INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Entwicklung und Implementierung einer Benutzerschnittstelle für Policy-Managementagenten

Bearbeiter: Dimitrios Golias
Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering
Betreuer: Stephen Heilbronner
Boris Gruschke

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Im Rahmen der Diplomarbeit von Igor Radisic [Radi 98] wurde der Versuch unternommen, ein Konzept für policybasiertes Management von Nomadischen Systemen (NoCS) zu erarbeiten und zu implementieren. Unter dem Begriff "Nomadische Systeme" werden dabei im weitesten Sinne mobile Endgeräte (Computer) angesehen, wie Laptops, Notepads und Personal Digital Assistants (PDAs). Ihre Mobilität stellt für das Netz- und Systemmanagement eine schwierige Aufgabe dar, da verschiedene Aspekte berücksichtigt werden müssen, bevor diese am Gesamtsystem (-netz) oder Teilbereichen dessen teilnehmen können. Einerseits soll die Mobilität solcher Objekte ermöglicht und unterstützt werden. Hierfür muß von Seiten des Managements dem NoCS gewisse Grundkonfigurationsdaten, wie z.B. IP-Adresse und DNS-Namen, zur Verfügung gestellt werden können. Andererseits muß der Ressourcenzugriff der NoCS in geeigneter Weise gesteuert, beeinflußt und überwacht werden. Zu denken sei hier etwa an Peripheriegeräte, wie Scanner, Plotter und Drucker. Oder an Server-Dienste, die den Teilnehmern gewisse Funktionalität zur Verfügung stellen, wie WWW- und Mailserver. Schließlich sollen, speziell im kommerziellen Bereich (z.B. in Hotels), auch solche Dienste den NoCS zur Verfügung stehen, deren Nutzung kostenpflichtig ist. Dafür sollte es, von Seite des Systemmangements einen kontrollierten Zugriff geben, um die Möglichkeit einer Abrechnung (Accounting) nutzen zu können.
Um das Management von NoCS hinsichtlich der hier nur kurz skizzierten Aspekten zu verwirklichen, wurde die Verwendung von Policies herangezogen. Mit der Formulierung von Policies soll die Konfigurierung von NoCS, sowie desen Zugriff auf Ressourcen gesteuert und die gewünschten Ziele erreicht werden.

1.1 Policybasiertes Management

Im policybasierten Management will man, wie der Name schon sagt, mit der Formulierung von Policies ein Management betreiben. Policies sind Regeln, die das Verhalten von Managed Objects steuern sollen (im unseren Fall sind dies die NoCS). Ein Beispiel für eine Policy wäre z.B.

> *Nomadische Systeme von Besuchern dürfen am Institut nur 50 Seiten pro Monat ausdrucken.*

Eine so formulierte Policy, auch ***High level Policy*** genannt, stellt nur den ersten Schritt dar. In dieser abstrakten Form ist sie zwar für Menschen verständlich, kann aber nicht maschinell interpretiert und durchgesetzt werden. Es muß eine Umsetzung dieser in einer anderen Form erreicht werden, so daß sie technisch realisierbar ist. Man spricht dann von einer ***Low level Policy***.
Der Entwicklungsprozeß von einer High level zu einer Low level Policy wird, je nach Abstraktionsebene, in

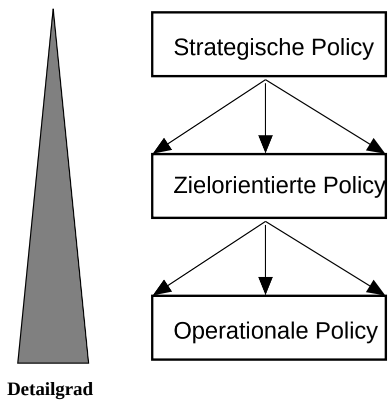

Abbildung 1: Die Policy-Hierarchie

mehreren Stufen unterteilt. Nach [Radi 98] und [Heil 98a] werden drei Hierarchiestufen unterschieden, die in Abbildung 1 dargestellt sind. Daraus ergeben sich folgende Policy-Typen:

- strategische,
- zielorientierte und
- operationale Policies.

Eine **strategische Policy** enthält nur eine informelle Beschreibung über das verfolgte Ziel. Sie wird umgangssprachlich verfaßt.

In einer **zielorientierten Policy** werden die Subjekte, Zielobjekte, Aktionen und Bedingungen, unter denen sie ausgeführt werden soll, konkretisiert. Sie sind ebenfalls nicht an eine konkrete Syntax gebunden und werden auch nicht interpretiert.
Erst in einer **operationalen Policy** wird gefordert, daß diese in einer festgelegten Syntax, die ***Policy Description Language (PDL)*** (siehe [Radi 98]), spezifiziert wird. Hierdurch wird erