassen der eckigen Klammern mitsamt des dazwischenstehenden `Description`-Qualifiers, zu

```
string Manufacturer;
```

Der Qualifier `Abstract` deutet, wie in der objektorientierten Modellierung üblich, an, daß eine Klasse nicht instantiiert werden kann. Diese Semantik sollte bei einer Übersetzung in Java-Klassen erhalten bleiben. In der Ausgabedatei wird eine derartige Klasse durch Voranstellen des Wortes `abstract` vor die jeweilige Klassendefinition gekennzeichnet.
Beispiel:

```
        [Association, Abstract, Description (
        "A generic association to establish dependency relationships"
        "between objects.") , Schema ("CIM") ]
class CIM_Dependency
{
  ...
}
```

wird zu[3]

```
abstract class CIM_Dependency
```

einstimmt. Da die erzeugten `.mo`-Dateien aber nur für die Weiterverarbeitung durch den `Moco` Compiler eingesetzt werden, sind Kommentare hier nicht notwendig. Für ein Studium der Klassenmodellierung sollten die MOF-Dateien, und nicht die `.mo`-Dateien herangezogen werden.

[3]Durch Voranstellen des Wortes `abstract` vor die Klassendefinition wird die MOF-Syntax durchbrochen. Die nachfolgende Umwandlung in das JMAPI-Format wird jedoch hierdurch etwas vereinfacht.

Ein weiterer Qualifier verdient bei der Umwandlung von MOF-Dateien in `.mo`-Dateien besondere Bedeutung: Durch den Identifikator `Association` werden Klassen gekennzeichnet, die Associations zwischen Managed Objects modellieren. Wie bereits erläutert, handelt es sich bei Associations nicht um Managed Objects, deshalb werden die entsprechenden Klassen samt zugehörigen Qualifiern vom Skript entfernt und nicht in die Ausgabedatei übernommen.

Enthält die MOF-Klassendefinition Methoden, so müssen diese im Java-Code als `public abstract` gekennzeichnet werden, da die Implementierung der Methoden fehlt, und die Methoden auch von anderen Java-Packages aus aufrufbar sein sollen. Das Skript `pass1.pl` fügt daher diese beiden Schlüsselworte vor Methodendeklarationen ein.
Beispiel:

```
uint8 start();
```

wird zu

```
public abstract uint8 start();
```

4.1.2 Schritt 2: Das Skript `pass2.pl`

Aufgabe dieses Skriptes ist das Erzeugen einer Datei, die Java Syntax besitzt und der Vererbungshierarchie der JMAPI Base Managed Object Classes genügt.

Aus jeder CIM-Klasse `CIM_`*SomeClass*, die ein Managed Object beschreibt, wird in JMAPI ein MO mit Remote Interface `CIM_`*SomeClass* und zugehöriger Implementierung `CIM_`*SomeClass*`Impl`.

Wurzel der JMAPI-MO-Klassenhierarchie bildet die Klasse `ManagedObjectImpl` mit Remote Interface `ManagedObject`. CIM-Klassen, die innerhalb von CIM keine Oberklassen besitzen, werden auf MOs abgebildet, die direkt von `ManagedObjectImpl` erben. Beispielsweise wird das aus `CIM_ManagedSystemElement` zu generierende JMAPI MO `CIM_ManagedSystemElementImpl` als Unterklasse von `ManagedObjectImpl` deklariert und erbt damit die Methoden für den JMAPI Transaktionsmechanismus, den Security Context und die Verwaltung von Associations.

Die restliche Vererbungshierarchie wird einfach auf JMAPI isomorph übertragen.

Enthält beispielsweise die Ausgabedatei des Skripts `pass1.pl` eine Zeile der Form

```
abstract class CIM_PhysicalElement:CIM_ManagedSystemElement
```

so muß diese transformiert werden in die Zeile

```
public abstract class CIM_PhysicalElementImpl
    extends CIM_ManagedSystemElementImpl
    implements CIM_PhysicalElement
```

Neben dem Aufbau einer Klassenhierarchie hat das Skript `pass2.pl` auch die Aufgabe, die CIM Datentypen auf entsprechende JMAPI Datentypen abzubilden.

Die Konvertierung der Datentypen ist in Tabelle 4.1 zusammengestellt. Da es in Java keine `unsigned` Datentypen gibt, ist es erforderlich, im Falle der Datentypen `uint8`, `uint16` und `uint32` jeweils auf den nächstgenaueren `signed`-Datentyp abzubilden. Eine Abbildung von `uint32` ist ferner nur unter Verwendung der Klasse `java.math.BigInteger` möglich.

MOF	Java
bool	boolean
string	String
uint8	short
sint8	byte
uint16	int
sint16	short
uint32	long
sint32	int
uint64	java.math.BigInteger
sint64	long
real32	float
real64	double
char16	char
datetime	String

Tabelle 4.1: Abbildung der Datentypen von MOF nach Java

4.1.3 Schritt 3: Das Skript `pass3.pl`

In diesem Schritt sollen für alle Attribute `get`- und `set`-Methoden definiert werden.

Beispiel:

Für das Attribut

```
String Description;
```

werden folgende zwei Methoden hinzugefügt:

```
public String getAttrDescription(Session session)
    throws AuthorizationException, RemoteException
{
        return (String) getPropertyByName(session, "Description");
}
public void setAttrDescription(Session session, String Description)
    throws NotInUpdateException, AuthorizationException,
           RemoteException
{
        setPropertyByName(session, "Description", Description);
}
```

4.1.4 Schritt 4: Das Skript `break-it-up.pl`

Als letztes muß nun für jede JMAPI Managed Object Klasse eine eigene Datei generiert werden. Das Skript `break-it-up.pl` erzeugt aus der Eingabedatei `CIM_CoreSchema20.mo` die `.mo`-Dateien, die zu den Managed Objects des CIM Core Modells korrespondieren.

Im Anhang A ist als Beispiel für eine der erzeugten Dateien die Datei `CIM_ServiceAccessPoint.mo` angegeben.

4.2 Verknüpfungen und Gruppenbildung von Managed Objects

Zur Modellierung von Verknüpfungen zwischen Managed Objects bietet JMAPI zwei verschiedene Modellierungskonstrukte: **References** und **Associations**.

Bei References handelt es sich um spezielle MO-Attribute, deren Werte Objektreferenzen sind. Associations hingegen werden über spezielle Association Objects realisiert. Im folgenden wird zunächst auf die Unterschiede von References und Associations hinsichtlich der Modellierung von Verknüpfungen zwischen MOs eingegangen. Im Anschluß daran wird geschildert, wie JMAPI Associations in der JMAPI-Referenzimplementierung [JMA97] realisiert werden. Daran anschließend werden typisierte und untypisierte JMAPI Associations erläutert, welche eine Besonderheit der Referenzimplementierung darstellen. Schließlich werden die zur Verfügung stehenden Methoden für die Verwaltung von JMAPI Associations vorgestellt.

Aus ablauf- oder aufbauorganisatorischen Gründen ist es oft nützlich, Gruppen von Managed Objects zu bilden. Als Beispiel sei etwa eine Reihe von Druckern genannt, die alle derselben Abrechnungsdomäne angehören. Die von JMAPI bereitgestellten Möglichkeiten zur Gruppenbildung von MOs werden ebenfalls in diesem Abschnitt erläutert.

4.2.1 Unterschiede zwischen Associations und References

References müssen als MO-Attribute bei der Definition der Managed-Object-Klasse angegeben werden. Wird das Objektmodell später erweitert, können ohne Veränderung der Managed-Object-Klasse keine References mehr hinzugefügt werden.

Alternativ zu References können Associations eingesetzt werden. JMAPI Associations stellen, ebenso wie References, unidirektionale Verknüpfungen zwischen Managed Objects her. Im Gegensatz zu References erlauben Associations die Verknüpfung von MO-Klassen auch in den Fällen, in denen dies nicht bereits bei der Definition der MO-Klassen durch Anlegen entsprechender Reference-Attribute vorgesehen war. Sie bieten demnach eine größere Flexibilität im Hinblick auf Erweiterungen des Objektmodells.

Ein weiterer Unterschied zwischen Associations und References besteht hinsichtlich der Semantik beim Löschen eines der beteiligten Managed Objects. Soll ein Managed Object gelöscht werden, auf das von Seiten eines anderen Managed Objects mittels einer *Reference* verwiesen wird, so wird das Löschen vom System nicht gestattet, um die referentielle Integrität nicht zu verletzen. Das Löschen eines der beiden an einer *Association* beteiligten Managed Objects ist hingegen immer möglich und zieht das Löschen der Association nach sich. References können z.B. zur Modellierung von Abhängigkeiten genutzt werden. Hängt beispielsweise ein MO `A` von der Existenz eines anderen MOs `B` ab, so sollte es nicht möglich sein, `B` zu löschen ohne vorher auch `A` gelöscht zu haben. Wird diese Abhängigkeit mittels einer Reference von `A` nach `B` modelliert, so wird dies durch die Überwachung der referentiellen Integrität automatisch gewährleistet. Im Vergleich zu References kann durch Associations eine „lose Kopplung" der beiden beteiligten Managed Objects modelliert werden, bei der z. B. jederzeit eines der beiden MOs gelöscht werden kann.

In [JMA97] werden Associations und References etwas anders modelliert als im Meta Schema von CIM. CIM sieht References lediglich innerhalb von Associations vor, während JMAPI References als Bestandteile von Managed Objects kennt. Ferner sind bei JMAPI nur binäre Associations möglich, während CIM es auch ermöglicht, mehr als zwei Objekte über Associations zu verknüpfen. Es ist zu vermuten, daß mit der Angleichung der JMAPI MO-Basisklassen an den neuen CIM-Standard dieser Unterschied verschwinden wird und es keine References als Attribute von MOs mehr geben wird.

4.2.2 Realisierung von Associations in der JMAPI-Referenzimplementierung

Wie bereits erwähnt, werden Associations mittels spezieller Objekte, der Association Objects, realisiert. Ein Association Object verwaltet die beiden an der Association beteiligten MOs. Association Objects sind Instanzen von Association-Klassen. Im Falle von [JMA97] implementieren JMAPI-Association-Klassen das Interface `sunw.admin.arm.manager.Association`. Dieses besteht aus den folgenden vier Methoden.

```
public abstract String getName();
public abstract ManagedObject getSource();
public abstract ManagedObject getTarget();
public abstract String getAssociationClass();
```

Hierüber ist es möglich, die vier für eine JMAPI Association wichtigen Größen abzufragen: *Source* und *Target* sind die beiden durch die Association verknüpften Managed Objects. Ferner ist jedes Association Object durch einen Namen und die JMAPI-Association-Klasse, die es instantiiert, charakterisiert. Die Association-Klasse ist im Zusammenhang mit typsisierten Associations, auf die im Abschnitt 4.2.3 eingegangen wird, von Bedeutung.

JMAPI unterscheidet zwei grundlegende Klassen, die das `Association`-Interface implementieren. Auf Seiten des Managed Object Servers ist dies die Klasse `ServerAssociation`, auf Seiten des Clients die Klasse `ClientAssociation`. Erstere kann nur auf dem Server verwendet werden. Methoden, die einem Client eine Aufzählung aller Associations liefern, an denen ein gegebenes MO beteiligt ist, müssen daher `ServerAssociations` erst in `ClientAssociations` konvertieren. `ClientAssociations` sind serialisierbar und werden übers Netz an den Client übertragen. Die Unterscheidung zwischen `ServerAssociations` und `ClientAssociations` ist implementierungstechnisch bedingt. `ServerAssociations` stellen eine leichtgewichtigere und effizientere Implementierung des `Association`-Interface dar.

4.2.3 Typisierte und untypisierte JMAPI Associations

In [JMA97] wird zwischen typisierten und untypisierten Associations unterschieden. Untypisierte Associations sind Instanzen der Klasse `sunw.admin.arm.manager.ServerAssociation` bzw. `sunw.admin.arm.manager.ClientAssociation`. Typisierte Associations sind Instanzen benutzerdefinierter Unterklassen von `ServerAssociation` bzw. `ClientAssociation`.

Im folgenden wird zunächst gezeigt, wie sich CIM Associations mit Hilfe von untypisierten JMAPI Associations darstellen lassen[4]. Im Anschluß daran wird der Sinn von typisierten JMAPI Associations erläutert.

Die Abbildung von CIM Associations auf JMAPI Associations kann alleine mit Hilfe des Namens-Attributs (und der Source- und Target-References) erfolgen. Dem Namensattribut ist lediglich der Typname der entsprechenden CIM Association zuzuweisen. Als Beispiel werde die CIM Association `CIM_Dependency` des Core Models betrachtet. Die zugehörige Definition in der MOF-Datei lautet:

```
        [Association, Abstract, Description (
        "A generic association to establish dependency"
        "relationships between objects.") , Schema ("CIM") ]
class CIM_Dependency
{
        [Description (
        "Antecedent represents the independent object"
        "in this association."), Multiplicity ("MV") ]
    CIM_ManagedSystemElement REF Antecedent;
        [Description (
        "Dependent represents the object dependent on"
        "the Antecedent.", "Multiplicity", "MV") ]
    CIM_ManagedSystemElement REF Dependent;
};
```

Wenn die Managed Object Klasse `CIM_ManagedSystemElement` als Unterklasse von `ManagedObjectImpl` bereits definiert und in die Datenbank importiert ist, kann zwischen zwei Instanzen dieser Klasse eine JMAPI Association mit Namen „`CIM_ManagedSystemElement`“ erzeugt werden. Der Name repräsentiert in diesem Fall die CIM-Klasse der Association.

In diesem Zusammenhang stellt sich nun die Frage, warum JMAPI typisierte Associations bietet. Die Begründung hierfür liegt in der Möglichkeit,

[4] Im Gegensatz zu JMAPI Associations können CIM Associations mehr als zwei Objekte verknüpfen. Die folgende Konstruktion trifft nur auf binäre CIM Associations zu. Höherwertige Associations, die mehr als zwei Objekte verknüpfen, müssen aus binären Associations aufgebaut werden.

daß verschiedene Clients auf derselben Menge von Managed Objects Associations mit demselben Namen anlegen können, ohne hierbei in Konflikte zu geraten. Zu diesem Zweck müssen nur Unterklassen von `ServerAssociation` und `ClientAssociation` gebildet werden, deren Konstruktoren *package visibility* besitzen. Associations dieser Unterklassen können dann nur innerhalb des entsprechenden Java-Packages aufgerufen werden und sind vor äußeren Eingriffen geschützt. Die Kombination von Association-Typ und Association-Namen ist jeweils eindeutig.

4.2.4 Methoden zur Verwaltung von JMAPI Associations

Das Anlegen von Associations erfolgt über Methodenaufrufe der Klasse `ManagedObjectImpl`. Mit dem Aufruf `mo1.addAssociation(session, "name", mo2)` wird eine Association mit Namen „name" zwischen `mo1` und `mo2` angelegt, wobei `mo1` das Source- und `mo2` das Target-Objekt ist. Der Aufruf `mo1.addAssociations(session, "name", mo2)`[5] legt zwei entsprechende Associations an, wobei jedes MO einmal als Source- und einmal als Target-Objekt fungiert.

Bezüglich der Abfragemöglichkeit nach bestehenden Associations und dem Löschen bestehender Associations gibt es Methodenaufrufe zum Auflisten bzw. Löschen aller Associations.

- an denen ein gegebenes MO beteiligt ist. (`enumerateAssociations`, `removeAssociations`)[6].
- an denen ein gegebenes MO beteiligt ist und die einen bestimmten Namen haben (`enumerateAssociations`, `removeAssociations`).
- die ein gegebenes MO als Source-Objekt haben (`enumerateAssociationsFrom, removeAssociationsFrom`).
- die ein gegebenes MO als Source-Objekt und die einen bestimmten Namen haben (`enumerateAssociationsFrom, removeAssociationsFrom`).
- die ein gegebenes MO als Target-Objekt haben (`enumerateAssociationsTo, removeAssociationsTo`).
- die ein gegebenes MO als Target-Objekt und die einen bestimmten Namen haben (`enumerateAssociationsTo, removeAssociationsTo`).
- die zwischen einem gegebenen MO und einem anderen MO bestehen (`enumerateAssociationsWith, removeAssociationsWith`).
- die zwischen einem gegebenen MO und einem anderen MO bestehen und die einen bestimmten Namen haben (`enumerateAssociationsWith, removeAssociationsWith`).

[5] man beachte den Plural

[6] Die Methoden sind, wie in nächsten Punkt deutlich wird, alle mehrfach überladen.

Für typisierte Associations existieren analoge Aufrufe, z. B. `enumerateTypedAssociations`, mit einem zusätzlichen Parameter, der den Typ spezifiziert.

4.2.5 Gruppieren von Managed Objects

Neben dem Anlegen von Associations zwischen MOs bietet JMAPI die Möglichkeit, MOs zu Gruppen zusammenzufassen. JMAPI-Gruppen können zur Realisierung eines Domänenkonzepts eingesetzt werden (vgl. 8.3). Es werden zwei Arten von Gruppen unterschieden, **Explicit Groups** und **Query Based Groups**. Mitglieder einer Gruppe sind stets einzelne MOs. Ein MO kann mehreren Gruppen angehören.

Die Gruppenmitglieder einer Explicit Group müssen vom Benutzer explizit angegeben werden. Sie können entweder einzeln hinzugefügt werden, oder es können die Mitglieder der gegebenen Gruppe um die Mitglieder einer anderen Gruppe erweitert werden.

Bei einer Query Based Group werden die Gruppenmitglieder dynamisch von JMAPI verwaltet. Alle Gruppenmitglieder einer Query Based Group müssen derselben MO-Klasse angehören. Bei einer Anfrage nach den Mitgliedern der Gruppe wird ein beim Anlegen der Gruppe bestimmter Anfrageausdruck ausgewertet. Hierbei handelt es sich um einen Ausdruck über der Menge der MO-Attribute. Es können für ein oder mehrere Attribute der MO-Klasse, welcher die Gruppenmitglieder angehören sollen, bestimmte Werte spezifiziert werden. Der Query Based Group gehören dann jeweils alle MOs der betreffenden Klasse an, deren Attributwerte zum Zeitpunkt der Anfrage die spezifizierten Werte besitzen.

Groups werden von speziellen `Group` Managed Objects verwaltet. Über einen auf dem Management Server vorhandenen `GroupController` lassen sich neue Gruppen anlegen, oder bereits vorhandene Gruppen für die weitere Bearbeitung auswählen.

4.3 Vergleich mit dem OMG Topology Service

In diesem Abschnitt soll zunächst der Topology Service der OMG vorgestellt werden. Im Anschluß daran wird gezeigt, wie sich CIM Associations auf den OMG Topology Service abbilden lassen.

4.3.1 Der OMG Topology Service

Der Topology Service der Object Management Group (OMG) [Hew97] stellt einen allgemeinen Dienst für die Verwaltung von Beziehungen (Associations) zwischen *Entities*[7] bereit.

Durch Angabe von Regeln *(Rules)* wird unter allen möglichen Associations eine Teilmenge von gültigen Associations ausgezeichnet. Die Einhaltung dieser

[7] Im Rahmen des Topology Service wird nicht von Objekten, sondern von Entities gesprochen, um der Tatsache Ausdruck zu verleihen, daß es sich nicht um CORBA-Objekte handeln muß.

Regeln wird dann vom Topology Service überwacht. Zur Abfrage der abgespeicherten Topologieinformation steht eine spezielle Anfragesprache, die *Topology Query Language* (TQL), zur Verfügung.

Im folgenden soll kurz auf nähere Einzelheiten eingegangen werden.

Entities und topologischer Typ

Der Topology Service befaßt sich mit der Verwaltung von Associations zwischen *Entities*. Jede Topological Entity besteht aus einem Identifikator und einer optionalen Objektreferenz. Diese verweist, falls vorhanden, auf das zugehörige Objekt, dessen topologischer Zustand vom Topology Service verwaltet wird, und welches den nicht-topologischen Zustand beinhaltet. Durch Zuweisung eines topologischen Typs an eine Entity wird diese zu einer *Topologically Managed Entity*. Der Topology Service stellt über den *Topology Meta Data Manager* (siehe Abbildung 4.2) eine Schnittstelle zum Anlegen und Verwalten von topologischen Typen bereit. Neue topologische Typen können bereits vorhandene Typen spezialisieren. Sie bilden somit eine Typhierarchie. Ein topologischer Typ kann mehrere andere Typen spezialisieren.

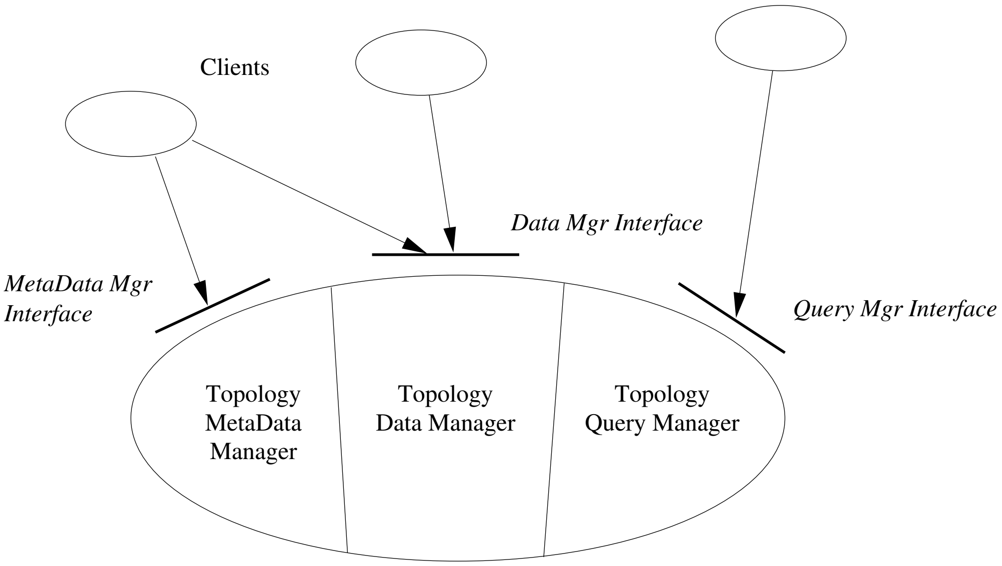


Abbildung 4.2: Architektur des Topology Service

Associations und topologische Regeln

Associations im Rahmen des Topology Service sind binäre Verknüpfungen zwischen Entities. Topologische Regeln *(Topology Rules)* legen fest, zwischen welchen topologischen Entities Associations angelegt werden können. Maßgeblich hierfür sind die oben erwähnten topologischen Typen der beiden beteiligten Entities. Neben den topologischen Typen werden auch die Kardinalitäten spezifiziert, die angeben, wieviele Associations einer bestimmten Art eine Entity

eines gegebenen topologischen Typs eingehen kann. Zudem wird jeder der beiden Entities eine Rolle in Form eines Strings zugewiesen.
Eine topologische Regel ist somit durch folgendes 8-Tupel gekennzeichnet [Hew97]:

$$(Type_1, Role_1, minCardinality_1, maxCardinality_1,\\ Type_2, Role_2, minCardinality_2, maxCardinality_2)$$

Aggregate Entities

Topologically Managed Entities können zu *Aggregate Entities* zusammengefaßt werden. Hierbei gilt die Einschränkung, daß innerhalb einer Aggregate Entity höchstens eine Instanz eines Topological Types enthalten sein kann. Aggregate Entities werden durch ein beliebiges ihrer Mitglieder identifiziert, es existieren keine Identifikatoren für Aggregate Entities. Durch die innerhalb einer Aggregate Entity enthaltenen einzelnen Entities sollen unterschiedliche Sichtweisen desselben Objekts modelliert werden.

Enforcer

Die durch die Angabe von Regeln definierbare Semantik ist oft nicht ausreichend, deswegen sieht der Topology Service vor, durch Angabe von *Enforcern* zusätzliche Regeln für gültige Associations zu definieren.

Ein *Enforcer* enthält eine `validate` Methode, die als Eingabeparameter die von einer Änderung betroffene Topological Entity verlangt. Als Ergebnis der Ausführung ergibt sich ein Boole'scher Wert.

Bei der Registrierung von Enforcern können spezielle *Invocation Conditions* angegeben werden. Diese legen fest, unter welchen Bedingungen eine Überprüfung durch den betreffenden Enforcer erfolgen soll. Hierdurch sollen unnötige Aufrufe vermieden werden. Beispiele für Invocation Conditions sind die Änderung des topologischen Zustandes einer Entity oder einer assoziierten Entity.

Topological Query Language

Mit Hilfe der Topology Query Language sind in komfortabler Weise Anfragen mittels Angabe von *Aggregate Expressions* und *Association Patterns* möglich, die ausgehend von einer gegebenen Entity einen Teilgraphen der Gesamttopologie liefern, welcher gewissen spezifizierten Forderungen genügt.

Notifications

Notifications dienen dazu, Zustandsänderungen einzelner Entity-Instanzen oder der Instanzen eines speziellen topologischen Typs anzuzeigen. Zum Erzeugen von Notifications dient der OMG Notification Service [BEA98].

Schnittstellen

Der Topology Service hat drei Schnittstellen (vgl. Abbildung 4.2): Über das **Meta Data Manager** Interface werden Regeln, Enforcer und topologische Typen definiert. Das **Data Manager** Interface dient dem Management einzelner Entities sowie der zwischen ihnen bestehenden Associations. Das **Query Manager** Interface schließlich stellt die Schnittstelle für Anfragen mittels TQL dar.

4.3.2 Abbildung der CIM Associations auf den OMG Topology Service

In diesem Abschnitt soll gezeigt werden, wie sich die CIM Associations auf den Topology Service abbilden lassen.

Es sind folgende zwei Schritte durchzuführen:

- Anlegen eines neuen topologischen Typs für jede CIM-Klasse, die keine Association und keine Indication darstellt.
- Für jede CIM Association ist nun eine Regel zu definieren, die die topologischen Typen enthält, die den referenzierten CIM-Klassen entsprechen. Die minimalen und maximalen Kardinalitäten sind, falls in der Definition der Association nicht anders angegeben, mit 0 und ∞ festzulegen.

Beispiel: Es soll die CIM Association `CIM_HostedService` des Core Models abgebildet werden:

Die MOF Definition lautet:

```
        [Association, Description(
        "An association between a Service and the System on which the"
        "functionality resides. The cardinality of this association"
        "is 1-to-many. A System may host many Services ...")]
class CIM_HostedService:CIM_Dependency
{
        [Override ("CIM_Dependency:Antecedent") , Description (
        "The hosting System") , Multiplicity ("SV") ]
    CIM_System REF Antecedent;
        [Override ("CIM_Dependency:Dependent") , Description (
        "The Service hosted on the System") , Multiplicity ("MV") ]
    CIM_Service REF Dependent;
};
```

Nach Definition der topologischen Typen `CIM_System` und `CIM_Service` kann eine Regel definiert werden.

$$R = (CIM\_System, Antecedent, 1, 1, CIM\_Service, Dependent, 0, \infty)$$

Diese Regel trägt der $1 : n$ Kardinalität in der CIM-Definition Rechnung.

Bemerkungen

Der Topology Service sieht die Benutzung des Notification Service für Benachrichtigungen im Fall von Topologieänderugen vor. JMAPI kennt keinen vergleichbaren Mechanismus. Das Anlegen oder Löschen von JMAPI Associations hat keine Erzeugung von Notifications zur Folge.

Die Abfragemöglichkeiten bzgl. der vorhandenen Topologie sind bei JMAPI sehr eingeschränkt, da nur jeweils Associations zum nächsten Nachbarn abgefragt werden können und nicht ganze Teilgraphen als Ergebnis einer Anfrage geliefert werden, wie dies bei Verwendung von TQL möglich ist.

4.4 Zusammenfassung

Das JMAPI-Informationsmodell stützt sich auf das CIM der DMTF ab. Zum jetzigen Zeitpunkt liegt jedoch noch keine Übereinstimmung der zu JMAPI gehörenden MO-Basisklassen und der CIM-Klassen vor, diese ist allerdings für die Zukunft angekündigt. Bereits mit der aktuell vorliegenden Version ist es jedoch relativ einfach mit Hilfe von in diesem Kapitel vorgestellten Skripten möglich, auf der Basis von vorliegenden MOF-Definitionen, Dateien zu erzeugen, die unter Zuhilfenahme des JMAPI Managed Object Compilers eine CIM-konforme Basisklassenhierarchie generieren, die dann im Rahmen von JMAPI genutzt werden kann.

Der Topology Service der OMG stellt einen sehr allgemeinen und mächtigen Dienst für die Verwaltung der Beziehung zwischen Entities dar. Im voranstehenden Kapitel wurde gezeigt, wie sich CIM Associations auf den Topology Service abbilden lassen. Die von JMAPI zur Verfügung gestellten Methoden zur Abfragbarkeit von Topologie-Informationen auf Basis der CIM Associations sind im Gegensatz zu den Möglichkeiten der TQL des Topology Service sehr eingeschränkt.

Kapitel 5

JMAPI Events und der OMG Notification Service

Neben gewöhnlichen Überwachungs- und Steuerungsaktionen, welche von den Clients initiiert und in synchroner Weise mittels RMI ausgeführt werden, spielen JMAPI Events eine wichtige Rolle. JMAPI Events repräsentieren Ereignisse, von deren Eintreffen Clients oder Managed Objects mittels Event Messages in asynchroner Weise benachrichtigt werden können. Beispiele hierfür sind etwa Zustandsänderungen von Managed Objects, deren Status gerade von einem Client überwacht wird oder die Überschreitung des Schwellwerts einer Meßgröße auf einem Agenten. Der Event-Mechanismus entbindet Komponenten, die am Eintreffen dieser Ereignisse interessiert sind, von der Notwendigkeit, durch fortwährendes Abfragen *(Polling)* den Zustand einer zu überwachenden Ressource zu überprüfen und gegebenenfalls entsprechende Maßnahmen einzuleiten. Events tragen somit zur Reduzierung der Netzlast bei.

Im folgenden wird zunächst das JMAPI Event Model erläutert. Im Anschluß daran wird ein Vergleich mit dem Notification Service der Object Management Group gezogen, was wiederum für die Konstruktion von Gateways von Bedeutung ist.

5.1 Das JMAPI Event Model

Das JMAPI Event Model kennt drei verschiedene Rollen: **Event Producer** erzeugen Events und senden sie in Form von Event Messages an **Event Dispatcher**, welche die Verteilung an die interessierten **Event Consumer** übernehmen. Events können sowohl vom JMAPI System selbst ausgelöst (wie z. B. ein SHUTDOWN Event des Managed Object Servers), als auch durch Methoden von MOs und Agenten, generiert werden. Vom JMAPI System erzeugte Events werden auch als *Notifications* bezeichnet.

Ein Charakteristikum dieser Kommunikation ist es, daß mehrere Komponenten an einem Event interessiert sein können. Die Kommunikation ist also von der Form $1 : n$, im Gegensatz zur Remote Method Invocation, wo jeweils *ein* Client mit *einem* Server in synchroner Weise kommuniziert.

Die Entkopplung der Kommunikation von Event Producern und Event Consumern erfolgt mittels der Event Dispatcher. Jeder von einem Producer erzeugte Event wird an einen Event Dispatcher gesandt. Andererseits muß sich jeder Event Consumer, der Events empfangen möchte, bei mindestens einem Event Dispatcher registrieren. Trifft ein Event bei einem Event Dispatcher ein, so übernimmt dieser die Benachrichtigung aller bei ihm registrierten Event Consumer. Der Event Producer braucht sich somit nicht um die Verwaltung aller interessierten Event Consumer zu kümmern, da diese Aufgabe vom Event Dispatcher übernommen wird.

Um nicht eine Unzahl von Nachrichten zu erzeugen, an denen die Consumer u.U. gar nicht interessiert sind und die von diesen dann erst aussortiert werden müßten, bedient sich JMAPI zum einen der Strukturierung von Events mittels **Event Trees**, zum anderen kommen spezielle **Filter** zum Einsatz.

Jedem JMAPI Event wird bei seiner Erzeugung ein Knoten in einem konzeptionellen Baum, dem Event Tree, zugewiesen, der alle Events einer Baumstruktur eingliedert. Ein Beispiel eines solchen Event Trees ist in Abbildung 5.1 dargestellt.

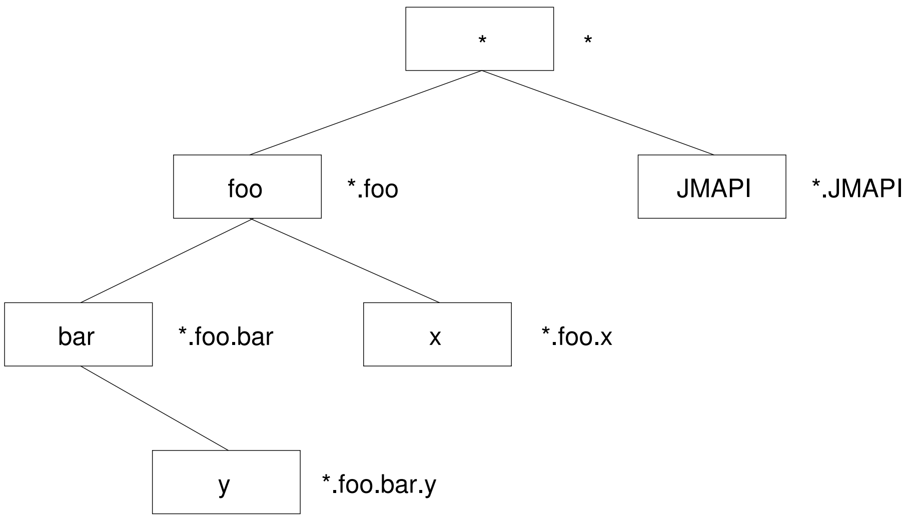


Abbildung 5.1: Beispiel für einen Event Tree

Jeder Knoten in diesem Baum wird mit einer Zeichenkette bezeichnet. Der Wurzel des Baums ist der String `*` zugeordnet. Jeder andere Knoten erhält einen beliebigen String zugewiesen, der allerdings für alle Nachfolgeknoten eines gegebenen Knotens eindeutig sein muß. Die eindeutige Identifikation eines Knotens im gesamten Baum kann nun durch die Angabe des Pfades von der Wurzel zum Knoten erfolgen, wobei die den Pfadknoten zugeordneten Strings konkateniert und durch „.“ voneinander getrennt werden. Der Baum in Abbildung 5.1 besteht demnach aus den Knoten `*`, `*.foo`, `*.JMAPI`, `*.foo.bar`, `*.foo.x` und `*.foo.bar.y`.

Der Event Tree gestattet die Strukturierung benutzerdefinierter Events auf der einen Seite und der vom JMAPI System automatisch generierten Notifications auf der anderen Seite. Der Unterbaum mit Wurzelknoten `*.foo` ist ein

Beispiel für einen Teilbaum, der der Strukturierung benutzerdefinierter Ereignisse dienen soll. Die Bezeichnung der Knoten ist frei wählbar.

Die von JMAPI erzeugten Notifications werden im Unterbaum `*.JMAPI` verwaltet (vgl. Abbildung 5.2), und sind dort vorgegebenen standardisierten Knoten zugeordnet.

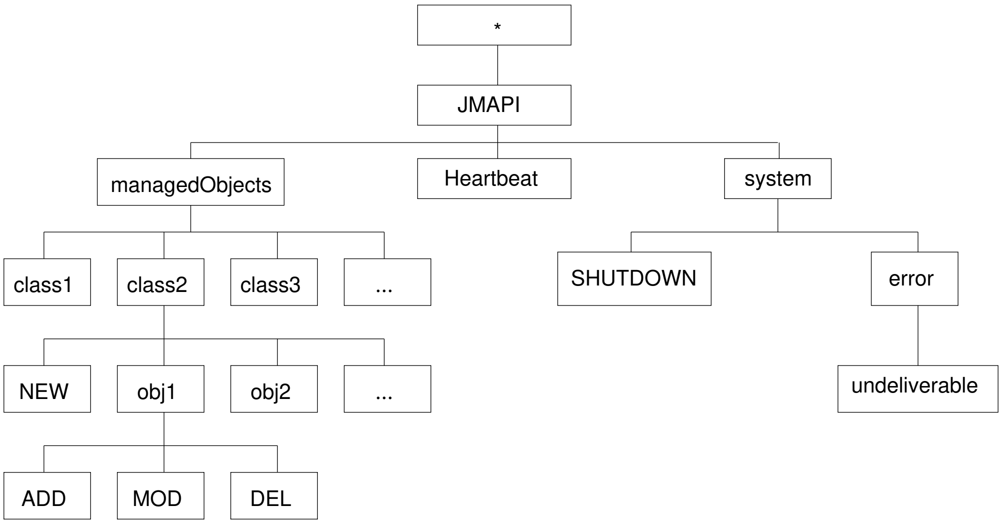


Abbildung 5.2: Unterbaum für JMAPI Notifications

Notifications werden u.a. beim Eintreten folgender Ereignisse erzeugt:

- Eine MO Instanz wurde neu angelegt.
- Eine MO Instanz wurde gelöscht.
- Die Attributwerte eines MO wurden verändert.
- Der Managed Object Server wird heruntergefahren (SHUTDOWN).
- Ein Event konnte dem Event Dispatcher nicht zugestellt werden, weil die Anzahl der *Retry Attempts* überschritten wurde (siehe hierzu die Erläuterungen weiter unten).

Ferner gibt es einen *Heartbeat*, der standardmäßig alle 60 Sekunden vom Managed Object Server erzeugt wird. Er kann z. B. eingesetzt werden, um festzustellen, ob der Managed Object Server noch läuft.

Als Beispiel für einen Strukturierungsknoten für JMAPI Notifications sei der String `*.JMAPI.system.SHUTDOWN` aufgeführt, der den Knoten beschreibt, dem Notifications zugeordnet sind, die einen SHUTDOWN des Servers signalisieren.

In gleicher Weise wie jeder JMAPI Event einem Knoten im Event Tree zugeordnet wird, geben Event Consumer zum Zeitpunkt der Registrierung bei einem Event Dispatcher einen Knoten im Baum an. Dies deutet ein Interesse an allen Events an, die beim entsprechenden Event Dispatcher eintreffen und dem betreffenden Knoten *oder einem direkten oder indirekten Nachfahren des Knotens im Baum* zugeordnet sind. Registriert sich ein Consumer beispielsweise am Knoten `*.JMAPI`, so bekundet er damit auch sein Interesse an `*.JMAPI.system.SHUTDOWN` Ereignissen.

Kann durch die baumartige Strukturierung der Events im Zusammenhang mit der Registrierung an bestimmten Knoten bereits eine Reihe von uninteressanten Events ausgesondert werden, bietet der Einsatz von Filtern eine zusätzliche Möglichkeit, den Netzverkehr zu reduzieren.

Neben der Angabe eines Knotens im Event Tree gibt jeder Event Consumer auch ein `Filter`-Objekt an. Hierbei handelt es sich um ein Objekt mit einer `evaluate`-Methode, die nach der Auswertung des Events einen Wahrheitswert zurückliefert. Aufgrund dieses Ergebnisses wird dann der Event entweder als uninteressant verworfen oder aber weiterverarbeitet.

Für die Weiterverarbeitung sieht JMAPI zwei verschiedene Möglichkeiten vor. Zum einen kann ein Consumer bei der Registrierung ein `Action`-Objekt angeben. Dieses besitzt eine `performAction`-Methode, die im Falle der Evaluation des Filters zum Wert `true` zur Ausführung gelangt. Alternativ kann der Consumer bei der Registrierung auch einen Host und eine Portnummer angeben. In diesem Fall würde der Event dann an den entsprechenden Host weitergesandt werden.

Beim Eintreffen eines Events bei einem Event Dispatcher wird zunächst derjenige Knoten des von diesem Dispatcher verwalteten Event Trees behandelt, dem der eingetroffene Event zugeordnet ist. Die Filter der dort registrierten Consumer werden ausgewertet und der Event wird entsprechend weiterverarbeitet. Als nächster Schritt wird in analoger Weise mit dem Vaterknoten im Baum verfahren. Diese Vorgehensweise setzt sich solange fort, bis die Wurzel des Baumes erreicht ist.

Auf dem Managed Object Server läuft immer ein Event Dispatcher. Zusätzlich können auf beliebig vielen Appliances weitere Event Dispatcher instantiiert werden.

Jedem Event Dispatcher ist ein `EventMO` Managed Object zugeordnet[1], welches für die Steuerung des Event Dispatchers verantwortlich ist (siehe Abbildung 5.3). Über das `EventMO` kann der entsprechende Event Dispatcher gestartet oder angehalten werden, oder es können sich Event Consumer beim zugehörigen Event Dispatcher registrieren. Ferner gibt es einen **Event Dispatcher Controller** mit zugehörigem Managed Object `EvdController`, welcher die Verwaltung aller einzelnen Event Dispatcher übernimmt. Ein Client kann mit Hilfe des Event Dispatcher Controllers folgende Operationen ausführen:

- Auflisten aller vorhandenen Event Dispatcher
- Anlegen von neuen Dispatchern
- Ermitteln von RMI Client Stubs bereits existierender Dispatcher
- Starten aller Event Dispatcher in der Management Domäne

[1] Man beachte, daß das `EventMO` einen *Event Dispatcher* repräsentiert, und keinen Event. Events haben keine zugehörigen MOs.

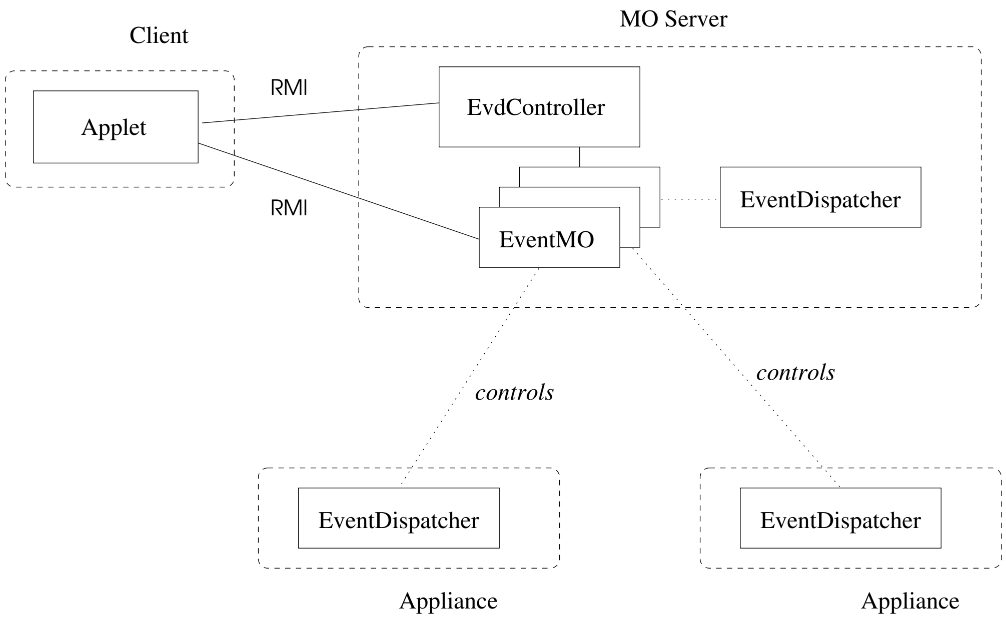


Abbildung 5.3: Event Dispatcher Hierarchie

- Anhalten aller Dispatcher in der Management Domäne

Bei der Erzeugung von JMAPI Events können folgende Parameter spezifiziert werden:

- der String `BranchName`, der den Knoten im Event Tree, dem der Event zugeordnet ist, identifiziert.

- die eigentlichen Daten des Events in Form eines `VecList` Objekts. Hierbei handelt es sich um eine Unterklasse von `java.util.Vector`, die die Möglichkeit bietet, über Schlüssel in Form eines `String`s auf Objekte zuzugreifen. In diesem Sinne bestehen also die Daten eines Events aus *key–value* Paaren.

- ein Integer Wert `RetryCount` mit folgender Bedeutung: Wenn der Event dem Event Dispatcher aus irgendeinem Grund nicht zugestellt werden kann, so wird entsprechend dem Wert von `RetryCount` mehrfach versucht, den Event erneut zu übermitteln.

- ein Integer Wert `RetrySleep`, der die Anzahl von Sekunden angibt, die im Falle einer fehlgeschlagenen Zustellung an den Event Dispatcher zwischen erneuten Zustellversuchen gewartet wird.

Events werden als Event Messages verschickt. Abbildung 5.4 zeigt den prinzipiellen Aufbau einer JMAPI Event Message.

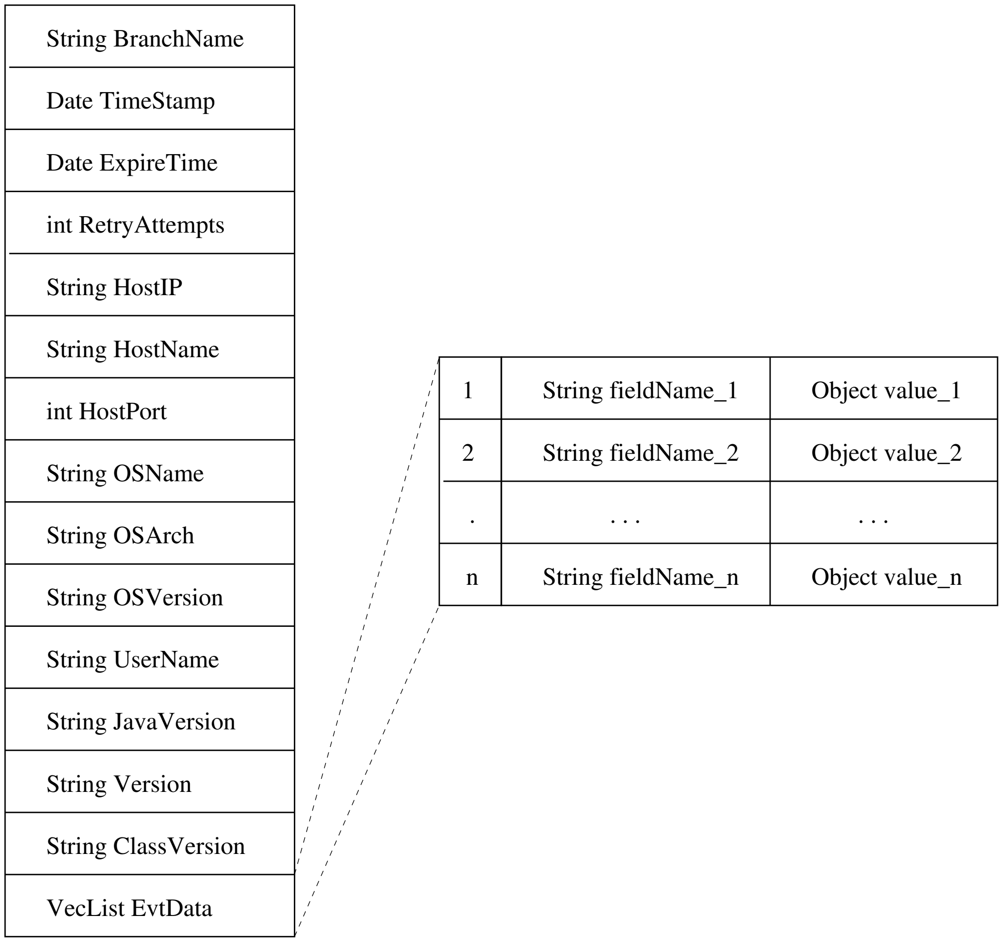


Abbildung 5.4: Prinzipieller Aufbau einer JMAPI Event Message

5.2 Überblick über den OMG Notification Service

Da in diesem Kapitel ein Vergleich des JMAPI Event Services mit dem OMG Notification Service vorgenommen werden soll, wird zunächst auf letzteren kurz eingegangen.

Der OMG Notification Service [BEA98] erweitert den CORBA Event Service [OMG97] um folgende Möglichkeiten:

- Unterstützung von strukturierten Events
- Definition von Filtern, die festlegen, an welchen Events ein Client interessiert ist.
- Event Supplier können feststellen, an welchen Event-Typen Clients eines Channels interessiert sind, um nur solche Events weiterzuleiten.
- Event Consumer können ermitteln, welche Event-Typen von einem Supplier angeboten werden, um sich so für neue Event-Typen zu registrieren.

- Möglichkeit der Konfiguration verschiedener Quality-of-Service Parameter (beispielsweise *Delivery Guarantee*, *Event Aging* und *Event Prioritization*).
- Ein Event Type Repository kann angelegt werden, welches Information über die Struktur von Events eines Event Channels zugänglich macht.

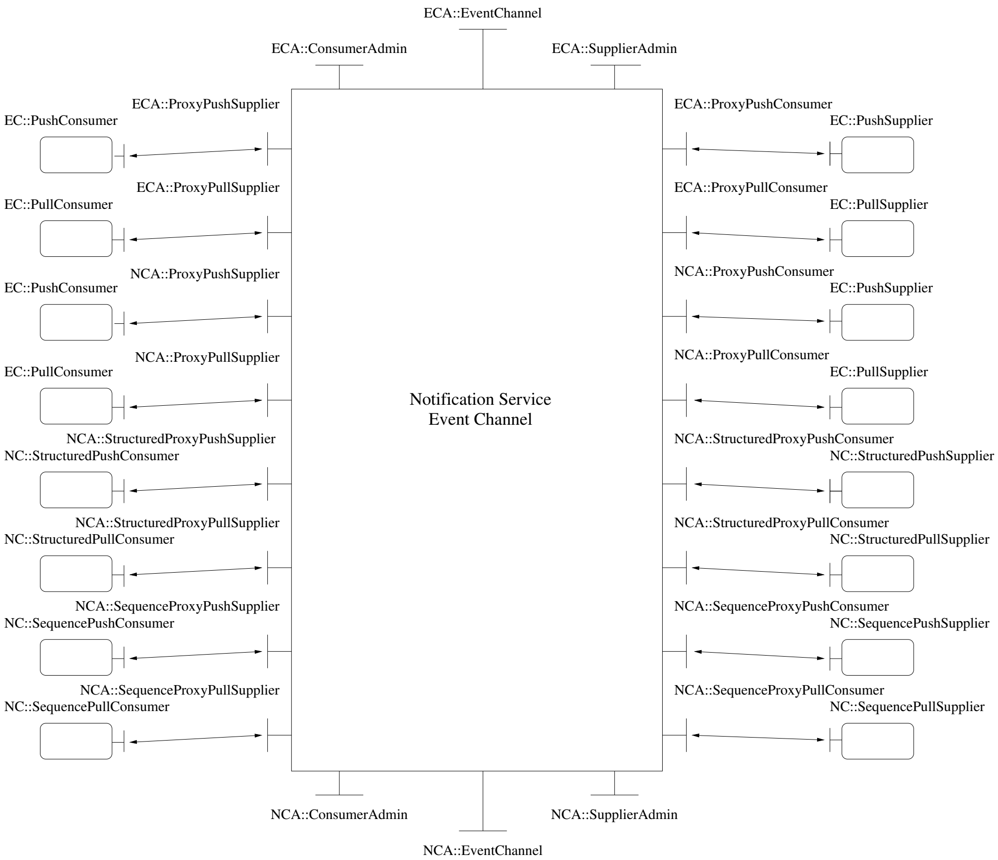


Abbildung 5.5: Architektur des Notification Service

Abbildung 5.5 stellt das Architekturmodell des Notification Service dar.

Events werden von *Suppliers* generiert und an den *Event Channel* weitergeleitet. Dieser nimmt Events über eines der *Proxy Consumer* Interfaces entgegen. Auf der anderen Seite erhalten *Consumer* von *Proxy Suppliern* des Channels die Events.

Sowohl auf der Supplier- als auch auf der Consumer-Seite wird jeweils ein *Push Model* und ein *Pull Model* unterstützt. Beim Pull Model erhält der Consumer die Events mittels eines Polling-Mechanismus. Beim Push Model geht die Initiative zur Event-Übermittlung vom Supplier aus.

Der Notification Service enthält als Teilmenge die Interfaces des Event Service. Sollen die Möglichkeiten des Event Filtering und der Quality-of-Service-Konfiguration genutzt werden, müssen jedoch die entsprechenden Interfaces des Notification Service eingesetzt werden.

Proxy Supplier und Proxy Consumer werden von Supplier-Admin-Objekten und Consumer-Admin-Objekten erzeugt. Innerhalb eines Channels können zudem mehrere Supplier-Admin- und Consumer-Admin-Objekte existieren. Es ist nun möglich, Filter- und Quality-of-Service-Parameter für den ganzen Channel, für Admin-Objekte innerhalb eines Channels oder aber für einzelne Proxy-Objekte zu konfigurieren. Hierbei gilt die Regel, daß die Filter- und QoS-Einstellungen, die für ein Admin-Objekt gelten, auch für alle Proxy-Objekte, die von diesem Admin-Objekt erzeugt wurden, Gültigkeit haben. Analog vererben sich auch die Filter- und QoS-Parameter, die auf Channel-Basis gesetzt wurden, auf alle Admin-Objekte, und damit auf die Proxy-Objekte, innerhalb des Channels. Durch die hierdurch resultierende Möglichkeit der hierarchischen Konfiguration wird die Administration der Event Channels erheblich vereinfacht.

5.2.1 Strukturierte Events

Neben den beim Event Service vorhanden Events vom Typ `Any` und den *typed Events* sieht der Notification Service auch strukturierte Events vor. Für einen Vergleich mit dem JMAPI Event Service sind diese am interessantesten, da sie den JMAPI Events am ähnlichsten sind. Abbildung 5.6 zeigt den Aufbau eines strukturierten Events.

Jedes strukturierte Event unterteilt sich in einen Header und einen Body. Der Header untergliedert sich wiederum in zwei Teile, den Fixed Header und den Variable Header. Motivation hierfür ist es, den Header möglichst klein zu halten. Die wesentliche Information findet sich im Fixed Header. Sollte es nötig sein, weitere Information im Header unterzubringen, so wird diese in den *Optional Header Fields* plaziert. Die eigentlichen Daten des Events befinden sich im Event Body. Hier besteht die Möglichkeit, Teilinformation, die als Grundlage für Filterentscheidungen dienen kann, herauszuseparieren und in Form von *Filterable Body Fields* zu speichern.

5.2.2 Quality-of-Service-Parameter

Der Notification Service sieht unter anderem folgende Quality of Service Parameter vor:

- **Reliability**: Es wird zwischen *EventReliability* und *ConnectionReliability* unterschieden, welche jeweils die Werte *BestEffort* und *Persistent* annehmen können. Hierüber kann konfiguriert werden, ob der Event Channel im Falle eines Neustarts die Verbindungen zu den bisherigen Clients automatisch wiederherstellt und ob im Falle des Ausfalls einer Verbindung vom Event Channel zu einem Event Consumer die Events für den jeweiligen Consumer zwischengespeichert werden, bis entweder ein Timeout auftritt, oder die Verbindung wieder hergestellt wird.

- **Priority**: Jedem Event kann ein Integer-Wert im Intervall zwischen -32 767 und 32 767 zugewiesen werden, wobei kleinere Zahlen eine niedrigere Priorität widerspiegeln.

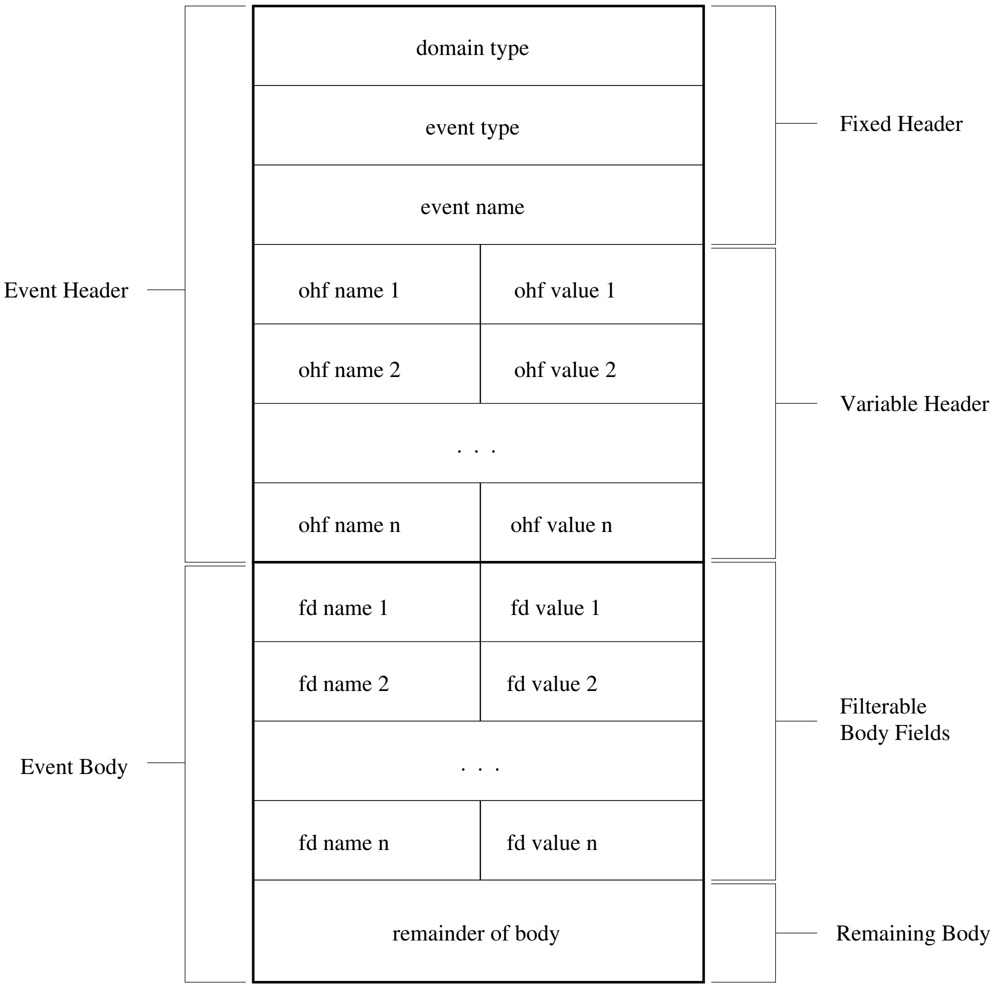


Abbildung 5.6: Aufbau eines Structured Event

- **Expiry Times**: Hier ist es möglich, einen Zeitpunkt zu definieren, nach dem ein Event nicht mehr ausgeliefert werden soll.

- **Earliest Delivery Time**: Es ist möglich, einen frühestmöglichen Zeitpunkt zu spezifizieren, zu dem ein Event ausgeliefert werden soll.

Ferner sind Parameter vorgesehen, die es gestatten, Policies festzulegen, welche Aussagen darüber treffen, wie mit zwischengespeicherten Events zu verfahren ist. Hierdurch sollen Fragen, wie etwa in welcher Reihenfolge Events ausgeliefert werden sollen, oder welche Events im Falle eines Puffer-Überlaufs zuerst verworfen werden sollen, geklärt werden.

5.2.3 Notification Service Filter

Der OMG Notification Service unterscheidet zwei verschiedene Arten von Filtern:

- Forwarding Filter und
- Mapping Filter

Forwarding Filter, die das Interface `CosNotifyFilter::Filter` implementieren, haben einen Einfluß darauf, welche Events weitergeleitet werden, und sind in dieser Hinsicht vergleichbar mit JMAPI-Event-Filtern.

Mapping Filter implementieren das Interface `CosNotifyFilter::MappingFilter` und dienen dazu, die Weiterleitung von Events über die Veränderung ihrer Priorität und Lebensdauer zu beeinflussen. JMAPI bietet kein Äquivalent hierzu.

Jedem Proxy-Objekt und jedem Admin-Objekt können jeweils mehrere Filter zugeordnet sein. Jedem Filter sind wiederum mehrere *Constraints* zugeordnet. Ein Constraint legt für einen oder mehrere Event-Types einen Boole'schen Ausdruck über der Menge der Header Fields und der Filterable Body Fields gemäß einer gewissen Constraint-Grammatik fest. Dieser Ausdruck wird für eintreffende Events ausgewertet. Falls er den Wert `true` liefert, wird das Event weiterverarbeitet, ansonsten wird es verworfen.

5.3 Vergleich der beiden Modelle

- Beiden Modellen gemeinsam ist das Vorhandensein von Event-Producern und Event-Consumern als Quellen und Senken von Events.

- Während die Weiterleitung von Events beim Notification Service mittels Event Channels erfolgt, wird eine vergleichbare Rolle bei JMAPI von den Event Dispatchern übernommen.

- Der Notification Service unterscheidet sowohl auf Supplier- wie auch auf Consumer-Seite zwischen *Push Model* und *Pull Model*. Es ist auch ein Mischen beider Modelle möglich. Im Gegensatz dazu gibt es beim JMAPI Event Model nur eine Push-Semantik. Die Events werden vom Producer an den Event Dispatcher gesandt, welcher dann die Weiterleitung an die entsprechenden Consumer übernimmt.

- Grundlegend für die Registrierung von Event Consumern für bestimmte Events im Rahmen von JMAPI ist der Event Tree, welcher Events in ein baumartiges Schema einordnet. Der Notification Service kennt keine baumartige Gliederung von Events. Die Registrierung erfolgt mittels der in den Constraints der Filterobjekte angegebenen Event-Typen.

- JMAPI Events sind den strukturierten Events des Notification Service ähnlich. Die `VecList`-Daten des JMAPI Events können auf die Filterable Body Fields eines Structured Events abgebildet werden.

- Sowohl der Notification Service als auch das JMAPI Event Model bieten die Möglichkeit des Event Filtering. Die JMAPI Filter sind mit Forwarding Filtern des Notification Services vergleichbar. Mapping Filter haben in JMAPI kein Äquivalent.

- Die Konfigurierbarkeit von Quality-of-Service-Parametern beim JMAPI Event Model ist gegenüber der beim Notification Service sehr eingeschränkt. Insbesondere kennt JMAPI keine priorisierten Events. Die Reliability kann bei JMAPI Events grob über die Parameter `RetryCount` und `RetrySleep` beeinflußt werden. Eine Expiry Time ist ebenfalls bei beiden Diensten vorhanden. Eine Earliest Delivery Time kann bei JMAPI Events nicht angegeben werden. Ebenso fehlt die Möglichkeit, Policies für die Auslieferung zwischengespeicherter Events festzulegen.

Kapitel 6

JMAPI-Agenten

Agenten dienen der tatsächlichen Ausführung von Managementaktionen auf den *Appliances* (vgl. Abbildung 2.1). Attributänderungen und Methodenaufrufe, die sich auf ein zu einer physischen Ressource gehörendes MO beziehen, müssen die Ausführung entsprechender Aktionen des Agenten nach sich ziehen. Die Implementierung der Agenten kann in verschiedener Form erfolgen. Nach einer kurzen Auflistung der Möglichkeiten folgen weitere Einzelheiten.

- JMAPI stellt spezielle *Agent Objects* zur Verfügung, die über eine *Agent Object Factory* verwaltet werden.
- SNMP-Agenten können über die mitgelieferten SNMP-Klassen angesprochen werden.
- Agenten, die mit Hilfe des Java Dynamic Management Kit (JDMK) entwickelt wurden, können ebenfalls über die RMI-Schnittstelle in das JMAPI Framework integriert werden.
- Die Verwendung von Agenten, die andere Kommunikationsprotokolle als RMI oder SNMP verwenden, wird von JMAPI zwar nicht direkt unterstützt, sie können aber dennoch eingesetzt werden. In diesem Fall ist die Implementierung des Managed Objects allerdings aufwendiger, da JMAPI hier keine Hilfsklassen zur Verfügung stellt.

6.1 Agent Objects und die Agent Object Factory

Bei Agent Objects handelt es sich um RMI Remote Objects. Die Managed Objects auf dem MO-Server kommunizieren tatsächlich mit Client Stubs, die dann mit Hilfe der Server Skeletons die eigentliche Kommunikation zwischen den MOs und den Agent Objects abwickeln. Gewöhnliche RMI Remote Objects haben die Eigenschaft, vom Garbage-Collection-Mechanismus der Java VM entfernt zu werden, sobald keine Referenzen von Client-Seite mehr auf das Objekt verweisen. Um die Möglichkeit von *long running agents* zu schaffen, die dieser Einschränkung nicht unterliegen, sieht JMAPI die Verwendung sog. *named agent objects* vor.

Zur Erzeugung neuer Agent-Object-Instanzen bedient sich ein MO der **Agent Object Factory**. Hierbei handelt es sich, analog zur MO Factory auf dem MO Server, um einen Prozeß, der beim Hochfahren des JMAPI-Systems auf den jeweiligen Appliances gestartet wird. Die Agent Object Factory bietet die Methoden `newAgentObj` und `newNamedAgentObj` für das Anlegen entsprechender Agent-Object-Instanzen an.

Im allgemeinen wird es nicht immer möglich sein, sämtliche auf den Agenten durchzuführenden Operationen vollständig in Java zu implementieren. Mittels des Java Native Interface (JNI) können jedoch einzelne Methoden der Agent Objects als *native methods* in einer anderen Programmiersprache implementiert werden. Der entsprechende Code wird in Shared Libraries zusammengefaßt, die dann vom Agent Object mittels des im folgenden erläuterten Library Loaders bei Bedarf geladen und auf dem Agenten gespeichert werden (siehe Abbildung 6.1). Durch Verwendung von *native methods* wird allerdings in den sonst plattformunabhängigen Code des Java-Agenten eine Abhängigkeit von Hardware- und Betriebssystemgegebenheiten induziert. Um dennoch die Möglichkeit zu haben, plattformunabhängige Agenten einzusetzen, stellt das **JMAPI Native Library Loading Interface** einen Mechanismus zur Verfügung, der das automatische Laden der richtigen Bibliothek vom Managed Object Server ermöglicht (vgl. [Sun97c], [Sun97g])[1]. Bei der Konfiguration des Agenten mit Hilfe der JMAPI-Properties-Datei (vgl. Abschnitt 9.1 und Anhang D) wird das Verzeichnis auf dem Server angegeben, von dem aus Bibliotheken für den jeweiligen Agenten zu laden sind. Hierdurch können Betriebssystemversion und Hardware-Architektur des Agenten berücksichtigt werden. Wie bereits erwähnt, werden vom Server geladene Bibliotheken auf dem Agenten für evtl. weitere Verwendung lokal gespeichert. Um sicherzustellen, daß die vom Agenten gecachete Bibliotheksversion auch der aktuellsten Version auf dem Server entspricht, werden vom JMAPI Native Library Loading Interface intern Versionsnummern verwaltet.

Der Ladevorgang läuft im einzelnen wie folgt ab ([Sun97g], [Sun97c]):

1. Das Agent Object fordert mittels Aufrufs der statischen Methode `loadLibrary` der Klasse `sunw.arm.agents.LibraryLoader` eine Bibliothek an.

2. Zunächst wird überprüft, ob die Bibliothek auf dem Appliance bereits vorliegt. Ist dies der Fall, wird mit Hilfe der lokal und auf dem MO Server verwalteten Versionsnummern festgestellt, ob die Version, die lokal vorliegt mit der aktuellsten Version auf dem Server übereinstimmt.

3. Ist die lokal vorhandene Version veraltet, oder ist die Bibliothek noch nicht vorhanden, wird die aktuellste Version vom Server geladen.

Bei der Installation einer neuer Bibliotheksversion auf dem Server wird die dort verwaltete aktuellste Versionsnummer entsprechend angepaßt.

[1] Um diesen Mechanismus nutzen zu können, muß auf dem MO Server ein spezieller Prozeß gestartet werden, der sog. *Net Class Server*.

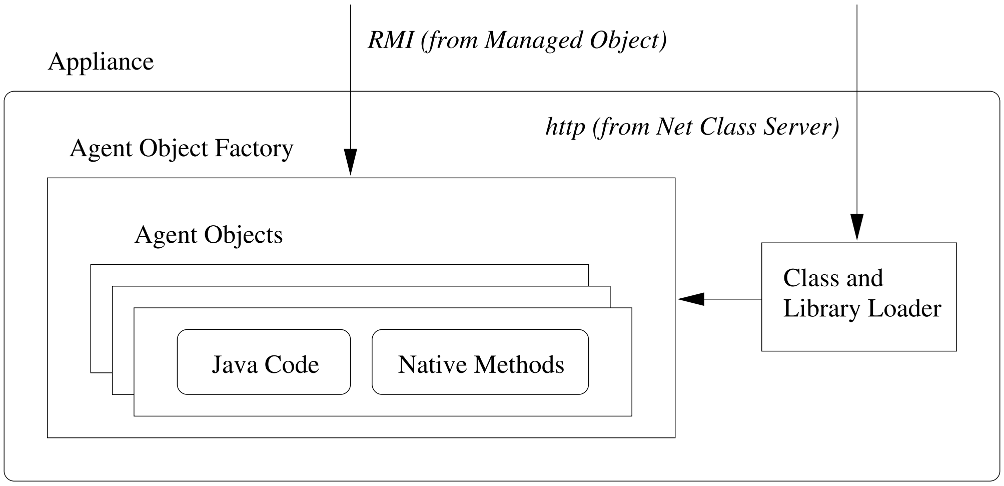


Abbildung 6.1: Agent Object Factory und Benutzung des JNI

Mit dem oben beschriebenen Mechanismus wird sichergestellt, daß die Agenten immer über die jeweils neueste Version verfügen. Ein Versions-Update ist nur auf dem Server vorzunehmen. Die Agenten ziehen die notwendigen Änderungen automatisch nach.

6.2 Die SNMP-Klassen

Um die Kommunikation von MOs mit SNMP-Agenten zu ermöglichen, bietet [JMA97] das von der Firma Cisco Systems entwickelte Java-Package `sunw.admin.snmp`, welches eine relativ komfortable SNMP-Schnittstelle bereitstellt [Sun97g].

Die Initialisierung der SNMP-Umgebung erfolgt durch Aufruf der Methode `SnmpMain.initialize()`. Will ein MO mit einem SNMP-Agenten kommunizieren, muß es zunächst ein `SnmpSession`-Objekt erzeugen. Mit Hilfe dieses Session-Objekts können mehrere `SnmpRequests`, die etwa *Get-*, *GetNext*, *Set*-Aufrufe darstellen, generiert werden, welche dann an den Agenten gesandt werden. Mittels der Klasse `SnmpPollRequest` können auch periodisch Requests generiert werden. IP-Adresse des Agenten, UDP-Port-Nummer und weitere Parameter, die die Kommunikation mit dem Agenten betreffen, wie etwa maximale PDU-Größe, werden von einem `SnmpPeer`-Objekt verwaltet, und können dort konfiguriert werden.

Die Behandlung von SNMP-Traps geschieht über die Klasse `SnmpTrapAgent`. Objekte, die an eintreffenden SNMP-Traps interessiert sind (sog. *Trap Receiver*), können sich beim `SnmpTrapAgent` registrieren. Der `SnmpTrapAgent` wartet nun auf einem spezifizierbaren Port (standardmäßig Port 162) auf SNMP-Trap-PDUs und benachrichtigt dann die registrierten Trap Receiver, die dann die Weiterverarbeitung vornehmen.

Die Verwaltung der MIB-Information übernimmt die Klasse `MibStore`. Sie speichert zu allen MIB-Variablen die drei Größen Object Descriptor, Object

Identifier und zugehörigen SMI-Datentyp, z. B. „`ifName`“, „`1.3.6.1.2.1.2.1`“ und „`Integer32`“.

Das Paket `sunw.admin.snmp` von [JMA97] enthält, soweit festgestellt werden konnte, keine vollständige MIB-2. Auch ein MIB-Compiler ist in [JMA97] nicht enthalten.

6.3 Einsatz von JMAPI-Agenten als Proxy-Agenten

Bei der Integration von SNMP-Agenten in das JMAPI-Framework sind verschiedene Ansätze denkbar (vgl. Abbildung 6.2).

Zum einen können die SNMP-Klassen dazu verwendet werden, die MOs auf dem MO Server direkt mit den SNMP-Agenten kommunizieren zu lassen (Abbildung 6.2a).

Als weitere Möglichkeit bietet sich der Einsatz eines JMAPI-Agenten als Proxy-Agent, der mit den MOs über RMI kommuniziert, und die RMI-Methodenaufrufe in SNMP-Operationen umsetzt, die er dann auf dem SNMP-Agenten ausführt (Abbildung 6.2b).

In Erweiterung des letzten Punktes wäre es ebenso denkbar, daß ein JMAPI-Agent mit mehreren SNMP-Agenten kommuniziert und damit selbst als Mid-Level-Manager agiert (Abbildung 6.2c).

Der Einsatz von JMAPI-Agenten als Proxy-Agenten oder Mid-Level-Manager bringt einerseits einen Performance-Verlust durch den zusätzlichen Kommunikationsschritt über RMI mit sich. Andererseits kann in diesen Fällen der Transaktionsmechanismus zur Mehrbenutzersynchronisation wieder genutzt werden (vgl. Abschnitt 6.6), eine Möglichkeit, die bei der direkten Kommunikation des MO Servers mit den SNMP-Agenten nicht besteht. Der Einsatz von JMAPI-Agenten als Mid-Level-Manager scheint jedoch wenig sinnvoll, da JMAPI-Agenten keine eigene Datenbank besitzen. Alle MO-Information müßte weiterhin auf dem MO Server gespeichert sein, wodurch keine Dezentralisierung des Managements erfolgen könnte.

Zusammenfassend ist von den drei obengenannten Ansätzen der erste sinnvoll, wenn Performance eine Rolle spielt und auf den JMAPI-Transaktionsmechanismus verzichtet werden soll. Der zweite Ansatz mit einem Proxy-Agenten ist hingegen von Vorteil, wenn man Mehrbenutzersynchronisation betreiben möchte.

6.4 Das Java Dynamic Management Kit

Beim Java Dynamic Management Kit (JDMK) handelt es sich um ein Produkt von Sun Microsystems, Inc, welches die Entwicklung von sog. *dynamischen Management-Agenten* unterstützen soll ([Sun98a] und [Gof98]). Dieser Begriff soll im folgenden zunächst kurz erläutert werden.

Die meisten der heute eingesetzten Agenten sind in ihrer Funktionalität beschränkt, indem sie nur einfache Managementaktionen ausüben können. Bei der Ausführung dieser Aktionen bedürfen sie der engen Zusammenarbeit mit einem Manager und verursachen eine dementsprechend große Netzlast. Zudem sind sie

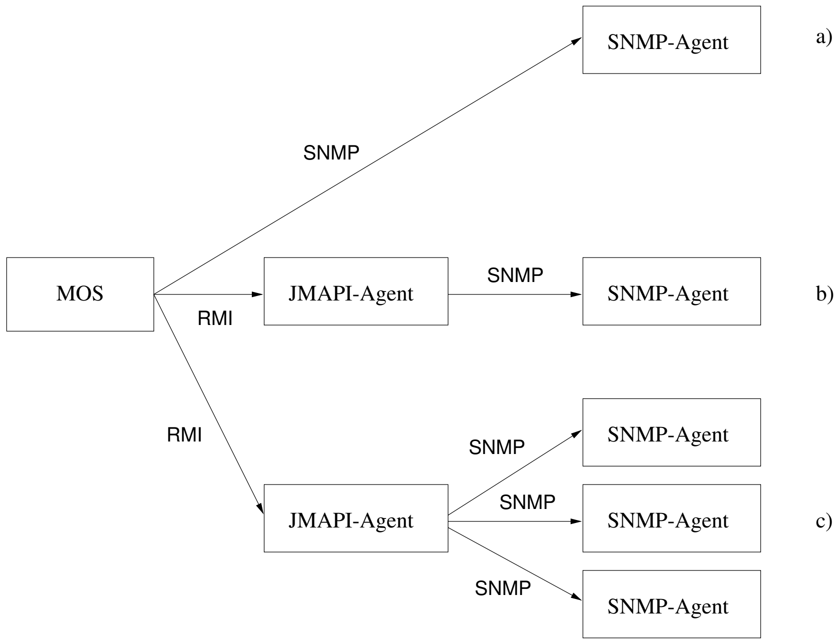


Abbildung 6.2: Einsatz von JMAPI-Agenten als Proxy-Agenten

in ihrem Aufbau statisch. Soll etwa ihr Funktionsumfang erweitert werden, so ist dies zumeist mit einem Neustart des Agenten verbunden, eine Änderung im laufenden Betrieb kommt i.d.R. nicht in Frage. Ferner müssen Änderungen oft vor Ort durchgeführt werden und können nicht immer *remote* über das Netz erfolgen. Der hieraus resultierende hohe Wartungsaufwand schlägt sich auch in entsprechend hohen Kosten nieder.

Grundidee bei der Entwicklung des JDMK war es daher, die Möglichkeit für autonomere Agenten zu schaffen, die auch komplexere Aktionen selbständig ausführen und soweit wie möglich ohne das Eingreifen eines Managers agieren können. Der Aufbau der Agenten sollte ferner so gestaltet werden, daß auch im laufenden Betrieb Updates am Agenten vorgenommen werden können.

JDMK-Agenten besitzen einen modularen Aufbau (vgl. Abbildung 6.3). Grundlage hierfür sind Software-Komponenten auf der Basis der JavaBeans-Technologie [Sun97f] *(JavaBeans Components for Management)*, die je nach Bedarf zu einem Agenten hinzugefügt, ausgetauscht oder entfernt werden können. Diese JavaBeans-Komponenten lassen sich unterteilen in

- **Core Management Services**, welche grundlegende Dienste, wie z. B. *Dynamic Class Loading Service*, *Dynamic Library Loading Service*, *Relationship Service*, *Basic Notification Service*, *Filtering Service* und *Monitoring Service* bereitstellen.

- **Master Agent Services**, die der Verteilung von JavaBeans-Komponenten auf andere Agenten durch Master Agents mittels der im folgenden erläuterten Push-Technik dienen *(Cascading Service)*.

- **Managed Objects**, welche die gerätespezifischen Managementfunktionen implementieren.
- **Protocol Adaptors**, welche es den Agenten erlauben, verschiedene Kommunikationsprotokolle transparent zu nutzen. Die zur Zeit unterstützten Protokolle sind RMI, HTTP und SNMP.

Das Grundgerüst eines Agenten, zu welchem die Komponenten addiert werden, wird durch das *Common Management Framework* (CMF) gebildet.

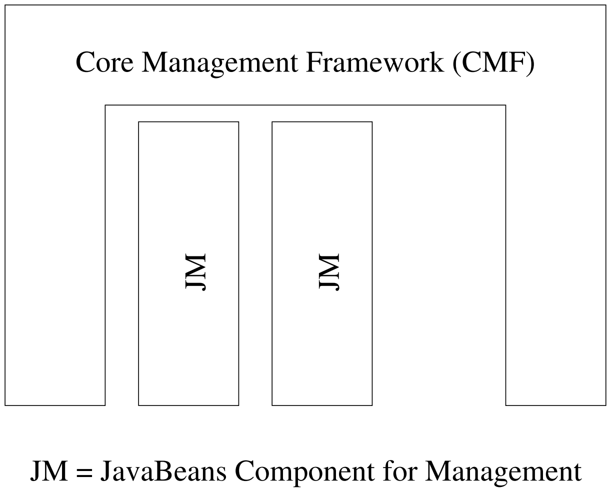


Abbildung 6.3: Aufbau eines JDMK-Agenten

JDMK Agenten können die von ihnen benötigten JavaBeans-Komponenten sowohl mittels *Pull*-, als auch über *Push*-Technik erhalten (vgl. Abbildung 6.4) [Sun98a]. Bei der Pull-Technik fordert der Agent die entsprechenden Komponenten von einem Web-Server an. Dies geschieht beispielsweise beim Starten des Agenten. Zusätzlich besteht von Seiten des Managers auch die Möglichkeit, Komponenten, die dem System neu hinzugefügt wurden, auf die Agenten durch eine Push-Operation zu übertragen.

Da nur diejenigen Dienste geladen werden müssen, die auch tatsächlich gebraucht werden, können die JDMK-Agenten im allgemeinen relativ klein gehalten werden (minimal 200-300 kB nach [Gof98]).

Integration mit JMAPI

JDMK-Agenten können über die SNMP- und RMI-Protokolladaptoren in das JMAPI-Framework eingebunden werden. In diesem Sinn sind JDMK und JMAPI miteinander kompatibel.

Allerdings gehen JDMK und JMAPI von verschiedenen Modellen aus. Während JMAPI einen klassischen zentralisierten Ansatz verfolgt, in dem ein (oder evtl. mehrere, vgl. Kapitel 8) Managed Object Server als Ausgangspunkt von Managementaktionen betrachtet werden, versucht JDMK gerade

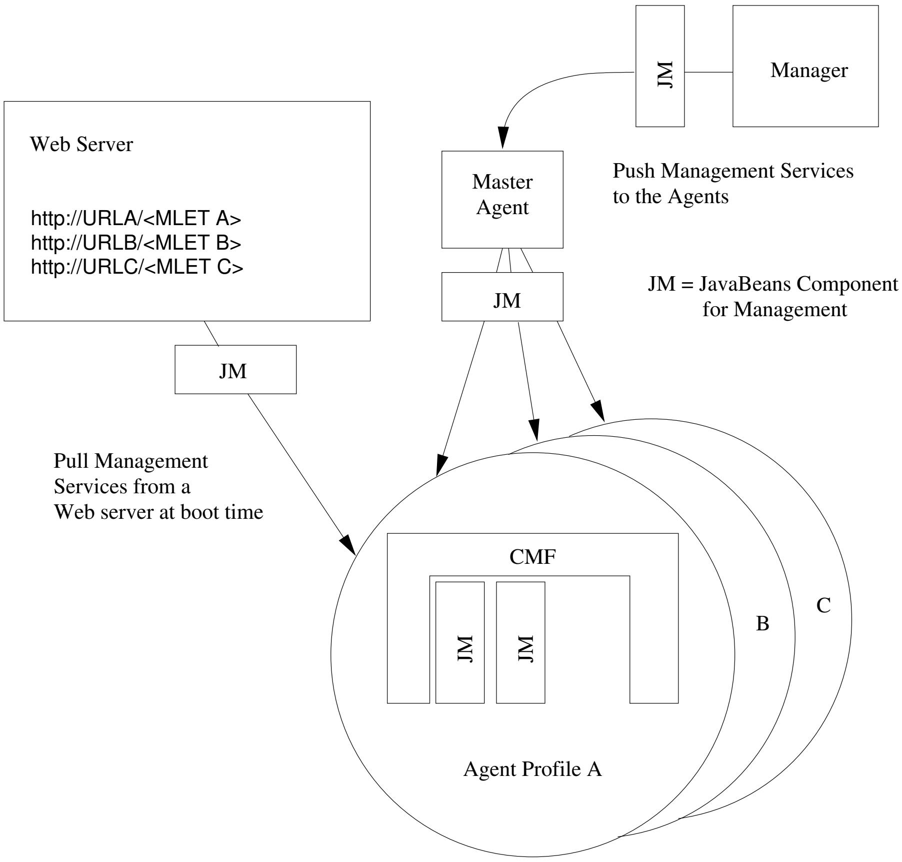


Abbildung 6.4: Dynamisches Laden von Management Services mittels Push- oder Pull-Technik

durch Einführung autonomerer Agenten, die auch untereinander kommunizieren können, die Nachteile dieses Organisationsmodells aufzuheben.

Da JDMK allerdings nur ein Toolkit für die Entwicklung von *Agenten* ist, könnte es von den GUI-Komponenten von JMAPI profitieren, wenn es um die Erstellung einer Benutzeroberfläche geht.

6.5 CORBA-Agenten

In den Abschnitten 6.1, 6.2, 6.3 und 6.4 verlief die Kommunikation zwischen dem Managed Object Server und den Agenten jeweils über RMI oder SNMP. Die Einbindung von Agenten auf der Basis anderer Kommunikationsprotokolle, wie etwa CORBA IIOP, ist ebenfalls möglich. JMAPI liefert in diesem Fall allerdings keine Unterstützung in Form von Java-Packages, wie dies etwa bei SNMP der Fall ist. In Abschnitt 7.4 wird als Beispiel eine JMAPI-konforme Benutzeroberfläche für einen am Lehrstuhl vorhandenen CORBA-Agenten vorgestellt.

Für die Integration von CORBA-Agenten sind folgende Möglichkeiten denkbar:

- Die JMAPI-Clients kommunizieren direkt mit den CORBA-Agenten. Dieser Ansatz ist sinnvoll, wenn es sich bei den CORBA-Objekten um Factory-Objekte handelt, die andere Objekte erzeugen und verwalten.

- Es werden JMAPI-Proxy-Agenten analog zu Abschnitt 6.3 eingesetzt. Die JMAPI-Managed-Objects auf dem MO Server kommunizieren in diesem Fall über RMI mit den Proxy-Agenten auf den Appliances. Die Proxy-Agenten sind in diesem Fall echte JMAPI-Agenten, und können somit die gesamte von JMAPI bereitgestellte Funktionalität nutzen.

- Zu jedem CORBA-Objekt wird auf dem MO Server ein JMAPI-MO angelegt. Dieses kommuniziert mit seinem zugehörigen CORBA-Objekt über IIOP. In diesem Fall sind alle MO-Instanzen doppelt vertreten, nämlich einmal als JMAPI-MO und einmal als CORBA-Objekt. Dies kann zu Inkonsistenzen führen, wenn beispielsweise ein CORBA-Objekt gelöscht wird, ohne das zugehörige JMAPI-MO zu entfernen.

6.6 Der Transaktionsmechanismus

Aktionen, die von den Agenten ausgeführt werden, sind im JMAPI-Kommunikationsmodell die Folge von Manipulationsoperationen an Managed Objects.

Im Gegensatz zu reinen Leseoperationen auf MOs können Änderungen, die den Zustand einzelner MOs betreffen, bzw. neue MO-Instanzen anlegen oder vorhandene MOs löschen, nur innerhalb eines sog. **Update-Kontexts** erfolgen. Dieser faßt eine Reihe von Änderungsaktionen zu einer Transaktion mit den aus dem Datenbankbereich bekannten ACID[2]-Eigenschaften zusammen. Obwohl ein erfolgreich abgeschlossener Update-Kontext Änderungen in der Datenbank zur Folge hat, ist er nicht mit der darunterliegenden Datenbanktransaktion zu verwechseln.

Sind die Agenten, die von Änderungen einer Reihe von MOs innerhalb einer Transaktion betroffen sind, als in Java implementierte Agent Objects realisiert, die durch eine Agent Object Factory verwaltet werden (vgl. Abschnitt 6.1), so bietet JMAPI die Möglichkeit, die Änderungsaktionen auf den Agenten ebenfalls in den Tranksaktionsmechanismus mit einzubeziehen.

Zur Verdeutlichung der Zusammenhänge soll im folgenden exemplarisch vorgeführt werden, wie ein Client Modifikationen an Managed Objects vornimmt, die als Folge Aktionen von Agenten nach sich ziehen. Abbildungen 6.5 und 6.6 zeigen schematisch die zugehörigen Abläufe.

[2]Atomicity, Consistency, Isolation, Durability

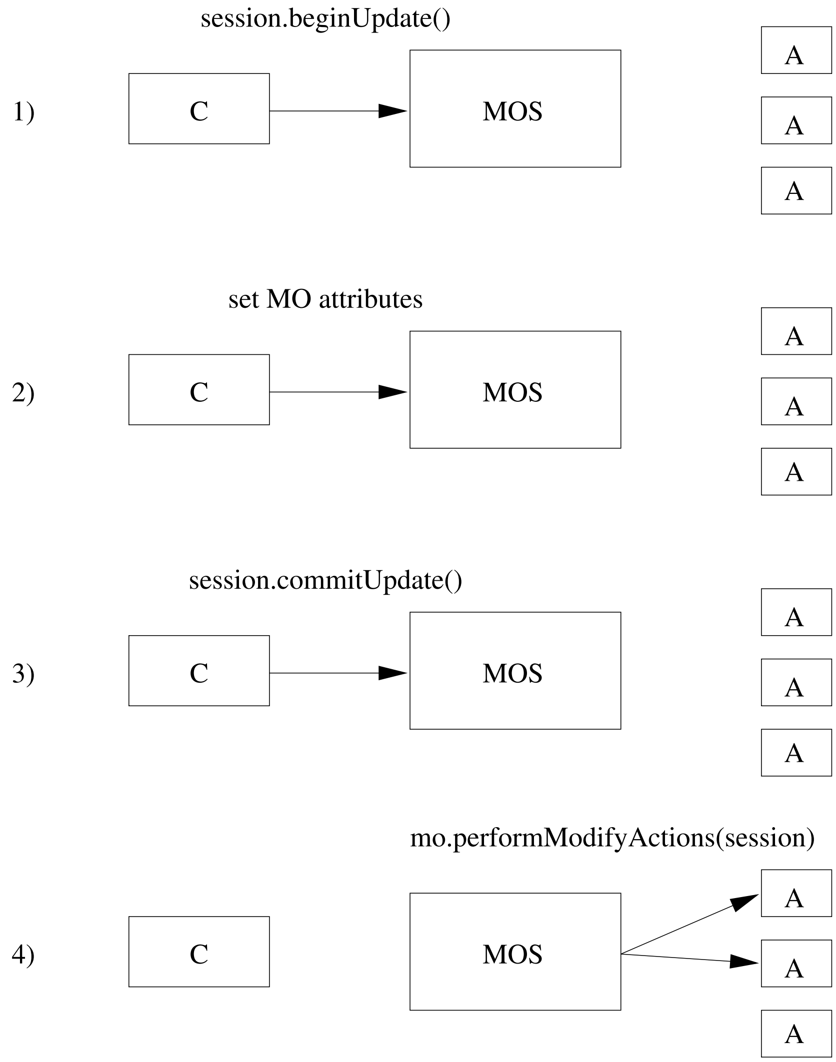


Abbildung 6.5: Ablauf eines Updates, Teil 1

Zunächst öffnet der Client mit dem Aufruf `beginUpdate` auf einem `Session`-Objekt einen neuen Update-Kontext[3]. Jetzt kann er mittels der `get`- und `set`-Methoden, die die in Betracht kommenden Managed Objects für jedes ihrer Attribute bereitstellen, Modifikationen an einzelnen Managed Objects vornehmen (Schritt 2). Hat der Client die gewünschten Veränderungen am Zustand der MOs vorgenommen, schließt er mittels des Aufrufs `commitUpdate` die Transaktion ab (Schritt 3). An dieser Stelle werden von JMAPI

[3]Das `Session`-Objekt, welches zum Beginn der Kommunikation eines Clients mit dem MO Server erzeugt wird, verwaltet nebem dem Update-Kontext auch einen **Security Context**, der einen Autorisierungsmechanismus bereitstellen soll. In [JMA97] ist diese Funktionalität allerdings noch nicht implementiert, es existiert lediglich ein Platzhalterobjekt, welches eine geplante Erweiterung andeutet.

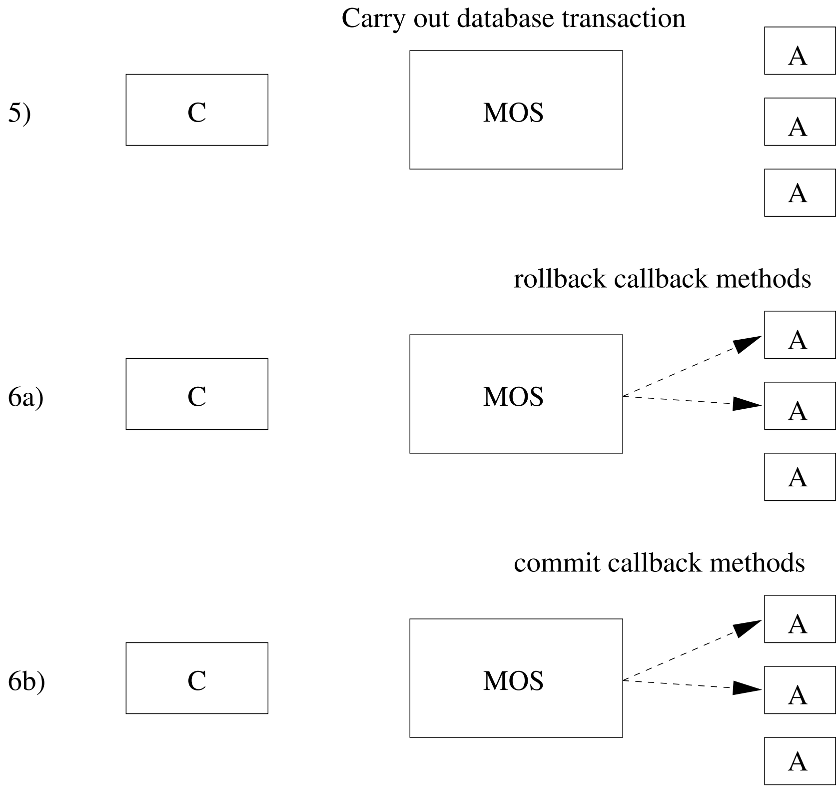


Abbildung 6.6: Ablauf eines Updates, Teil 2

automatisch die `performModifyActions`-Methoden eines jeden Objekts, das verändert wurde, aufgerufen (siehe hierzu auch Abschnitt 3.2). Die Standard-Implementierung dieser Methoden ist leer. Der Anwendungsentwickler kann durch Umdefinieren dieser Methoden Aktionen auf den Agenten ausführen lassen, die dann bei jeder Modifikation des MO-Zustandes zum Einsatz kommen (Schritt 4). Als nächstes wird versucht, die Änderungen in der Datenbank zu speichern (Schritt 5 in Abbildung 6.6). Tritt hierbei ein Fehler auf, so wird ein Rollback der Transaktion veranlaßt. Da es möglich ist, daß auf den Agenten im Rahmen der `performModifyActions`-Ausführung Änderungen vollzogen wurden (vgl. Schritt 4), muß in diesem Fall dafür Sorge getragen werden, diese Änderungen rückgängig zu machen. Zu diesem Zweck wird beim Aufruf von `performModifyActions` dem Agenten in einer für den Benutzer transparenten Weise ein `CommitAndRollback` Objekt übergeben. Der Agent kann mit Hilfe dieses Objekts beim Manager zwei Callback-Funktionen registrieren, jeweils für die beiden Fälle einer erfolgreichen oder fehlgeschlagenen Datenbanktransaktion. Im Falle eines Rollbacks wird dann die Rollback-Callback Funktion ausgeführt (Schritt 6a in Abbildung 6.6), im Fall eines Commits die Commit-Callback-Funktion (Schritt 6b).

Wie aus diesen Ausführungen ersichtlich ist, beinhaltet jede Modifikation eines Agenten drei mögliche Phasen. Die **Setup-Phase**, in obiger Erläuterung das Aufrufen von Methoden auf dem Agenten von der `performModifyActions` Methode aus (vgl. Schritt 4), kann bereits Änderungen auf dem Agenten durchführen. Je nachdem, ob das Schreiben der modifizierten Attributwerte des Managed Objects in die Datenbank erfolgreich war oder nicht, wird im Anschluß hieran die **Commit-Phase** oder die **Rollback-Phase** (vgl. Schritt 6a bzw. 6b) eingeleitet.

Im Hinblick auf die Implementierung von Aktionen auf den Agenten ergeben sich somit verschiedene Möglichkeiten. Beispielsweise könnte während der Setup-Phase nichts gemacht werden und alle Änderungen innerhalb der Commit-Phase erfolgen. Ein Rollback wäre dann nicht notwendig. Alternativ könnten auch alle Änderungen während der Setup-Phase vollzogen werden, die Commit-Phase wäre dann leer und die Rollback-Phase müßte die Aktionen der Setup-Phase rückgängig machen.

Vorangehend wurde der Ablauf von Modifikationsoperationen auf eine Reihe von Managed Objects an einem Beispiel geschildert. Sollen MO-Instanzen gelöscht werden, oder neue Instanzen angelegt werden, geschieht dies im wesentlichen auf dieselbe Weise. Nach Abschluß des Update-Kontexts mit `commitUpdate` werden in diesen Fällen die Methoden `performDeleteActions` bzw. `performNewActions` ausgeführt.

Ein Rollback wird nicht nur durch eine gescheiterte Datenbanktransaktion ausgelöst, sondern kann auch von einem Client mittels Aufruf der Methode `abandonUpdate` ausgelöst werden.

Kapitel 7

Die JMAPI-Benutzeroberfläche

Eines der Ziele von JMAPI ist es, dem Benutzer eine konsistente Oberfläche für die Durchführung der Managementaufgaben zur Verfügung zu stellen. Diesem Zweck dienen die Klassen des **Admin View Module** (AVM), die auf dem Java Abstract Window Toolkit (AWT) aufbauen und dieses um Elemente erweitern, die die einfache Erstellung einer für diesen speziellen Zweck angepaßten Oberfläche erleichtern sollen. Bei [JMA97] handelt es sich hierbei um die Klassen des Java-Packages `sunw.admin.avm.base`. Zur Umsetzung eines einheitlichen *Look and Feels* werden ferner zwei Style Guides mit ausgeliefert, der **User Interface Style Guide** [Sun97d] und der dazu komplementäre **User Interface Visual Design Style Guide** [Sun97e]. Ersterer befaßt sich mit der Beschreibung der wesentlichen Elemente, aus denen sich die JMAPI-GUI zusammensetzt, sowie welches Verhalten diese Oberflächenelemente bei der Interaktion mit dem Benutzer aufzuweisen haben. Der zweite Style Guide spezifiziert das visuelle Erscheinungsbild der Oberfläche, wie Farben, Fonts, Größe der GUI-Komponenten und deren Plazierung.

Die Tatsache, daß das AVM auf dem AWT aufsetzt, hat den Vorteil der einfachen Portabilität, da hierdurch eine Unabhängigkeit von dem verwendeten Window-System gegeben ist.

7.1 Benutzersicht von JMAPI

Grundlage der Interaktion des Benutzers mit dem JMAPI-Managementsystem ist die Browser-Oberfläche mit den bekannten Navigationsmöglichkeiten, wie beispielsweise Hyperlinks, Back- und Forward-Buttons und Bookmarks. Eine auf der bekannten Web-Browsing-Metapher basierte Oberfläche erleichtert es dem Anwender, sich schnell zurechtzufinden und ermöglicht diesem, sich auf die eigentliche Aufgabe zu konzentrieren.

Der Benutzer interagiert mit dem System mittels innerhalb des Browsers dargestellter Web-Seiten, sowie über Dialog-Fenster, die als eigenständige Elemente auf dem Bildschirm erscheinen.

Es lassen sich zwei grundsätzlich verschiedene Verwaltungsaufgaben unterscheiden: Die Anzeige von Statusinformation bereits existierender Managed Objects einerseits und das Neuanlegen bzw. Ändern von MOs andererseits. Zur Unterstützung dieser Aufgaben stellt JMAPI zwei verschiedene GUI-Komponenten bereit. Statusdaten werden durch **Content Manager** (siehe Abbildung 7.1) dargestellt, Manipulationen von MOs geschehen mittels **Property Books** (vgl. Abbildung 7.2). Property Books dienen ferner der Konfiguration des Erscheinungsbildes von GUI-Komponenten, wie Web-Seiten oder Content Managern.

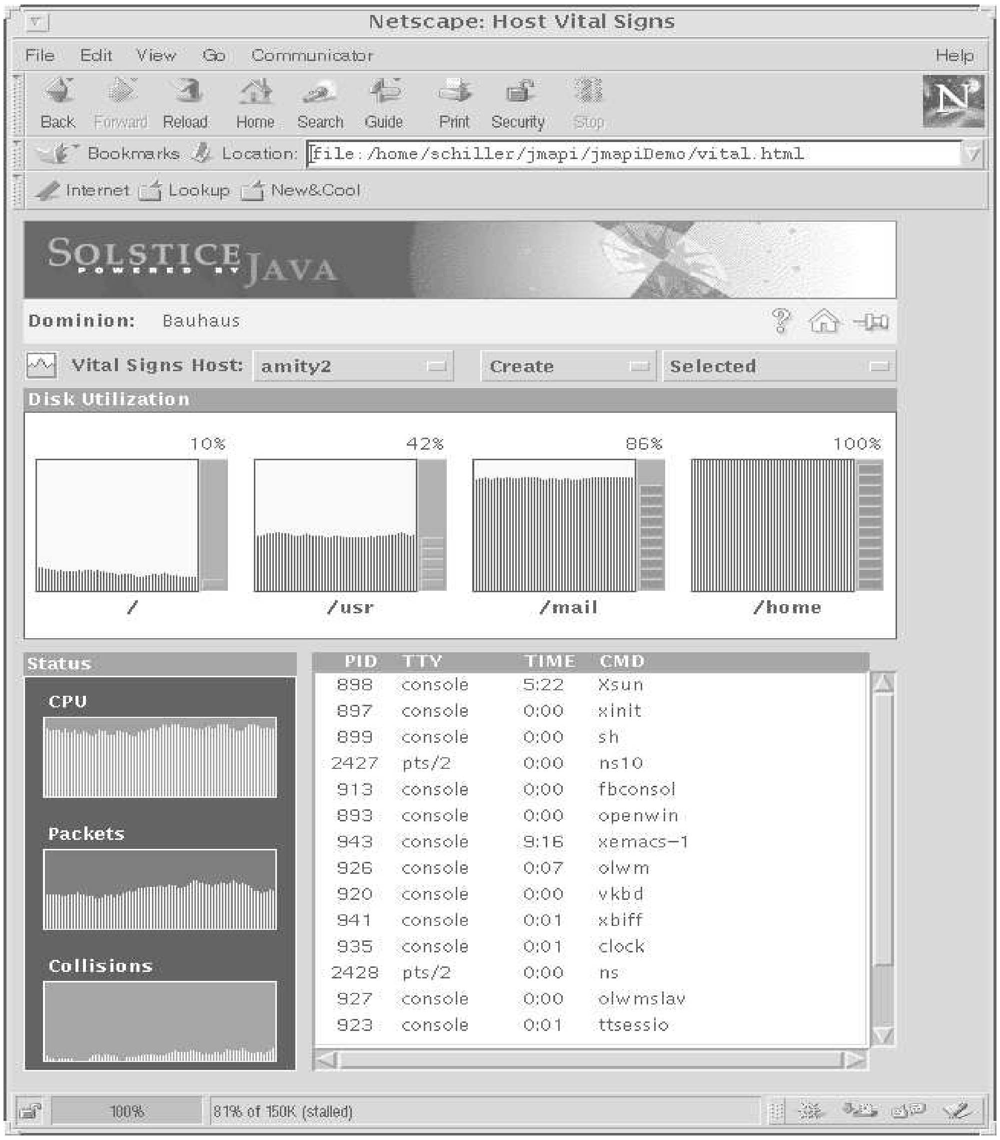


Abbildung 7.1: Eine Web-Seite mit mehreren Content Managern

Content Manager können entweder innerhalb eines Dialog-Fensters, oder im Rahmen einer Web-Page dargestellt werden. Welche Darstellungsform gewählt wird, liegt im Ermessen des Anwendungsprogrammierers. Ein Dialog-Fenster bietet den Vorteil, daß die Information sichtbar bleibt, auch wenn im Browser

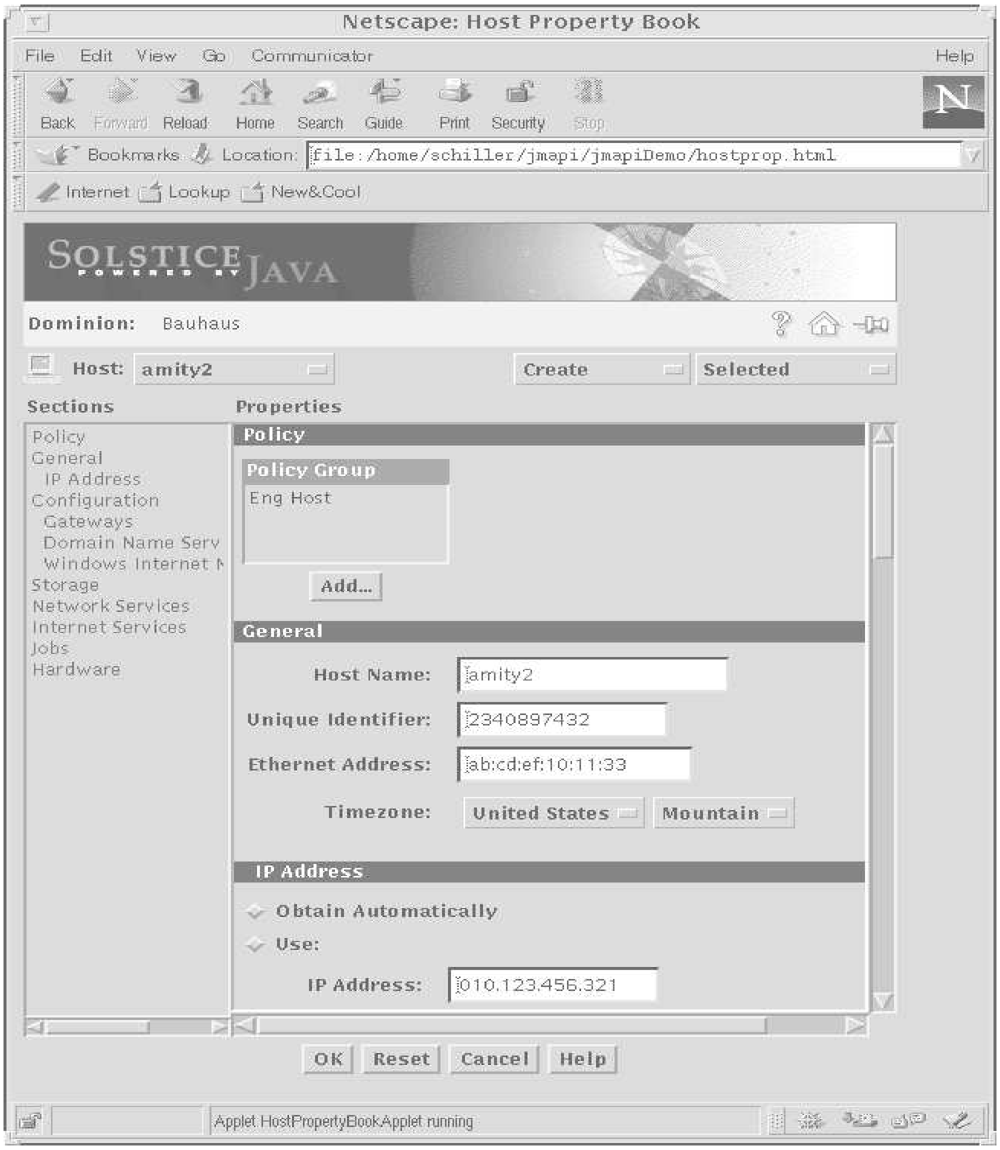


Abbildung 7.2: Eine Web-Seite mit einem Property Book

zu einer anderen Seite gewechselt wird. Auf einer Web-Page können jedoch mehrere Content Manager gleichzeitig dargestellt werden, ohne daß der Benutzer mehrere Dialogfenster verwalten muß.

Property Books können ebenfalls auf einer Web-Seite oder in einem Dialog-Fenster dargestellt werden. Die Wahl der Darstellung soll sich nach [Sun97d] an folgenden Richtlinien orientieren:

- Die Konfiguration bereits existierender MOs sollte mittels eines Property Books auf einer Web-Seite erfolgen.
- Für die Erstkonfiguration neu anzulegender MOs ist ein Dialog-Fenster zu verwenden.

- Soll das Erscheinungsbild einer GUI-Komponente mittels eines Property Books verändert werden, soll ebenfalls ein Dialog-Fenster eingesetzt werden.

Innerhalb eines Content Managers können verschiedene *Status Components* eingesetzt werden. Hierbei handelt es sich um spezielle GUI-Komponenten, die speziell für die Anzeige von Statusdaten geeignet sind, wie beispielsweise `StripChart` für die zweidimensionale Darstellung einer Folge von Meßwerten, oder `Gauge` für die balkenartige Darstellung gemessener Größen.

7.2 Bezug zu anderen APIs

Parallel zur Entwicklung von JMAPI wird zur Zeit an einer Reihe anderer APIs gearbeitet, deren Funktionalität sich teilweise mit der von JMAPI bereitgestellten Funktionalität überlappt. Die JavaHelp API [Sun98c] für die Erstellung eines Java-basierten integrierten Hilfe-Systems stellt ein Beispiel hierfür dar. Hinsichtlich der Gestaltung der Benutzeroberfläche existiert mittlerweile mit den Java Foundation Classes (JFC) [Sun98b] eine Entwicklungsbasis, die größere Flexibilität und eine reichhaltigere Palette von GUI-Komponenten bei der Oberflächenerstellung als das herkömmliche AWT bietet.

Mit fortschreitendem Entwicklungsstadium von JMAPI wird es zu erwarten sein, daß einige Klassen, die zunächst Teil von JMAPI waren, nicht mehr benötigt werden, da die von ihnen bereitgestellte Funktionalität durch entsprechende Klassen anderer APIs realisiert wird. Bereits in Abschnitt 2.3.1 wurde erwähnt, daß aus diesem Grund die in [JMA97] noch enthaltenen Klassen zum Erstellen einer integrierten Online-Hilfe in der nächsten JMAPI-Version nicht mehr enthalten sein werden, da die entsprechenden Aufgaben von den JavaHelp-Klassen übernommen werden können.

In analoger Weise werden Überschneidungen zwischen den AVM-Klassen von JMAPI und JFC-Klassen in Zukunft beseitigt werden. Als Beispiel kann die Klasse `sunw.admin.avm.base.Browser` zur Darstellung einer hierarchischen Baumstruktur angeführt werden, die bereits in [JMA97] als `deprecated` markiert ist, und durch die JFC-Klasse `JTree` ersetzt werden soll.

7.3 Integration mit dem JMAPI Framework

Bei den im Package `sunw.admin.avm.base` enthaltenen Klassen handelt es sich hauptsächlich um GUI-Komponenten, die zwar für die Entwicklung einer JMAPI-konformen Oberfläche entwickelt wurden, die aber ansonsten in keiner Beziehung etwa zu den MO-Klassen stehen, auf deren Instanzen der Benutzer eigentlich mittels des GUI zugreifen möchte. Das eigentliche Bindeglied zu den Managed Object Klassen sollen nach [Sun97c] die *AVM Integration Classes* liefern. Diese Klassen sind allerdings in [JMA97] nicht vollständig enthalten[1].

[1] Nach der Installation der Solaris-Version von [JMA97] finden sich in den Unterverzeichnissen von `/opt/JMAPI/examples/glue` einige Klassen, die Teile der AVM Integration Classes darstellen. Die vorhandenen Klassen sind in der vorliegenden unmodifizierten Form nicht

Zu den Hauptaufgaben der AVM Integration Classes gehören die automatische Erzeugung von Property Books zu einer zugehörigen MO-Instanz. Ferner können Komponenten wie Management-Applets, Web-Seiten und Hyperlinks an zentraler Stelle auf dem Management Server registriert werden um sie allen Clients zugänglich zu machen.

Die AVM Integration Classes sollen das Erstellen einer JMAPI-konformen Benutzeroberfläche für die Manipulation von Managed Objects erleichtern, sind aber für die Zusammenarbeit von JMAPI-Client und Managed Object Server nicht unbedingt notwendig. Im Rahmen von [Rad98] wurde die am Lehrstuhl vorhandene JMAPI-Datenbank zur Speicherung bestimmter Managementinformation verwendet. Für die Manipulation der hierbei eingesetzten Managed Objects wurde als Teil der vorliegenden Arbeit eine graphische Oberfläche auf Basis von Content Managern, Property Books und Dialogfenstern erstellt. Nähere Einzelheiten hierzu sowie der zugehörige Programmcode finden sich in Anhang C.

Das Beispiel demonstriert die Zusammenarbeit der AVM-Klassen mit dem Managed Object Server zum Anlegen, Modifizieren und Löschen von Managed Objects. Aufgrund der fehlenden AVM Integration Classes konnte auf sie in diesem Beispiel nicht zurückgegriffen werden.

7.4 Benutzeroberfläche für einen CORBA-Agenten

Als Beispiel für die Verwendung der AVM-KLassen soll im folgenden die Erstellung einer Benutzeroberfläche für einen CORBA-Agenten dienen. Neben der Verwendung der AVM-Klassen führt es auch eine Möglichkeit zur Integration anderer Managementarchitekturen in das JMAPI-Framework vor.

Im Rahmen von [Mül98] wurde die prototypische Implementierung eines CORBA-Agenten vorgenommen, der u.a. zur Benutzerverwaltung auf den UNIX-Workstations des Lehrstuhls eingesetzt werden kann[2]. Der Agent liest die vorhandene Benutzerinformation aus der lokalen Passwort-Datei und aus den NIS-Maps aus und erzeugt daraufhin für jeden Benutzer ein `Account`-Objekt. Die Menge aller Account-Objekte wird von einem `AccountFactory`-Objekt verwaltet.

Am Beispiel der `Account`-Objects soll im folgenden demonstriert werden, wie die Erstellung einer JMAPI-konformen Oberfläche mit Hilfe der AVM-Klassen realisiert werden kann.

Es wäre zunächst denkbar, eine Managed-Object-Klasse `AccountMO` anzulegen und dann für jeden Benutzer ein entsprechendes `AccountMO`-Objekt auf dem Managed Object Server zu verwalten. Mit Hilfe der *AVM Integration Classes* (vgl. Abschnitt 7.3) ließe sich dann eine Benutzeroberfläche konstruieren, die auf die `AccountMO`-Klassen zugreift.Änderungen an `AccountMO`-Objekten

einsetzbar, allerdings kann man durch das Studium des Quellcodes wertvolle Einblicke in die Funktionsweise der AVM Integration Classes gewinnen. Die Dokumentation muß mittels `javadoc` aus dem Source-Code generiert werden.

[2]Die Implementierung erfolgte unter AIX, aus diesem Grund ist der Agent nur auf IBM-Rechnern lauffähig.

müßten auf entsprechende Änderungen an den zugehörigen CORBA `Account`-Objekten abgebildet werden.

Diese Vorgehensweise hätte allerdings folgende Nachteile:

- Für jeden Benutzer würden sowohl ein JMAPI-`AccountMO`-Objekt auf dem Managed Object Server als auch ein CORBA-`Account`-Object angelegt werden. Diese doppelte Repräsentation würde nur für die einfachere Erstellung der Benutzeroberfläche eingeführt werden, wäre aber ansonsten unnötig.

- Da die Kommunikation der Managed Objects mit der CORBA-`AccountFactory` mittels IIOP erfolgt, kann nach Abschnitt 6.6 der in JMAPI integrierte Transaktionsmechanismus nicht genutzt werden. Wie in Abschnitt 8.4 gezeigt wird, kann es daher u.U. sogar zu Inkonsistenzen zwischen dem Zustand des `AccountMO` und des CORBA-`Account`-Objekts kommen.

- Mit dem doppelten Verwaltungsaufwand ist auch ein Performance-Verlust verbunden.

Aus diesen Gründen wurde eine andere Vorgehensweise gewählt, bei der das JMAPI-Client-Applet direkt auf das CORBA-`AccountFactory`-Objekt zugreift (siehe Abbildung 7.3).

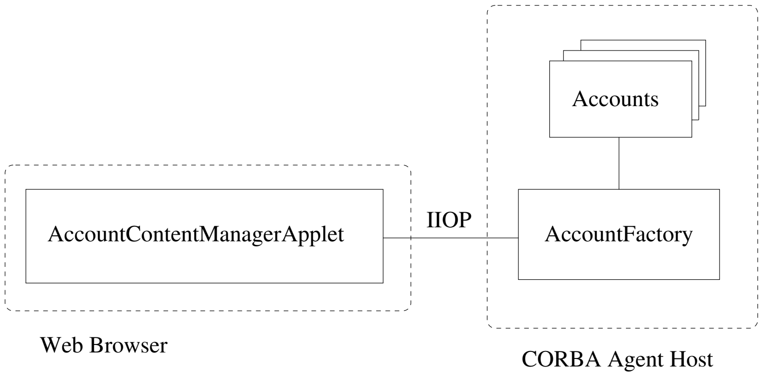


Abbildung 7.3: Kommunikation des JMAPI-Applets mit dem CORBA-Agenten

Zur Darstellung der Account-Daten wurde die Klasse `sunw.admin.avm.base.SimpleContentManager` gewählt. Sie ermöglicht die Darstellung von Daten wahlweise in Tabellenform, oder als eine Reihe von Icons. Abbildung 7.4 zeigt die zugehörigen Darstellungen der Account-MOs.

Neben der Darstellungsform als Icons oder als Tabelle lassen sich auch noch eine Reihe von Filter- und Sortiereigenschaften festlegen. Hierzu dient das `View`-Pulldown-Menü am oberen Rand des Applets, über welches vier Dialog-Fenster anwählbar sind.

- Der „`Properties...`“ Dialog erlaubt zwischen der Icon- und der Tabellendarstellung zu wählen.

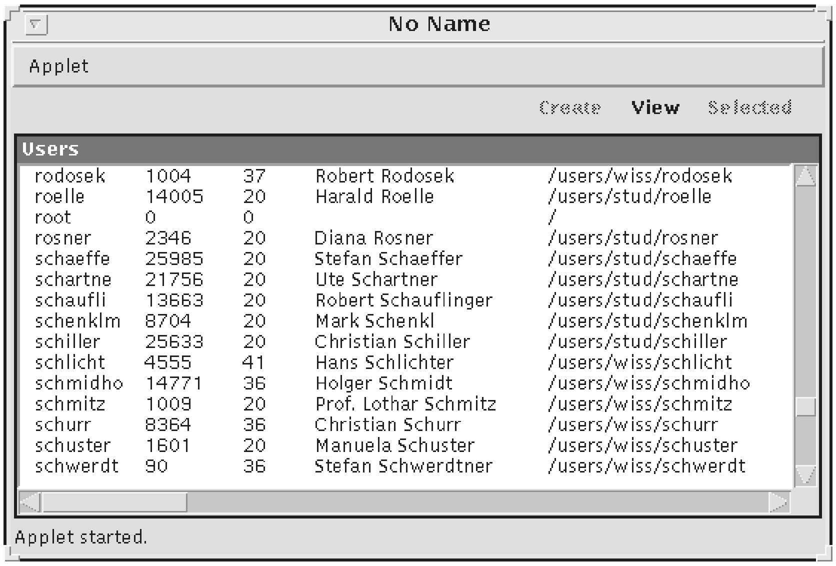
- Mittels des „`Filter...`“ Dialogs kann eine Teilmenge von `Account`-Objecten ausgewählt werden, deren Attributwerte gewisse Bedingungen erfüllen, beispielsweise „Alle Benutzer, deren Kennung mit dem Buchstaben `S` beginnt“. Dies ist etwa vergleichbar mit einer Selektionsoperation bei einer relationalen Datenbankabfrage.

- Der „`Sort...`“ Dialog ermöglicht ein Sortieren der dargestellten `Account`-Objekte. Es kann nach einem oder mehreren Attributen sortiert werden. Ferner ist es möglich, in auf- oder absteigender Reihenfolge zu sortieren.

- Mittels des „`Display...`“ Dialogs kann schließlich aus der Menge aller Attribute eine Teilmenge ausgewählt werden. Nur die ausgewählten Attribute werden dann in der Tabellenform angezeigt. Dies entspricht einer Projektionsoperation bei einer Anfrage an eine relationale Datenbank.

Die Realisierung der obengenannten `Filter`-, `Sort`- und `Display`-Funktionen erfolgt über sogenannte *Pipes*: Die Klasse `sunw.admin.avm.base.TableDataContentManager`, von der die Klasse `SimpleContentManager` erbt, verwaltet je ein `TableFilterPipe`-, `TableSortPipe`- und ein `TableViewPropertiesPipe`-Objekt, welche die obigen Funktionen umsetzen.

Der zugehörige Programmcode findet sich in Anhang B.



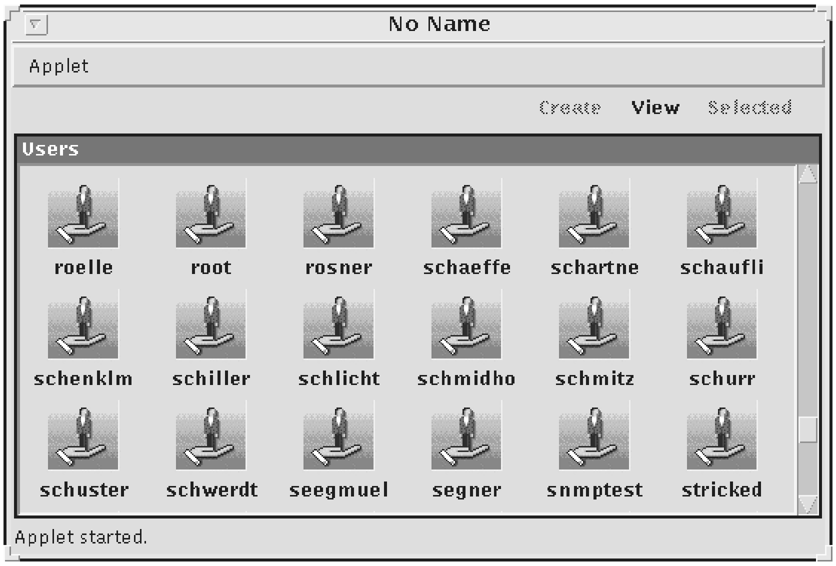


Abbildung 7.4: Tabellen- und Icondarstellung der Benutzer-Accounts

Kapitel 8

Eignung von JMAPI für das integrierte Management

Im folgenden soll eine Bewertung der JMAPI-Managementarchitektur im Hinblick auf den Einsatz als Basis integrierter Managementanwendungen erfolgen. Nach einer Auflistung der Anforderungen, die an integrierte Managementlösungen zu stellen sind, folgt die nähere Betrachtung, wie sich Informationsmodell, Organisationsmodell, Kommunikationsmodell und Funktionsmodell im Fall von JMAPI darstellen.

Schließlich werden die Ergebnisse in einer abschließenden Bewertung zusammengefaßt.

8.1 Anforderungen an integriertes Management

Nach [HA93] sollte das Ziel eines integrierten Managements folgenden Anforderungen genügen:

- Integration von Architekturen bzw. System- und Netztypen
- Integration von Management-Funktionsbereichen
- Integration von Organisationsaspekten (Domänenkonzept)
- Einheitliches Konzept für eine Management Datenbasis
- Weitgehend vereinheitlichte Konzepte für Netz- und Systemmanagement
- Unterstützung verteilter Anwendungen und verteilter Systeme
- Einheitliche Progammierschnittstellen und Bedienoberflächen.

Um diesen Anforderungen gerecht zu werden, muß jede Netzmanagement-Architektur, die für ein integriertes Management in heterogener Umgebung geeignet sein soll, folgende Teilmodelle berücksichtigen [HA93]:

- Informationsmodell (Beschreibung von Managementobjekten)

- Organisationsmodell (Behandlung und Unterstützung von Organisationsaspekten)

- Kommunikationsmodell (Beschreibung von Kommunikationsvorgängen zu Managementzwecken)

- Funktionsmodell (Strukturierung von Managementaufgaben)

In den folgenden Abschnitten wird aufgezeigt, inwieweit die Ausprägungen dieser Teilmodelle im Falle von JMAPI den Anforderungen an ein integriertes Management Rechnung tragen.

8.2 Informationsmodell

Das Informationsmodell befaßt sich mit der Modellierung der managementrelevanten Aspekte der realen Welt. Hinsichtlich der Integration mit anderen Managementarchitekturen spielt es eine wichtige Rolle, da es festlegt, welche Information überhaupt verfügbar ist, wie auf diese Information zugegriffen werden kann und ob und in welcher Weise die Information veränderbar ist.

Die angestrebte Verwendung von CIM als Basis des JMAPI-Informationsmodells ist aus folgenden Gründen positiv zu werten:

- CIM stellt einen herstellerübergreifenden Standard dar, der über die DMTF von einer Vielzahl von Firmen unterstützt wird. Dieser Aspekt ist wichtig, da es die Integration zukünftig entwickelter Management-Lösungen anderer Hersteller, die sich ebenfalls auf CIM abstützen, erleichtert.

- WBEM, als ebenfalls im Bereich des Web-based Management in der Entwicklung befindliches Produkt, legt ebenfalls CIM als Informationsmodell zugrunde. Somit stellt WBEM kein Konkurrenzprodukt dar, das andere Standards verwendet und u. U. inkompatible Managementanwendungen schafft.

- CIM ist ein objektorientiertes Modell und kann dadurch die Vorteile eines derartigen Ansatzes gegenüber Datentyp-basierten Modellen, welche etwa dem SNMP-Management oder dem DMI-Modell der DMTF zugrundeliegen, nutzen. Insbesondere können neue Managed-Object-Klassen durch Spezialisierung bereits existierender Klassen erzeugt werden. Die CIM-Schemata stellen eine Reihe generischer Basisklassen zur Verfügung, aus denen speziellere Klassen durch Verfeinerung gewonnen werden können. Managementanwendungen, die auf der Basis der allgemeinen Klassen entwickelt wurden, bleiben auch für die spezielleren Klassen einsetzbar.

- CIM sieht die Integration bereits existierender Modelle, wie z. B. DMI oder SNMP, mittels sog. *Mappings* vor (siehe [Des98]). Dies erleichtert die Integration von SNMP- und DMI-basierten Systemen.

Die CIM-Schemata stellen generische Klassen sowohl für den Bereich des Netz- als auch des Systemmanagements bereit. Damit wird auch die oben aufgestellte Forderung nach einheitlichen Konzepten für das Management dieser beiden Gebiete berücksichtigt.

Eine genaue Analyse von CIM geht über den Umfang dieser Arbeit hinaus und soll daher an dieser Stelle nicht vorgenommen werden.

8.3 Organisationsmodell

Das Organisationsmodell ist maßgeblich für die Flexibilität bei der Anpassung eines Managementsystems an strukturelle Entscheidungen und räumliche Gegebenheiten sowie für die Möglichkeit der Zuordnung von Managementobjekten zu Zuständigkeitsbereichen. Zu diesen Zwecken legt es Rollen und ein Domänenkonzept fest.

Wie in Abschnitt 2.2.1 beschrieben, kennt JMAPI die drei Rollen Client, Managed Object Server und Agent. Diese Rollenverteilung resultiert aus einem zentralisierten Management-Ansatz, bei dem der Managed Object Server die Steuerung der in ihrem Funktionsumfang beschränkten Agenten übernimmt (siehe hierzu auch die Gegenüberstellung mit dem Java Dynamic Management Kit in Abschnitt 6.4).

Der hauptsächliche Vorteil einer zentralen Architektur besteht in der einfacheren Wartung. Die wesentlichen Komponenten, wie die Datenbank, der Web-Server und die von den Agenten zu ladende Software, sind auf dem Managed Object Server konzentriert.

Andererseits ist das Vorhandensein einer zentralen Komponente auch mit Nachteilen verbunden. Zum einen ist das System im Hinblick auf Wachstum sehr unflexibel. Die gesamte Kommunikation der Teilkomponenten läuft über den MO Server, eine Kommunikation der Agenten untereinander ist nicht vorgesehen. Bei einer entsprechend großen Zahl von Agenten wird dieser somit schnell zum Engpaß. Außerdem leidet auch die Robustheit des Managementsystems. Mit dem Ausfall des MO Servers kommt das gesamte System zum Erliegen.

Während es in [JMA97] nur möglich ist, einen einzelnen MO Server einzusetzen, wird in [Gio98b] ein erweitertes Architekturmodell vorgestellt, das zukünftigen JMAPI-Versionen zugrunde liegen soll (vgl. Abbildung 8.1). Dieses Modell sieht die Möglichkeit der Verwendung mehrerer MO Server vor, die untereinander mittels RMI kommunizieren sollen. Hierdurch würden die oben beschriebenen Nachteile eines zentralisierten Ansatzes teilweise aufgehoben. Einzelheiten zu diesem erweiterten Architekturmodell waren zum Zeitpunkt der Fertigstellung dieser Arbeit noch nicht erhältlich.

Aufbau- oder ablauforganisatorische Gründe machen oft die Gruppenbildung von Ressourcen erforderlich. Einige Managementarchitekturen, wie beispielsweise das OSI-Management (vgl. [HA93]), sehen hierzu die Bildung von Domänen vor. JMAPI bietet die in 4.2 geschilderte Möglichkeit der Gruppierung von Managed Objects. Hiermit lassen sich beispielsweise Teilbereiche nach verschiedenen Aspekten, wie etwa Abrechnung und Sicherheit, bilden.

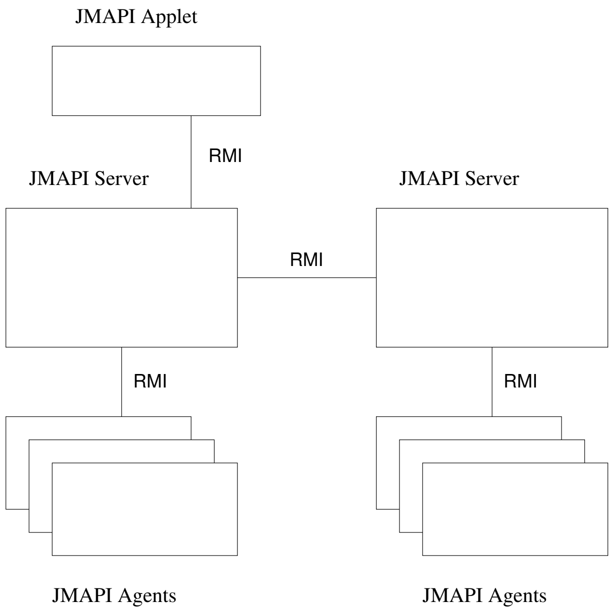


Abbildung 8.1: Mehrere Managed Object Server (geplant)

4.2

8.4 Kommunikationsmodell

Aufgabe des Kommunikationsmodells ist die Festlegung der kommunizierenden Partner und des Kommunikationsmechanismus zur Durchführung der Überwachungs- und Steuerungsaufgaben.

JMAPI führt kein neues Kommunikationsprotokoll ein, sondern stützt sich auf bereits vorhandene Protokolle ab. Im wesentlichen kommt Java RMI zum Einsatz, nur für die Kommunikation mit nicht-Java-basierten Agenten muß auf andere Kommunikationsprotokolle ausgewichen werden. Die Verwendung existierender, bereits etablierter Protokolle erhöht die Akzeptanz JMAPI-basierter Lösungen und vermeidet Schwierigkeiten, die mit der Einführung neuer Protokolle verbunden sind[1]. Ferner wird durch die Verwendung bereits existierender Protokolle die Integration von Agenten anderer Managementarchitekturen, wie etwa SNMP- oder CORBA-Agenten, erleichtert.

Die Verwendung von anderen Kommunikationsprotokollen als RMI ist allerdings mit dem Nachteil verbunden, daß in diesem Fall der in JMAPI integrierte Transaktionsmechanismus (vgl. Abschnitt 6.6) nicht vollständig genutzt werden

[1] Als Beispiel einer gescheiterten Einführung eines neuen Protokolls kann das Hyper Media Management Protocol (HMMP) genannt werden. HMMP war Teil der von Microsoft vorgeschlagenen Architektur von WBEM und sollte als Internet Draft an eine der IETF Working Groups übergeben werden. HMMP konnte innerhalb der IETF jedoch keine große Anhängerschaft gewinnen [Gio98a]. Seit der Übernahme von WBEM durch die DMTF wird die Weiterentwicklung von HMMP nicht mehr betrieben, die zugehörigen Dokumente zu HMMP sind von der Web-Seite `http://wbem.freerange.com` nicht mehr erhältlich.

kann. Als Beispiel für die hier auftretenden Schwierigkeiten betrachte man einen Client, der auf einem Managed Object Attributwerte verändert, die Aktionen auf einem Agenten nach sich ziehen sollen. Führt nun der Client ein Commit der entsprechenden Transaktion durch, so wird versucht, die veränderten Attributwerte in die Datenbank zu übernehmen. Wird hierbei festgestellt, daß zur gleichen Zeit ein anderer Client ebenfalls Modifikationen am gleichen MO durchgeführt hat, so wird von JMAPI automatisch ein Rollback der Transaktion veranlaßt[2]. Ein Zurücksetzen der alten Attributwerte ist unproblematisch, da dies bei einem Rollback vom DBMS automatisch durchgeführt wird. Schwierigkeiten bereitet in diesem Zusammenhang die Frage, was mit den Änderungen auf dem Agenten geschehen soll. Bei Verwendung von RMI kann der in Abschnitt 6.6 geschilderte Mechanismus über `CommitAndRollback`-Objekt und Callback Funktionen genutzt werden. Dieser würde dann in transparenter Weise einen Rollback veranlassen. Verwendet man allerdings ein anderes Kommunikationsmodell, bietet sich diese Möglichkeit nicht. Da keine Methode des MOs *nach* der Datenbanktransaktion ausgeführt wird, müßte der Client überprüfen, ob sich die Attributwerte tatsächlich geändert haben, um so festzustellen, ob die Transaktion erfolgreich war, und gegebenenfalls eine Rollback-Methode des MOs aufrufen. Wie jedoch unmittelbar einsichtig ist, wäre diese Vorgehensweise sehr problematisch, zieht man die Möglichkeit von Race-Conditions durch Modifikationen anderer Clients mit in Betracht.

8.5 Funktionsmodell

Das Funktionsmodell einer Managementarchitektur nimmt eine Aufteilung der Managementaufgaben in verschiedene Management-Funktionsbereiche, wie etwa Konfigurations-, Abrechnungs-, Sicherheit-, Fehler- und Leistungsmanagement, vor und versucht dann, für die einzelnen Teilbereiche generische Managementfunktionen festzulegen.

Das Funktionsmodell findet bei JMAPI keinerlei Ausprägung. Dies steht im Zusammenhang mit der Tatsache, daß es sich bei JMAPI um keine vollständige Management-Plattform handelt (siehe Abschnitt 2.4).

8.6 Abschließende Bewertung

JMAPI kann als Java-basiertes System die Vorteile der Plattformunabhängigkeit von Java nutzen, die für den Einsatz in heterogenen Systemumgebungen relevant ist. Vor allem für den Bereich der Oberflächenintegration könnte sich JMAPI nach seiner Fertigstellung als wertvoll erweisen.

Im Hinblick auf das integrierte Management bietet JMAPI die Möglichkeit, Agenten anderer Managementarchitekturen in das JMAPI-Framework zu integrieren. Für SNMP-Agenten wird dies durch eine mitgelieferte SNMP-Klassenbibliothek unterstützt. Die Einbindung von CORBA oder JDMK-

[2] Wie bereits erwähnt, verwendet JMAPI *Optimistic Concurrency Control*, d.h. es werden keine Sperren gesetzt, wie dies bei Verwendung von *Lock Based Concurrency Control* der Fall wäre.

Agenten ist jedoch auch möglich. Vergleicht man den Einsatz von JMAPI-Agenten mit dem von Agenten anderer Managementarchitekturen, so sind folgende Punkte von Bedeutung:

- Bei der Verwendung von Fremdagenten, die nicht von der JMAPI Agent Objct Factory verwaltet werden, besteht zunächst keine Unterstützung für die Mehrbenutzersynchronisation mehr, wie sie im Falle von JMAPI-Agenten durch den integrierten Transaktionsmechanismus bereitgestellt wird. Als möglichen Ausweg könnte man einen JMAPI-Agenten als Proxy-Agenten einsetzen (vgl. Abschnitt 6.3).

- JMAPI-Agenten haben gegenüber Agenten anderer Managementarchitekturen den Vorteil der zentralen Wartungsmöglichkeit vom Managed Object Server aus. Appliances laden bei Bedarf immer die jeweils neueste Klasse vom MO Server. Ein Versions-Update eines Agenten ist nur auf dem Server vorzunehmen und wird von dort aus automatisch auf die Agenten übertragen.

Hinsichtlich der Flexibilität und Robustheit des Systems ist der von JMAPI verfolgte zentrale Managementansatz als negativ anzusehen.

Mit zunehmender Größe des Systems wäre der Einsatz eines Naming Service sinnvoll, da beispielsweise Host-Namen/IP-Adressen und Port-Nummern sowohl des Managed Object Servers, der Agent Object Factory und der Event Dispatcher bekannt sein müssen. Auf Anfrage war von Sun zu erfahren, daß der Einsatz eines Naming Service nicht geplant ist (vgl. Anhang E).

Aufgrund obiger Flexibilitäts- und Skalierbarkeitsbetrachtungen wird sich das Einsatzgebiet von JMAPI vermutlich auf kleine bis mittelgroße Systeme beschränken. Als mögliches Einsatzszenario ist etwa die Verwaltung eines heterogenen Workstation-Clusters denkbar, bei dem die Workstations die Rolle der Appliances spielen und mittels JMAPI-Agenten gemanaged werden. Andere Geräte, wie z. B. Drucker, könnten über SNMP-Agenten in die Managementumgebung integriert werden, soweit für sie noch keine Java Virtual Machines zum Ausführen der JMAPI-Agenten-Software zur Verfügung stehen.

Für eine detailliertere Bewertung bleibt die Veröffentlichung der JMAPI-Spezifikation abzuwarten.

Kapitel 9

Erfahrungen im Umgang mit der Referenzimplementierung

In diesem Kapitel soll ein kurzer Erfahrungsbericht über die Arbeit mit der JMAPI-Referenzimplementierung gegeben werden. Es wurde ausschließlich mit der Solaris-Version gearbeitet, obwohl es auch eine Version für Windows NT gibt.

9.1 Installation und Konfiguration

Die Software ist als komprimiertes `tar`-Archiv von folgender URL erhältlich: `http://java.sun.com/products/JavaManagement`. Beim Entpacken stellte sich bemerkenswerterweise heraus, daß `gtar` (GNU-tar) mit den Optionen `-zxf` nicht dasselbe Ergebnis lieferte, wie die von Sun vorgeschlagene Verwendung von `/usr/bin/tar` in Kombination mit `uncompress`. `gtar` entpackt einige Dateien an die falsche Stelle[1].

Die Installation unter Solaris erfolgt einfach durch Einspielen des Packages `SUNWjmapi` mit dem `pkgadd` Kommando. Das Installationsverzeichnis ist `/opt/JMAPI`, deswegen sind für die Installation `root`-Rechte erforderlich.

Der Developer's Release 0.5 erfordert für die Nutzung der Managed-Object-Server-Funktionalität eine Datenbank. Ohne eine Datenbank können im wesentlichen nur die AVM-Klassen für die GUI-Entwicklung, sowie die SNMP-Klassen genutzt werden. Aus diesem Grund war es notwendig, eine geeignete Datenbank auszuwählen. Naheliegend schien zunächst die Verwendung einer frei verfügbaren Public-Domain-Datenbank, wie etwa msql, mysql oder Postgres. Zur Anbindung an JMAPI wird für diese Datenbanken ein JDBC-Treiber benötigt, welcher in allen Fällen auch vorhanden war. Wie sich jedoch beim anschließenden Setup mit JMAPI herausstellte, erfüllte keine der drei einfachen Datenbanken die von JMAPI gestellten Anforderungen. Aus diesem Grund wurde auf eine am Lehrstuhl vorhandene Sybase 11 Datenbank ausgewichen. Als JDBC-Treiber wurde der von Sun mit JMAPI erfolgreich getestete FastForward-Treiber von Connect Software ausgewählt (URL zum Download:

[1] Sun verwendet offensichtlich nicht `gtar` zum Archivieren.

http://www.connectsw.com).

Die Installation des JDBC-Treibers beschränkt sich im wesentlichen auf das Entpacken des zip-Archivs (release300.zip) und dem Anpassen der CLASSPATH-Environment-Variablen, damit diese die Treiberklassen enthält. Als Installationsverzeichnis für den Treiber wurde /proj/java/JMAPI/jdbcDriver gewählt. Als Pfad für den CLASSPATH ist /proj/java/JMAPI/jdbcDriver/FastForward/jdk113 anzugeben.

Als nächster Schritt ist die Konfiguration vorzunehmen. Die Konfigurationsdaten werden normalerweise in der Datei /opt/JMAPI/classes/properties eingetragen. Alternativ kann auch eine andere Datei gewählt werden. In diesem Fall muß die Environment Variable JMAPI_PROPERTIES auf den entsprechenden Pfadnamen gesetzt werden. Anhang D zeigt ein Beispiel für eine derartige Datei.

Variable	Wert
JAVA_HOME	/proj/java/jdk1.1-solaris/jdk1.1.6
JMAPI_HOME	/opt/JMAPI
PATH	${JMAPI_HOME}/bin:${JAVA_HOME}/bin:${PATH}
CLASSPATH	/proj/java/JMAPI/jdbcDriver/FastForward/ jdk113:${JMAPI_HOME}/classes:${JMAPI_HOME}/ examples:${CLASSPATH}
JMAPI_PROPERTIES	${JMAPI_HOME}/classes/properties

Tabelle 9.1: Zu setzende Environment-Variablen mit Beispielwerten

Ferner sind die anderen in Tabelle 9.1 aufgeführten Environment Variablen zu setzen.

Die Konfigurationsdatei enthält neben Angaben über Pfadnamen verschiedener von JMAPI verwendeter Dateien die beim Hochfahren des Servers zu startenden Dienste. Hierbei stehen die Managed Object Factory, die Agent Object Factory und der Net Class Loader zur Auswahl. Ferner enthält die Konfigurationsdatei Angaben über die von den Diensten zu verwendenden Portadressen. Schließlich finden sich auch noch die Konfigurationsdaten der Datenbank in der Datei.

Nun erfolgt der Aufruf des Skripts /opt/JMAPI/bin/setupdb. Dieses legt die für die Base Managed Object Klassen nötigen Schemata in der Datenbank an.

9.2 Starten und Anhalten von JMAPI

Das Starten und Anhalten von JMAPI erfolgt auf jedem beteiligten Host[2] mit Hilfe der Kommandos /opt/JMAPI/bin/startjmapi und /opt/JMAPI/bin/stopjmapi. Hierbei handelt es sich um Shell-Skripte, die die in der Konfigurationsdatei aufgelisteten Dienste starten. Für den Server sollte

[2]Die JMAPI-Software muß sowohl auf dem Server, als auch auf den Agenten installiert werden.

dies zumindest der Managed Object Server, für die Appliances entsprechend die Agent Object Factory sein.

9.3 Erzeugen neuer MO-Klassen

Nach dem Erstellen der `.mo`-Datei ist der Managed Object Compiler `Moco` aufzurufen. Für ein Übersetzen, mit anschließendem automatischen Importieren in die Datenbank, verwendet man den Aufruf:

```
java -DJMAPI_PROPERTIES=${JMAPI_PROPERTIES} moco.Moco -verbose
    -noprismvar -autoimport NewMOClass.mo
```

Sollen die temporär angelegten `.java`-Dateien nicht gelöscht werden, kann man noch die Option `-keeptmp` angeben. Beim Ausführen obigen Kommandos ist, wie immer, darauf zu achten, daß die `CLASSPATH`-Variable richtig gesetzt ist und das Verzeichnis `/opt/JMAPI/classes` enthält.

Weiterhin ist zu beachten, daß beim Importieren neuer MO-Klassen der MO Server gemäß Hinweisen von Sun nicht laufen sollte. Aufgrund der vorhandenen Ein-Benutzerlizenz für den JDBC-Treiber erfolgt bei einem Versuch, trotz laufendem Server in die Datenbank zu importieren, allerdings sowieso eine Fehlermeldung, daß dies aus Lizenzgründen nicht möglich ist.

9.4 Dokumentation

Die bei der Installation lokal abgelegte Dokumentation findet sich im Verzeichnis `/opt/JMAPI/doc` in Form von HTML- und PDF-Dateien. Für die Programmierung unentbehrlich ist die HTML-Dokumentation im `javadoc`-Format.

Wichtiger Hinweis:

Die *lokal* installierte Dokumentation im `javadoc`-Format ist momentan *aktueller* als die Dokumentation auf der Web-Page von Sun, wo sich die Dokumentation zu einer alten Version findet.

Kapitel 10

Zusammenfassung und Ausblick

Das Gebiet des Web-Based Management ist noch relativ neu. Einige Hersteller bieten für das Komponentenmanagement bereits seit einiger Zeit den Zugang über eine Web-Schnittstelle an (vgl. [Lor98]). Ein allgemeiner Rahmen für die Entwicklung von Software im Bereich des Web-basierten Netz-, System- und Anwendungsmanagements existiert jedoch noch nicht.

Mit JMAPI unternehmen Sun Microsystems und JavaSoft den Versuch, mittels eines Java-basierten Frameworks diese Lücke zu füllen. Die Plattformunabhängigkeit von Java in Verbindung mit der Fähigkeit des dynamischen Ladens ausführbaren Codes über das Netz lassen die Verwendung von Java als Basis verteilter Managementanwendungen als vorteilhaft erscheinen.

In dieser Diplomarbeit wurde die Eignung von JMAPI im Hinblick auf Einsatzmöglichkeiten für das integrierte Management untersucht. Die Bewertung stützte sich auf die vier Teilmodelle des Managements, das Informationsmodell, Organisationsmodell, Kommunikationsmodell und das Funktionsmodell.

Das Informationsmodell von JMAPI basiert auf dem Common Information Model der DMTF. Die Verwendung dieses herstellerübergreifenden Standards, dem ein objektorientierter Ansatz zugrundeliegt und der im Hinblick auf die Integration anderer Informationsmodelle wie SNMP oder DMI entwickelt wurde, erleichtert die Integration anderer Managementarchitekturen. Im Rahmen der Diplomarbeit wurde mittels einer Reihe von `Perl`-Skripten eine Möglichkeit erarbeitet, eine CIM-Beschreibung von Objektklassen im Managed Object Format (MOF) in eine JMAPI-konforme Beschreibung zu transformieren und somit für JMAPI nutzbar zu machen. Ferner wurde für die Konstruktion von Gateways gezeigt, wie sich CIM-Associations auf den Topology Service der OMG abbilden lassen.

Das Organisationsmodell unterscheidet die drei Rollen Client, Managed Object Server und Agent. Der MO Server bildet die zentrale Komponente der Architektur, und beinhaltet eine Datenbank sowie einen Web Server. Er agiert als Bindeglied zwischen den Client-Applets und den Agenten, die die eigentliche Management-Funktionalität implementieren. Das Vorhandensein einer zentralen Komponente erleichtert einerseits die Wartung, ist andererseits jedoch im

Hinblick auf Skalierbarkeit und Fehlertoleranz mit Nachteilen verbunden. Hinsichtlich der Integration anderer Managementarchitekturen ist insbesondere die Frage nach der möglichen Einbeziehung von JMAPI-fremden Agenten von Bedeutung. Im Rahmen dieser Arbeit wurde speziell der Einsatz von CORBA-Agenten und von Agenten, die mit dem ebenfalls von Sun Microsystems entwickelten Java Dynamic Management Kit (JDMK) entwickelt wurden, untersucht. Als Ergebnis läßt sich festhalten, daß für das Ziel der Oberflächenintegration eine Einbeziehung von Fremdagenten zwar sinnvoll und möglich ist, daß andererseits jedoch im Falle von Fremdagenten ein Teil der von JMAPI bereitgestellten Funktionalität nicht genutzt werden kann. So kann z. B. der in JMAPI integrierte Transaktionsmechanismus für die Mehrbenutzersynchronisation nicht mehr eingesetzt werden. In diesem Fall kann es dann u.U. zu Inkonsistenzen zwischen dem Zustand der MOs auf dem Server und den Agenten kommen. Ein weiteres Beispiel nicht-nutzbarer Funktionalität im Falle von JMAPI-fremden Agenten ist die mangelnde Fähigkeit, automatische Versions-Updates der Agenten-Software durchzuführen. Ferner ist das von JMAPI vorausgesetzte Organisationsmodell nicht für alle Agenten sinnvoll. Beispielsweise widerspricht der zentralisierte Managementansatz von JMAPI dem Konzept weitgehend autonomer Agenten, das der JDMK-Architektur zugrundeliegt. Als Beispiel zur Oberflächenintegration wurde im Rahmen dieser Diplomarbeit für einen am Lehrstuhl vorhandenen CORBA-Agenten eine JMAPI-konforme Oberfläche erstellt.

Das JMAPI-Kommunikationsmodell führt kein neues Kommunikationsprotokoll ein. Es kommt im wesentlichen Java RMI zum Einsatz. Alternativ können für die Kommunikation des MO Servers mit den Agenten auch andere Kommunikationsprotokolle, wie CORBA IIOP oder SNMP verwendet werden. Die Verwendung von RMI hat den zusätzlichen Vorteil, daß der im RMI integrierte Sicherheitsmechanismus genutzt werden kann.

Das Funktionsmodell findet bei JMAPI keinerlei Ausprägung.

Für zukünftige Versionen von JMAPI sind folgende Entwicklungen zu erwarten:

- Zum einen sollen die mit JMAPI mitgelieferten Managed-Object-Basisklassen der zur Zeit aktuellen CIM Version 2.0 angepaßt werden. Ein Konvertieren der CIM MOF-Dateien ins JMAPI `.mo`-Format wäre dann nur noch für neu hinzukommende hersteller- oder technologiespezifische CIM Extension Schemas erforderlich. Wünschenswert wäre in diesem Zusammenhang ein mitgelieferter Compiler, der die entsprechende Umsetzung von MOF-Dateien ins `.mo`-Format vornimmt.

- Ab der nächsten Version ist die Möglichkeit des Einsatzes mehrer Managed Object Server, die untereinander mittels Java RMI kommunizieren, angekündigt. Dies würde sich positiv auf die Flexibilität und Robustheit des Systems auswirken.

- Nach [Gio98b] bestehen Pläne, JMAPI Managed Objects zu JavaBeans-Komponenten zu machen. Dies würde die Vorteile der Komponententechnologie auch für JMAPI nutzbar machen, wie dies bereits bei JDMK der Fall ist.

Wünschenswert wären vor allem folgende Erweiterungen:

- Ein Naming Service würde sich positiv bemerkbar machen, da bisher, vor allem bei der Behandlung von Events, Host Namen und Port Adressen oft explizit aufgeführt werden müssen.

- In Hinblick auf Integration mit anderen Managementarchitekturen wäre es vorteilhaft, den in JMAPI integrierten Transaktionsmechanismus auch auf JMAPI-fremde Agenten, wie z. B. CORBA-Agenten, auszudehnen.

Die JMAPI-Spezifikation durchläuft zur Zeit noch den JavaSoft-Software-Review-Prozeß, der einer Veröffentlichung vorausgeht. Nach Informationen von der JMAPI-Mailing-Liste [Mai] ist die Spezifikation noch im Sommer 1998 zu erwarten (vgl. Anhang E). Zu diesem Zeitpunkt ist vermutlich auch mit einer neuen Version der Referenzimplementierung zu rechnen. Mit dem Erscheinen der Spezifikation sollte es möglich sein, einige der noch offenen Fragen, vor allem im Zusammenhang mit dem Einsatz mehrerer Managed Object Server, zu beantworten.

Anhang A

Konvertieren von MOF- in .mo-Dateien

A.1 Datei pass1.pl

```
#!/usr/bin/perl -w
#
# Converting MOF-files to .mo-files, Step 1
#
# Author: Christian Schiller
# Date:   08/15/98
#

# output filename
$filename = "_$ARGV[0]";

print STDERR "output will be written to $filename\n";
open(OUTPUT, ">$filename");


$squarebracket = 0;  # indicates whether square brackets
                     # have been opened but not closed yet

$qualifier = 0;      # indicates that a qualifier definition is
                     # spanning multiple lines

$abstract = 0;       # indicates whether an abstract
                     # qualifier is active

$association = 0;    # indicates whether an association qualifier
                     # is active


while (<>) {

    # skip comments and pragmas
    if (/\/\/|^#/) {
next;
    }

    # skip qualifier definitions
    if (/^Qualifier/i) {
if (/;/) {
    next;
} else {
    $qualifier = 1;
}
```

```
while ($qualifier) {
    $_ = <>;
    /;/ && do { $qualifier = 0; }
}
next;
    }

    # skip qualifiers
    if (/\[/) {
$abstract = 0;
/abstract/i && do { $abstract = 1; };
/Association/ && do { $association = 1};
if (/\[.*\]/) {
    next;
} else {
    $squarebracket = 1;
}
while ($squarebracket){
    $_ = <>;
    /abstract/i && do { $abstract = 1; };
    /Association/ && do { $association = 1};
    /\]/ && do { $squarebracket = 0; }
}
next;
    }

    # skip associations
    if ($association) {
while(! eof) {
    /}\s*;/ && last;
    $_ = <>;
}
    }
    if ($association) {
$association = 0;
next;
    }

    # mark abstract classes
    if ($abstract) {
s/class/abstract class/;
    }

    # add 'public abstract' in front of methods;
    # methods are recognized by scanning for '()'
    s/(\s*)(\w.*)\(\);/$1public abstract $2\(\);/;

    print OUTPUT;
}
```

A.2 Datei pass2.pl

```
#!/usr/bin/perl -w
#
# Converting MOF-files to .mo-files, Step 2
#
# Author: Christian Schiller
# Date:   08/15/98
#

# output filename
$_ = $ARGV[0];
/(_\w+)\.mof/i;
$filename = $1 . ".mo";
```

```
print STDERR "output will be written to $filename\n";
open (OUTPUT, ">$filename");

while(<>) {

    if (/class.*:/) {
# defined class extends another class
s/(.*)class\s+(\w+):(\w+)/public $1class $2Impl extends $3Impl implements $2/i;
    } else {
# defined class  extends ManagedObject
s/(.*)class\s+(\w+)/public $1class $2Impl extends ManagedObjectImpl implements $2/i;
    }

    # convert data types
    s/\bbool\b/boolean/i;
    s/\bstring\b/String/i;
    s/\buint8\b/short/i;
    s/\bsint8\b/byte/i;
    s/\buint16\b/int/i;
    s/\bsint16\b/short/i;
    s/\buint32\b/long/i;
    s/\bsint32\b/int/i;
    s/\buint64\b/BigInteger/i;
    s/\bsint64\b/long/i;
    s/\breal32\b/float/i;
    s/\breal64\b/double/i;
    s/\bchar16\b/char/i;
    s/\bdatetime\b/String/i;

    # no semicolon after '}'
    s/\}(\s)*;/\}/;

    print OUTPUT;
}
```

A.3 Datei pass3.pl

```
#!/usr/bin/perl -w
#
# Converting MOF-files to .mo-files, Step 3
#
# Author: Christian Schiller
# Date:   08/15/98
#

%objtype = (
   "boolean" => "Boolean",
   "char" => "Character",
   "byte" => "Byte",
   "short" => "Short",
   "int" => "Integer",
   "long" => "Long",
   "float" => "Float",
   "double" => "Double"
    );

# output filename
$_ = $ARGV[0];
/_(\w+)\.mo/i;
$filename = $1 . ".mo";

print STDERR "output will be written to $filename\n";

open (OUTPUT, ">$filename");
```

```
while(<>) {

    # copy everything to the output
    print OUTPUT;

    # add get/set funcions for the attributes
  SWITCH: {
      # extract the class name and add appropriate constructor
      /class\s+(\w+)Impl/ && do { $classname = $1; &addConstructor(); };

      # add get/set functions for reference types
      /String\s+(\w+);/ && do { &getSetRef("String", $1); last SWITCH };
      /BigInteger\s+(\w+);/ && do {&getSetRef("BigInteger", $1); last SWITCH };

      # add get/set functions for  primitive types
      /boolean\s+(\w+);/ && do { &getSetPrim("boolean", $1); last SWITCH };
      /char\s+(\w+);/ && do { &getSetPrim("char", $1); last SWITCH };
      /byte\s+(\w+);/ && do { &getSetPrim("byte", $1); last SWITCH };
      /short\s+(\w+);/ && do { &getSetPrim("short", $1); last SWITCH };
      /int\s+(\w+);/ && do { &getSetPrim("int", $1); last SWITCH };
      /long\s+(\w+);/ && do { &getSetPrim("long", $1); last SWITCH };
      /float\s+(\w+);/ && do { &getSetPrim("float", $1); last SWITCH };
      /double\s+(\w+);/ && do { &getSetPrim("double", $1); last SWITCH };

  }
}

sub addConstructor {
    # read until opening '{', then add constructor
    while (<>) {
/\{/ || do { print OUTPUT; next; };
last;
    }
    print OUTPUT;                   # '{'
    print OUTPUT <<HERE;
    public ${classname}Impl() throws RemoteException {
    }
HERE
    print OUTPUT "\n";
}

# for reference types
sub getSetRef {
    my ($type, $attrib) = @_;

    print OUTPUT <<HERE;
    public $type getAttr$attrib(Session session)
        throws AuthorizationException, RemoteException
    {
    return ($type) getPropertyByName(session, \"$attrib\");
    }
    public void setAttr$attrib(Session session, $type $attrib)
        throws NotInUpdateException, AuthorizationException, RemoteException
    {
            setPropertyByName(session, \"$attrib\", $attrib);
    }
HERE
}

# for primitive types
sub getSetPrim {
    my ($type, $attrib) = @_;

    print OUTPUT <<HERE;
    public $type getAttr$attrib(Session session)
        throws AuthorizationException, RemoteException
```

```
    {
    return (($objtype{$type}) getPropertyByName(session, \"$attrib\")).${type}Value();
    }
    public void setAttr$attrib(Session session, $type $attrib)
        throws NotInUpdateException, AuthorizationException, RemoteException
    {
            setPropertyByName(session, \"$attrib\", new $objtype{$type}($attrib));
    }
HERE
}
```

A.4 Datei break-it-up.pl

```
#!/usr/bin/perl -w
#
# Converting MOF-files to .mo-files, Step 4
#
# Author: Christian Schiller
# Date:   08/15/98
#

while (<>) {
    /^public/ && last;
}

while(!eof()) {

    /class\s+(\w+)Impl\s+/ && do { $basename = $1; };
    $filename = "schiller/jmapi/cim/" . $basename . ".mo";

    open (OUTPUT, ">$filename") || die "can't open $filename";

    print OUTPUT <<HERE;
package schiller.jmapi.cim;

import schiller.jmapi.cim.*;

import java.io.*;
import java.net.*;
import java.util.*;
import java.rmi.*;
import java.math.BigInteger;

import jrb.java.util.*;

import sunw.admin.arm.manager.*;
import sunw.admin.arm.common.*;

HERE

    print OUTPUT;

    while (<>) {
/^public/ && last;
print OUTPUT;
    }

    close OUTPUT;
    print STDERR "wrote $filename\n";
}
```

Anhang B

Code für die Oberfläche des CORBA-Agenten

B.1 Erläuterung

Der Code besteht aus zwei Java-Klassen, die beide in einer Quelldatei zusammengefaßt sind. Die beiden Klassen beschreiben ein Applet und einen Content Manager. Das `AccountContentManagerApplet` wickelt die Kommunikation mit der CORBA-`AccountFactory` ab und speichert die ausgelesenen Daten ab. Ferner dient es als Rahmen für den Content Manager, welcher zur Darstellung der Account Daten eingesetzt wird. Der Content Manager wird durch die Klasse `AccountContentMgr` definiert.

Das vorliegende Applet gestattet nur die Anzeige der Benutzerdaten, nicht jedoch das Löschen oder Neuanlegen von Benutzern. Zum einen hat dies seinen Grund in der Tatsache, daß hier nur an einem einfachen Beispiel das prinzipielle Vorgehen vorgeführt werden soll. Zum anderen ist die Funktionalität des Anlegens oder Löschens von Benutzern durch den CORBA-Agenten noch nicht vollständig implementiert.

B.2 Datei `AccountContentManagerApplet.java`

```
/*
 * AccountContentManagerApplet.java
 *
 */

import java.applet.*;
import java.awt.*;
import java.net.*;
import java.util.*;
import sunw.admin.avm.base.*;

public class AccountContentManagerApplet extends Applet {
    TopLevelContentManager mgr;
    ContentManager users;
    Image userImg;

    org.omg.CORBA.ORB orb;
    sysagent.AccountFactory accountFactory;
    sysagent.Account accounts[];

    Vector userVals = new Vector();

    public void init() {
        System.err.println("Initializing applet.");
```

```
        // initialize the ORB
        System.err.print("Initializing ORB ...");
        orb = org.omg.CORBA.ORB.init(this);
        System.err.println("o.k.");
        // get a reference for the account factory
        accountFactory = sysagent.AccountFactoryHelper.bind(orb,
                                                          "AccountFactory");
        accounts = accountFactory.list();
        System.err.println("Bound to account factory.");
        for(int i=0; i < accounts.length; i++) {

            System.err.println("account.Login() = " + accounts[i].Login());

            Vector accountVector = new Vector();
            accountVector.addElement(accounts[i].Login());
            accountVector.addElement(accounts[i].Password());
            accountVector.addElement(new Long(accounts[i].ID()).toString());
            accountVector.addElement(new Long(accounts[i].GID()).toString());
            accountVector.addElement(accounts[i].Comment());
            accountVector.addElement(accounts[i].Home());
            accountVector.addElement(accounts[i].Shell());
            userVals.addElement(accountVector);
        }

        System.err.println("Done collecting account data.");
        setLayout(new BorderLayout());
        Context.setProperty("headerBackground", new Color(70,70,170));
        Context.setProperty("headerForeground", Color.white);
        try {
            userImg = Context.getImage("images/user.gif", this);
        }
        catch (MalformedURLException e) {
            System.out.println("Invalid URL");
        }
        add("Center", mgr = new TopLevelContentManager());
        mgr.add("Center", users =
                new AccountContentMgr("Users", userImg, userVals));
        users.addItemListener(mgr);

        mgr.select(users);
    }
}




class AccountContentMgr extends SimpleContentManager {
        static private Command[] listCmds = new Command[4];
        static {
                listCmds[0] = new PropertiesCommand();
                listCmds[1] = new FilterCommand();
                listCmds[2] = new SortCommand();
                listCmds[3] = new DisplayCommand();
        }


        public AccountContentMgr(String title, Image img, Vector userVals) {
                super(title, img);

                // Set Table parameters
                Table table = getTable();
                table.setNumColumns(7);
                table.setColumnWidthInChars(0, 8);
                table.setColumnWidthInChars(1, 15);
```

```
        table.setColumnWidthInChars(2, 5);
        table.setColumnWidthInChars(3, 5);
        table.setColumnWidthInChars(4, 20);
        table.setColumnWidthInChars(5, 23);
        table.setColumnWidthInChars(6, 15);

        // Set up menus
        setCommands(Selectable.VIEW_COMMANDS, listCmds);

        // Create TableData, plus Filter, Sort, and View pipes.
        setTableData(new TableData(userVals));
        String[] cols = new String[7];
        cols[0] = "Login";
        cols[1] = "Password";
        cols[2] = "User ID";
        cols[3] = "Group ID";
        cols[4] = "Comment";
        cols[5] = "Home Directory";
        cols[6] = "Shell";

        setQuerySpace(makeQuerySpace());
        setFilterPipe(new TableFilterPipe(getTableData(), cols));

        setSortPipe(new TableSortPipe(getFilterPipe(), cols));

        int[] wids = new int[7];
        wids[0] = 8;
        wids[1] = 15;
        wids[2] = 5;
        wids[3] = 5;
        wids[4] = 20;
        wids[5] = 23;
        wids[6] = 15;

        setViewPipe(new TableViewPropertiesPipe(getSortPipe(),
            cols, wids));

        getViewPipe().addObserver(this);
        getTableData().changed();
}


private QuerySpace makeQuerySpace() {
        QuerySpace qs = new QuerySpace();
        QueryRelation relation[] = {QueryRelation.MATCHES};
        Vector values;

        // Login
        values = new Vector();
        values.addElement("*");
        qs.add("Account", "Login", QueryType.STRING,
               relation, values);

        //Password
        values = new Vector();
        values.addElement("*");
        qs.add("Account", "Password", QueryType.STRING,
               relation, values);

        // User ID
        values = new Vector();
        values.addElement("*");
        qs.add("Account", "User ID", QueryType.STRING,
               relation, values);

        // Group ID
```

```
                values = new Vector();
                values.addElement("*");
                qs.add("Account", "Group ID", QueryType.STRING,
                       relation, values);

                // Comment
                values = new Vector();
                values.addElement("*");
                qs.add("Account", "Comment", QueryType.STRING,
                       relation, values);

                // Home Directory
                values = new Vector();
                values.addElement("*");
                qs.add("Account", "Home Directory", QueryType.STRING,
                       relation, values);

                // Shell
                values = new Vector();
                values.addElement("*");
                qs.add("Account", "Shell", QueryType.STRING,
                       relation, values);

                return qs;
        }
}
```

Anhang C

Manipulation von Managed Objects mittels der AVM-Klassen

Im folgenden soll ein kurzes Beispiel dafür gegeben werden, wie mittels der Klassen des JMAPI Admin View Module eine graphische Oberfläche zum Anlegen, Modifizieren und Löschen von Managed Objects erstellt werden kann. Die Managed Objects, die diesem Beispiel zugrundeliegen, werden in einer anderen Diplomarbeit [Rad98] zum Speichern bestimmter Managementinformation in der JMAPI-Datenbank des Lehrstuhls genutzt.

C.1 Ausgangssituation

Die Arbeit [Rad98] beschäftigt sich mit dem Policy-basierten Management nomadischer Systeme. Als nomadische Systeme (NoS) werden in diesem Zusammenhang in erster Linie Computer, wie etwa Notebooks oder Personal Digital Assistants betrachtet. Jedem derartigen NoS ist eine eindeutige MAC-Adresse zugeordnet. Ferner kann jedes NoS zu Managementzwecken einer Domäne zugewiesen werden. Die Zuordung von MAC-Adressen zu Domänennamen sollte in einer Datenbank gespeichert werden. Hierfür wurde die am Lehrstuhl vorhandene JMAPI-Datenbank verwendet. Über eine graphische Benutzeroberfläche sollte es möglich sein, die in der Datenbank gespeicherte Information zu verändern.

C.2 Anlegen der `NomadicSystemMO`-Managed-Object-Klasse

Zur Speicherung der Ethernet-Adresse und des Domänennamens eines Nomadic Systems in der JMAPI-Datenbank wurde zunächst eine Managed-Object-Klasse `NomadicSystemMO` angelegt, die die beiden Attribute `ethernetAddress` und `domainName` besitzt. Die zugehörige `.mo`-Datei sieht folgendermaßen aus:

```
// NomadicSystemMO.mo

package schiller.jmapi;

import sunw.admin.arm.common.*;
import sunw.admin.arm.manager.*;

import java.io.*;
import java.rmi.*;

public class NomadicSystemMOImpl extends ManagedObjectImpl
```

```
    implements NomadicSystemMO {

    // Persistent properties

    /**
     * Ethernet Address
     */
    String ethernetAddress;

    /**
     * Domain Name
     */
    String domainName;

    // constructor
    public NomadicSystemMOImpl() throws RemoteException {
    }

    public String getEthernetAddress(Session session)
      throws AuthorizationException, RemoteException {
        return (String)getPropertyByName(session, "ethernetAddress");
    }

    public void setEthernetAddress(Session session, String ethernetAddress)
      throws NotInUpdateException, AuthorizationException, RemoteException {
        setPropertyByName(session, "ethernetAddress", ethernetAddress);
    }

    public String getDomainName(Session session)
      throws AuthorizationException, RemoteException {
        return (String)getPropertyByName(session, "domainName");
    }

    public void setDomainName(Session session, String domainName)
      throws NotInUpdateException, AuthorizationException, RemoteException {
        setPropertyByName(session, "domainName", domainName);
    }
}
```

C.3 Überblick über den Programmcode

Als Basis für die graphische Oberfläche zur Manipulation der `NomadicSystemMO`-Managed-Objects dient ein Applet, das einen Content Manager enthält. Der Content Manager zeigt die in der Datenbank vorhandenen `NomadicSystemMO`-Instanzen in Tabellenform an. Die in der Tabelle dargestellte Information kann über das `View`-Pulldown-Menü gesteuert werden. Über das `Create`-Pulldown Menü können neue MO-Instanzen angelegt werden. Für eine neu anzulegende MO-Instanz erscheint ein Property Book, das die Festlegung von Initialwerten für die Attribute erlaubt. Das `Selected`-Pulldown-Menü erlaubt zum einen das Löschen der in der Tabelle selektierten MO-Instanz. Vor dem tatsächlichen Löschen aus der Datenbank erscheint ein Dialogfenster, in dem bestätigt werden muß, daß die betreffende MO-Instanz tatsächlich zu löschen ist. Ferner können über das `Selected`- Menü die Attributwerte einer in der Datenbank vorhandenen MO-Instanz mittels eines Property Books verändert werden.

Im folgenden ist der Programmcode für das Applet, den Content Manager, die Hilfsklassen für den Content Manager und das Property Book aufgeführt. Die entsprechenden Java-Klassen sind in den beiden Dateien `NomadicSystemContentManagerApplet.java` und `NomadicSystemPropertyBook.java` enthalten.

C.4 Datei NomadicSystemContentManagerApplet.java

```
/*
 * Autor: Christian Schiller
 * Datum: 15. August 1998
 */

package schiller.jmapi;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import sunw.admin.avm.base.*;
import sunw.admin.arm.rmi.*;
import sunw.admin.arm.common.*;
import sunw.admin.arm.manager.*;
import schiller.jmapi.*;

/**
 * Applet used for testing the NomadicSystemContentManager
 *
 */
public class NomadicSystemContentManagerApplet extends Applet {

    TopLevelContentManager mgr;
    ContentManager nomadicSystems;
    Image userImg;

    public void init() {
        setLayout(new BorderLayout());
        Context.setProperty("headerBackground", new Color(70,70,170));
        Context.setProperty("headerForeground", Color.white);
        try {
            /*
             * This is just a dummy image. We're not using it,
             * but an image is required for setting up the
             * SimpleContentManager
             */
            userImg = Context.getImage("images/user.gif", this);
        } catch (Exception e) {
            e.printStackTrace();
        }
        add("Center", mgr = new TopLevelContentManager());
        mgr.add("Center", nomadicSystems =
                new NomadicSystemContentManager("Nomadic Systems", userImg));
        nomadicSystems.addItemListener(mgr);
        mgr.select(nomadicSystems);
    }
}

/**
 * This command is used when "Delete .." is selected from the "Selected" menu
 */
class DeleteCmd implements Command, ActionListener {

    SecurityContext sc = new SecurityContext("nobody", null);
    RemoteEnumeration moEnum = null;
    Session session = null;
    QuestionDialog questionDialog;
    NomadicSystemMO mo = null;

    static final String className = "schiller.jmapi.NomadicSystemMO";

    public DeleteCmd() {
        try {
```

```
            session = MOFactory.newSession(sc);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public String getName() {
        return "Delete ...";
    }
    public String getLabel() {
        return "Delete ...";
    }

    /**
     * This method gets the first MO from the database which
     * matches the given ethernet address and domain name.
     * It then pops up a question dialog asking whether the object
     * should be deleted. When deletion is confirmed, the object is
     * removed from the database.
     */
    public void execute(Object executor) {
        SimpleContentManager scm = (SimpleContentManager) executor;
        Table t = scm.getTable();
        int i = t.getSelectedIndex();
        String domainName = (String) t.getItem(i, 1);
        String ethernetAddress = (String) t.getItem(i, 0);
        System.err.println("Domain Name: " + domainName);
        System.err.println("Ethernet Address: " + ethernetAddress);
        if (session == null) {
            sc = new SecurityContext("nobody", null);
            try {
                Session session = MOFactory.newSession(sc);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        QueryExp queryExp = Query.and(
                                  Query.eq(Query.attr("domainName"),
                                           Query.value(domainName)),
                                  Query.eq(Query.attr("ethernetAddress"),
                                           Query.value(ethernetAddress))
                                  );
        try {
            moEnum = MOFactory.enumerateMOClass(className, queryExp);
            if (moEnum == null || ! moEnum.hasMoreElements()) {
                System.err.println("No matching MO found in database.");
                System.err.println("Domain Name: " + domainName);
                System.err.println("Ethernet Address: " + ethernetAddress);
                return;
            }
            mo = (NomadicSystemMO) moEnum.nextElement();
            System.err.println("MO found in database");

            // pop up question dialog and ask for confirmation
            System.err.println("Popping up question dialog ...");
            Frame f = new Frame(); // dummy frame for constructor
            questionDialog =
                new QuestionDialog(f, "Really Delete this NomadicSystemMO?");
            questionDialog.addActionListener(this);
            questionDialog.setSize(400, 200);
            questionDialog.setVisible(true);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
    // action listener for question dialog
    public void actionPerformed(ActionEvent ae) {
        String cmd = ae.getActionCommand();
        if (cmd.equals("yes")) {
            System.err.println("Deleting MO ...");
            try {
                session.beginUpdate();
                mo.deleteObject(session);
                session.commitUpdate();
                System.err.println("Deletion successful.");
                questionDialog.dispose();
            } catch (Exception e) {
                System.err.println("Couldn't delete MO");
                e.printStackTrace();
            }
        } else {
            System.err.println("Nothing deleted.");
            questionDialog.dispose();
        }
    }
}

/**
 * This command is used when "Properties ..." is selected from the
 * "Selected" menu
 */
class PropCmd implements Command {

    SecurityContext sc = new SecurityContext("nobody", null);
    RemoteEnumeration moEnum = null;
    Session session = null;
    static final String className = "schiller.jmapi.NomadicSystemMO";

    public PropCmd() {
        try {
            session = MOFactory.newSession(sc);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public String getName() {
        return "Properties ...";
    }
    public String getLabel() {
        return "Properties ...";
    }
    /**
     * This method displays the property book corresponding to the
     * first MO found in the database matching the selected ethernet
     * address and domain name.
     */
    public void execute(Object executor) {
        SimpleContentManager scm = (SimpleContentManager) executor;
        Table t = scm.getTable();
        int i = t.getSelectedIndex();
        String domainName = (String) t.getItem(i, 1);
        String ethernetAddress = (String) t.getItem(i, 0);
        System.err.println("Domain Name: " + domainName);
        System.err.println("Ethernet Address: " + ethernetAddress);
        if (session == null) {
            sc = new SecurityContext("nobody", null);
            try {
                Session session = MOFactory.newSession(sc);
            } catch (Exception e) {
                e.printStackTrace();
```

```
            }
        }
        QueryExp queryExp = Query.and(
                                        Query.eq(Query.attr("domainName"),
                                                 Query.value(domainName)),
                                        Query.eq(Query.attr("ethernetAddress"),
                                                 Query.value(ethernetAddress))
                                        );
        try {
            moEnum = MOFactory.enumerateMOClass(className, queryExp);
            if (moEnum == null || ! moEnum.hasMoreElements()) {
                System.err.println("No matching MO found in database.");
                System.err.println("Domain Name: " + domainName);
                System.err.println("Ethernet Address: " + ethernetAddress);
                return;
            }
            NomadicSystemMO mo = (NomadicSystemMO) moEnum.nextElement();
            PropertyBook pb = new NomadicSystemPropertyBook(mo);
            pb.setCreateFlag(true);
            pb.display("Existing Nomadic System", (Container) executor);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}


/**
 * Used when "Create ..." is selected from the "Create" menu.
 */
class CreateCmd implements Command {
    public String getName() {
        return "Create ...";
    }
    public String getLabel() {
        return "Create ...";
    }
    public void execute(Object executor) {
        PropertyBook pb = new NomadicSystemPropertyBook();
        pb.setCreateFlag(true);
        pb.display("New Nomadic System", (Container) executor);
    }
}


/**
 * The content manager user for displaying all nomadic systems
 * in the database.
 */
class NomadicSystemContentManager extends SimpleContentManager
    implements JMAPIObserver {

    static private Command[] viewCmds = new Command[3];
    static private Command[] selCmds = new Command[2];
    static private Command[] createCmds = new Command[1];

    static final String className = "schiller.jmapi.NomadicSystemMO";
    static final String moServerHost =
        "sunhegering2.nm.informatik.uni-muenchen.de";
    static final int moServerPort = 7401;

    private Vector tableRows = new Vector();
    private SecurityContext sc = null;
    private Session session = null;
    private ObserverProxy observerProxy = null;
```

```
public NomadicSystemContentManager(String title, Image userImg) {
    super(title, userImg);

    Table table = getTable();
    table.setNumColumns(2);
    table.setColumnWidthInChars(0, 14);

    table.setColumnWidthInChars(1, 20);

    getTableRows();
    setTableData(new TableData(tableRows));
    System.out.println("data: " + getTableData());

    // the commands for the view menu
    viewCmds[0] = new FilterCommand();
    viewCmds[1] = new SortCommand();
    viewCmds[2] = new DisplayCommand();
    // the commands for the selected menu
    selCmds[0] = new DeleteCmd();
    selCmds[1] = new PropCmd();
    // the command for the create menu
    createCmds[0] = new CreateCmd();

    setCommands(Selectable.VIEW_COMMANDS, viewCmds);
    setCommands(Selectable.CREATE_COMMANDS, createCmds);

    String cols[] = new String[2];
    cols[0] = "Ethernet Address";
    cols[1] = "Domain Name";
    int widths[] = new int[2];
    widths[0] = 12;
    widths[1] = 18;

    /*
     * The table data are first piped through a filter pipe,
     * then through a sort pipe and finally through a view pipe.
     */
    setFilterPipe(new TableFilterPipe(getTableData(), cols));
    setQuerySpace(makeQuerySpace());
    setSortPipe(new TableSortPipe(getFilterPipe(), cols));
    setViewPipe(new TableViewPropertiesPipe(getSortPipe(), cols, widths));
    getViewPipe().addObserver(this);

    getTableData().changed();

    /*
     * The following code should set up an observer proxy which
     * notifies the content manager of any changes made to the
     * NomadicSystemMO class or any of its instances.
     * In case of a change, a notification should be sent to the
     * observer proxy which in turn should call the update() method
     * of the content manager (see below).
     * For some reason it doesn't seem to work as it should, though.
     */
    try {
        observerProxy = new ObserverProxy(sc, this);
    } catch (Exception e) {
        observerProxy.addClassObserver(className);
    }
}

/*
 * Access the database and get the ethernet addresses and
 * domain names. Insert the data into a vector, which is then used to
 * construct the table data.
 */
```

```
private void getTableRows() {

    RemoteEnumeration moEnum = null;
    NomadicSystemMO mo = null;
    Vector moVector = null;

    try {
        if (sc == null) {
            System.err.println("Initializing MOFactory ...");
            MOFactory.initialize(moServerHost, moServerPort);
            System.err.println("o.k.");
            sc = new SecurityContext("nobody", null);
            session = MOFactory.newSession(sc);
            System.err.println("New session created.");
        }
        // enumerate MOs
        moEnum = MOFactory.enumerateMOClass(className);
        if (moEnum == null) {
            System.err.println("No instances found in database.");
            return;
        }
        System.err.println("Adding MOs to table data ...");
        while (moEnum.hasMoreElements()) {
            mo = (NomadicSystemMO) moEnum.nextElement();
            moVector = new Vector();
            moVector.addElement(mo.getEthernetAddress(session));
            moVector.addElement(mo.getDomainName(session));
            tableRows.addElement(moVector);
            System.err.println("MO added");
        }
        System.err.println("Done adding MOs to table data.");
        return;
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/*
 * Construct the query space for the filter pipe.
 */
private QuerySpace makeQuerySpace() {
    QuerySpace qs = new QuerySpace();
    QueryRelation relation[] = {QueryRelation.MATCHES};
    Vector values;

    // Ethernet Address
    values = new Vector();
    values.addElement("*");
    qs.add("Nomadic System", "Ethernet Address", QueryType.STRING,
           relation, values);

    // Domain Name
    values = new Vector();
    values.addElement("*");
    qs.add("Nomadic System", "Domain Name", QueryType.STRING, relation,
           values);

    return qs;
}


/*
 * Get the commands which are available when an MO is selected and
 * the "Selected" menu is pulled down.
 */
public Command[] getCommands(String commandType) {
```

```
        if (commandType.equals(Selectable.SELECTED_COMMANDS)) {
            Object[] a = getSelectedObjects();
            return (a.length > 0) ? selCmds : null;
        }
        return super.getCommands(commandType);
    }

    /*
     * This is needed for the JMAPIObserver interface. It should be
     * called by the observer proxy, but for some reason it's never called.
     */
    public void update(Observation o) {
        System.err.println("update() method invoked");
        getTableRows();
        getTableData().changed();
    }
}
```

C.5 Datei `NomadicSystemPropertyBook.java`

```
/*
 * Autor: Christian Schiller
 * Datum: 15. August 1998
 */

package schiller.jmapi;

import java.awt.*;
import sunw.admin.avm.base.*;
import sunw.admin.arm.manager.*;
import sunw.admin.arm.common.*;
import sunw.admin.arm.rmi.*;
import schiller.jmapi.*;

/**
 * The property book for NomadicSystemMO's
 */
public class NomadicSystemPropertyBook extends PropertyBook {

    static final String moServerHost =
        "sunhegering2.nm.informatik.uni-muenchen.de";
    static final int moServerPort = 7401;
    static final String className = "schiller.jmapi.NomadicSystemMO";

    NomadicSystemSection nss;

    SecurityContext sc = null;
    Session session = null;

    public NomadicSystemPropertyBook() {
        addSection("Attributes", nss =
                   new NomadicSystemSection(this, getLabelFont()));
        startSession();
        nss.setSession(session);
        try{
            session.beginUpdate();
            nss.setManagedObject(
                (NomadicSystemMO) MOFactory.newObj(session, className));
            session.commitUpdate();
        } catch (Exception e) {
            e.printStackTrace();
        }
        clearValues();
    }
```

```
    public NomadicSystemPropertyBook(NomadicSystemMO mo) {
        addSection("Attributes", nss =
                   new NomadicSystemSection(this, getLabelFont()));
        nss.setManagedObject(mo);
        startSession();
        nss.setSession(session);
        setValuesFromDB();
    }

    public void clearValues() {
        nss.setEthernetAddress("");
        nss.setDomainName("");
    }

    public void setValuesFromDB() {
        try{
            nss.setEthernetAddress(nss.mo.getEthernetAddress(session));
            nss.setDomainName(nss.mo.getDomainName(session));
            System.err.println("values set.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void startSession() {
        if (session == null) {
            try {
                System.err.println("Initializing MOFactory ...");
                MOFactory.initialize(moServerHost, moServerPort);
                System.err.println("o.k.");
                sc = new SecurityContext("nobody", null);
                session = MOFactory.newSession(sc);
                System.err.println("New session started.");

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

class NomadicSystemSection extends PropertySection {

    TextField ethernetAddress;
    TextField domainName;

    // reference to the MO this property book applies to
    NomadicSystemMO mo = null;
    Session session = null;

    public NomadicSystemSection(PropertyBook book, Font labelFont) {
        Label label;
        setLayout(new FieldLayout());

        // Ethernet Address
        label = new Label("Ethernet Address:");
        label.setFont(labelFont);
        add("Label", label);
        add("Field", ethernetAddress = new TextField("", 20));

        // Domain Name
        label = new Label("Domain Name:");
        label.setFont(labelFont);
        add("Label", label);
        add("Field", domainName = new TextField("", 20));
    }
```

```
    public String getEthernetAddress() {
        return ethernetAddress.getText();
    }

    public String getDomainName() {
        return domainName.getText();
    }

    public void setEthernetAddress(String ea) {
        ethernetAddress.setText(ea);
    }

    public void setDomainName(String dn) {
        domainName.setText(dn);
    }

    public void apply() {
        if (mo == null) {
            return;
        }

        String ea = getEthernetAddress();
        String dn = getDomainName();
        if (ea.equals("") || dn.equals("") || ea == null || dn == null) {
            System.err.println(
                "Apply failed: Empty string or null reference.");
            return;
        }
        try {
            System.err.println("Starting update ...");
            session.beginUpdate();
            mo.setDomainName(session, dn);
            mo.setEthernetAddress(session, ea);
            session.commitUpdate();
            System.err.println("Update successful.");
        } catch (Exception e) {
            System.err.println("Error while updating database.");
            e.printStackTrace();
        }
    }

    public void reset() {
        setEthernetAddress("");
        setDomainName("");
    }

    void setManagedObject(NomadicSystemMO theMO) {
        mo = theMO;
    }

    void setSession(Session theSession) {
        session = theSession;
    }
}
```

Anhang D

Beispiel für ein JMAPI Properties File

```
# JMAPI Management Server Properties File
#
# Copyright 1997 by Sun Microsystems, Inc.
# All rights reserved
#
#
# NOTE: All paths should use forward (/) slashes.
#       If a backward slash is desired, use (\\).
#
# NOTE: The java.util.Properties class does NOT strip off
#       trailing spaces.
#
# NOTE: Some properties also specify the value of environment
#       variables.  Properties of the form...
#
#                 jmapi.setenv.<var>=<value>
#
#       will instruct the JMAPI startup process to first set
#       an environment variable named <var> to the indicated
#       <value>.
#

jmapi.setenv.CLASSPATH=/proj/java/JMAPI/jdbcDriver/FastForward/jdk113:\
        /users/stud/schiller/classes:/opt/JMAPI/classes:/opt/JMAPI/examples

#
# JMAPI services to initialize
#

jmapi.services=sunw.admin.arm.agents.AgentObjFactoryImpl\
         sunw.admin.arm.rmi.ManagedObjFactoryImpl
#sunw.admin.arm.agents.NetClassServerImpl


#
# JMAPI Registry port numbers
#

jmapi.port.sunw.admin.arm.agents.AgentObjFactoryImpl=7400
jmapi.port.sunw.admin.arm.rmi.ManagedObjFactoryImpl=7401
jmapi.port.sunw.admin.arm.agents.NetClassServerImpl=7402
jmapi.port.sunw.admin.arm.event.EventDispatcher=7403


#
```

```
# JMAPI Administration files.  Please set to a temporary file  JMAPI
# can use.
#

jmapi.services.status.file=/tmp/jmapi.services.status


#
# JMAPI factory output redirection.  These are optional.
#

jmapi.server.stdout=/tmp/jmapi.server.out
jmapi.server.stderr=/tmp/jmapi.server.err
jmapi.agent.stdout=/tmp/jmapi.agent.out
jmapi.agent.stderr=/tmp/jmapi.agent.err
jmapi.classserver.stdout=/tmp/jmapi.classserver.out
jmapi.classserver.stderr=/tmp/jmapi.classserver.err


#
# JMAPI server hostname
#

jmapi.server.hostname=sunhegering2.nm.informatik.uni-muenchen.de

#
# Where to find downloadable classes on the JMAPI Management server
#

jmapi.server.classpath=/tmp

#
# Where to find downloadable native libraries on the JMAPI Management
# server
#

# jmapi.server.lib.path=/tmp/lib
# jmapi.server.shared.lib.path=/tmp/dlib

#
# Where to download native libraries on the JMAPI Agent host
#

# jmapi.agent.lib.path=/usr/lib

#
# Java home.
#
# Please include the correct library in the CLASSPATH
# depending on whether you are using the full JDK, or
# just the JRE...
#
#       JDK: ${JAVA_HOME}/lib/classes.zip
#       JRE: ${JAVA_HOME}/lib/rt.jar
#

jmapi.setenv.JAVA_HOME=/usr/local/dist/DIR/jdk1.1
#jmapi.setenv.JAVA_HOME=/usr/java

#
# JMAPI Persistence properties. (Sybase)
#

jmapi.database.JDBCURL=jdbc:ff-sybase://sunhegering11:63324/jmapi
jmapi.database.JDBCDRIVER=connect.sybase.SybaseDriver
```

```
jmapi.database.DBSERVERTYPE=JDBC
jmapi.database.DBSERVER=STP_SERVER_11
jmapi.database.DBNAME=jmapi
jmapi.database.DBOWNER=sa
jmapi.database.DBPASSWD=welcome

jmapi.database.DBHOST=sunhegering11

jmapi.database.DBDISK=jmapi
jmapi.database.DBLOG=jmapi
jmapi.database.DBSYSADMIN=sa
jmapi.database.DBSYSADMINPASSWD=welcome
jmapi.database.DBSIZE=15
```

Anhang E

Email-Korrespondenz mit Sun Microsystems

E.1 Fragen zur JMAPI-Architektur

```
Date: Tue, 2 Jun 1998 10:55:40 -0600 (MDT)
From: Bill Birnbaum <Bill.Birnbaum@Eng.Sun.COM>
To: Christian Schiller <schiller@informatik.tu-muenchen.de>
Cc: Bill.Birnbaum@Eng.Sun.COM
Subject: Re: JMAPI Architecture Questions

> Hello,
>
> There are a couple of questions that I would like to hear your comments
> on:
>
> 1. CIM
> ------
>    - I understand that the base managed object classes will eventually
>      be an implementation of the DMTF CIM model. Will there be some
>      support for the MOF syntax, e.g. a MOF2mo compiler?
>

We would like to do something like this, but we will do not even have a
schedule for this type of effort. Probably a next major version work item.

> 2. RMI/CORBA
> ------------
>    - Will you stick to RMI as the communication protocol or are you
>      planning on integrating some sort of CORBA support?
>

We plan on sticking to RMI for comminication between the browser/application
and MO server. Currently, you can use CORBA to communicate with agents. JMAPI
has a preference toward our RMI agents though its integration with our
concurrency and security model, but there is not anything stopping people from
using other agent technology such as SNMP and CORBA.

>
> 3. Scaling aspects
> ------------------
>    - From what I hear, the next JMAPI release is going to include
>      the possibility of several MO servers talking to each other.
>      Will it be possible to store the same data on several MO servers
>      for some sort of load balancing and/or fault tolerance?

We are still looking at this. Right now we are concentrating on getting the
```

```
basic APIs through the java standardization process. So this is most likely
the part of the next major version project.

>    - What is the communication between the MO servers going to look
>      like?

Do not know yet.

>
> 4. Fault Tolerance
> ------------------
>    - What happens when parts of the system fail? (e.g. an MO server
>      goes down, an agent crashes, etc.)

I believe  we can currently take advantage of the distributed capabilities of
some of the commercially available databases to have a backup MO server. If
the MO server goes down you would lose all of the currently active work, but
you switch over to a replicated database and MO Server and be able to continue
with minimal disruption.

It is up to the designer of the agent to determine the failure modes and
recovery from those modes. There are lots of things an agent in conjunction
with a Managed Object could do.

>
> 5. Naming Service
> -----------------
>    - I noticed that parts of the current release of JMAPI might
>      benefit from the use of a naming service. Are there any plans
>      along these lines?

No. We have been assuming that the network may not have a name service
installed or working. We do not want to be dependent on it.


Hope this helps,

Bill
```

E.2 Verwendbare Datenbanken und Treiber

```
Date: Thu, 2 Apr 1998 07:10:46 -0700 (MST)
From: Bill Birnbaum <Bill.Birnbaum@East.Sun.COM>
To: Julian Kolodko <julian.kolodko@star.com.au>
Cc: bill.birnbaum@East.Sun.COM,
    jmapi-interest@East.Sun.COM
Subject: Tested Database/JDBC combinations

Attached is the list of databases and JDBC driver we have had working as
outlined in the README:


1.1 DATABASE OPTIONS

As you can imagine, testing the combination of databases, operating systems,
and drivers that JMAPI will support is a fairly large task. This release
supports Sybase, Oracle, and JDBC. We are in the process of writing the
installation and setup instructions on how to use these various options.

We will also provide a list of specific versions of databases and drivers
that we will support. We have found many differences with the JDBC driver
implementations and only a few of them have the necessary compliance level
to support JMAPI.
```

Here is a list of databases that JMAPI supports and the interfaces used to support each:

Database	Version	OS	Driver
Sybase	10.x	Solaris & NT 3.5	Native JMAPI Driver
Sybase	11	NT 4.0	Native JMAPI Driver
Sybase	11	Solaris & NT	JDBC
Oracle	7	Solaris & NT	Native JMAPI Driver
Oracle	8	Solaris & NT	JDBC
MS SQL	6	NT	JDBC

1.2 JDBC SUPPORT IN THIS RELEASE

This JMAPI release includes support that allows you to use a JDBC driver to connect to a JDBC-compliant database. While any JDBC 1.2 compliant Type 4 driver should work, we have tested only with Connect Software's FastForward JDBC driver. For more information on this driver see the Connect home page at:

http://www.connectsw.com

```
> Hello all
>
>       Try as I might, I can not find a JDBC capable database and driver
> that will actually work with JMAPI.  I've tried Progress(v7.3D and v8.2A),
> Postgress (v6.0 - I have not tried v6.3 and the new JDBC driver as yet), and
> MSQL.  I've tried drivers from Openlink, InterSolv and Sun.  The common
> problem with all these combinations is that none of them support Level 3
> Transaction Isolation (thats what jrb.util.TestJDBCDriver reports anyhow).
>
>       When I run setupdb.bat using a Progress database I get errors like
> "Unable to understand after create" (from Intersolv Drivers)  and "Could not
> get tableschema info - Syntax error or access error: Base jmapi does not
> exist" (from Openlink drivers).
>
>       Why the correlation between Level 3 Isolation and problems with
> tables?  Is this level of consistency control really needed?
>
>       Has anyone ever managed to get a JDBC database to talk to JMAPI?
> Can anyone suggest an alternate driver/database?  Could this be a problem
> with Progress?
>
>       Thanks for your help
>
>       Julian Kolodko
>       Star Systems
>
```

E.3 Zukunft von JMAPI

```
From Bill.Birnbaum@Eng.Sun.COM Thu Aug 13 22:50:37 1998
Date: Wed, 10 Jun 1998 13:25:12 -0600 (MDT)
From: Bill Birnbaum <Bill.Birnbaum@Eng.Sun.COM>
To: Hong Zheng <euszeng@exu.ericsson.se>
Cc: jmapi-interest@East.Sun.COM
Subject: Re: External JMAPI comments

> Hi,
>
> We are considering using JMAPI for our managment
> system. Have you have the JMAPI 1.0 Spec yet?
> Seems that JMAPI will be part of JDK 1.2. If this
> is true, when will 1.2 be official released?
>

The JMAPI specification has finished partner review. We are doing a quick
editing pass and we should starting Java Licensee review shortly. The
Licensees have 60 days to provide comments. After that, assuming all goes well
it will be out for public review.

There is not any dependency on JDK 1.2 and I cannot say when 1.2 will be
released.

Bill
```

Abkürzungsverzeichnis

API	Application Programming Interface
AWT	Abstract Window Toolkit
CGI	Common Gateway Interface
CIM	Common Information Model
CMF	Common Management Framework
CORBA	Common Object Request Broker Architecture
DBMS	Database Management System
DMI	Desktop Management Interface
DMTF	Desktop Management Task Force
GUI	Graphical User Interface
HMMP	Hyper Media Management Protocol
HTTP	Hyper Text Transfer Protocol
IETF	Internet Engineering Task Force
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IP	Internet Protocol
JDBC	Java Database Connectivity
JDK	Java Development Kit
JDMK	Java Dynamic Management Kit
JMAPI	Java Management Application Programming Interface
JNI	Java Native Interface
MAC	Medium Access Control
MIB	Management Information Base
MO	Managed Object
MOF	Managed Object Format
NIS	Network Information Service
OMG	Object Management Group
ORB	Object Request Broker
PDU	Protocol Data Unit
RMI	Remote Method Invocation
SMI	Structure of Management Information
SNMP	Simple Network Management Protocol
TQL	Topological Query Language
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VM	Virtual Machine
WBEM	Web Based Enterprise Management

Literaturverzeichnis

[BEA98] BEA Systems, Inc., et al. *Notification Service, Joint Revised Submission*, April 1998. OMG TC Document Telecom/98-04-01.

[CL96] Patrick Chan und Rosanna Lee. *The Java Class Libraries.* Addison Wesley, 1996.

[Des97] Desktop Management Task Force, Inc. *Common Information Model (CIM) Core Common Schema, Version 0.1*, Dezember 1997.

[Des98] Desktop Management Task Force, Inc. *Common Information Model (CIM) Specification, Version 2.0*, März 1998.

[Fla97a] David Flanagan. *Java Examples in a Nutshell.* O'Reilly, Sebastopol, 1997.

[Fla97b] David Flanagan. *Java in a Nutshell.* O'Reilly, Sebastopol, Second Edition, 1997.

[Gio98a] Mike Gionfriddo. *Network Management with Java Technology.* Audio Tape des Vortrags, http://java.sun.com/javaone/javaone98/sessions/T417/, März 1998. JavaOne Developer Conference.

[Gio98b] Mike Gionfriddo. *Network Management with Java Technology.* Foliensammlung, http://java.sun.com/javaone/javaone98/sessions/T417/, März 1998. JavaOne Developer Conference.

[Gof98] Max Goff. *Java and Management.* Sun Microsystems, Inc., Juli 1998. Foliensammlung, Sun Software Technology Day 1998, München.

[HA93] Heinz-Gerd Hegering und Sebastian Abeck. *Integriertes Netz- und Systemmanagement.* Addison-Wesley, Bonn, 1993.

[Hew97] Hewlett-Packard Company. *Response to Topology Service RFP*, Oktober 1997. OMG TC Document Telecom/97-10-02.

[JMA97] SunSoft, Inc. *JMAPI Developer's Release 0.5*, Oktober 1997. http://java.sun.com/products/JavaManagement/download.html, Solaris Version: jmapi05-46.tar.Z.

[Lor98] Christian Lorentz. *Architektur und Technologien für Web-Based Management.* Foliensammlung, März 1998. CeBIT Vortrag, Cisco Systems.

[Mai] *JMAPI Mailing List.* email: jmapi-interest@east.sun.com, subscribe: jmapi-interest-subscribe@east.sun.com, unsubscribe: jmapi-interest-unsubscribe@east.sun.com.

[Mül98] Tobias Müller. *CORBA-basiertes Management von UNIX-Workstations mit Hilfe von ODP-Konzepten.* Diplomarbeit, Technische Universität München, Februar 1998.

[OH97] Robert Orfali und Dan Harkey. *Client/Server Programming with Java and CORBA.* John Wiley & Sons, Inc., New York, 1997.

[OMG97] OMG. *CORBAservices: Common Object Services Specification*, November 1997. http://www.omg.org/corba/csindx.htm.

[Rad98] Igor Radisic. *Konzeption eines Policy-basierten Konfigurationsmanagements für nomadische Systeme in Intranets.* Diplomarbeit, Technische Universität München, Februar 1998.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, und William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, 1991.

[Sie96] Jon Siegel, editor. *CORBA Fundamentals and Programming.* John Wiley & Sons, New York, 1996.

[Sun97a] Sun Microsystems, Inc. *Java Management API Help Developer's Guide*, Mai 1997.

[Sun97b] Sun Microsystems, Inc. *Java Management API Installation Guide for Solaris*, September 1997.

[Sun97c] Sun Microsystems, Inc. *Java Management API Programmer's Guide*, September 1997.

[Sun97d] Sun Microsystems, Inc. *Java Management API User Interface Style Guide*, Mai 1997.

[Sun97e] Sun Microsystems, Inc. *Java Management API User Interface Visual Design Style Guide*, Mai 1997.

[Sun97f] Sun Microsystems, Inc. *JavaBeans 1.01 Specification*, Juli 1997. http://java.sun.com/beans/docs/spec.html.

[Sun97g] SunSoft, Inc. *JMAPI Javadocs*, Oktober 1997. Nach der JMAPI-Installation im lokalen Verzeichnis /opt/JMAPI/doc/java_docs.

[Sun98a] Sun Microsystems, Inc. *Java Dynamic Management Kit, A White Paper*, 1998. http://www.sun.com/software/java-dynamic/wp-jdmk/.

[Sun98b] Sun Microsystems, Inc. *Java Foundation Classes (JFC)*, 1998. http://java.sun.com/products/jfc/index.html.

[Sun98c] Sun Microsystems, Inc. *JavaHelp, A New Standard for Application Help and Online Documentation*, 1998. http://java.sun.com/products/javahelp/index.html.

[Sun98d] Sun Microsystems, Inc. *RMI - Remote Method Invocation*, 1998. http://java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html.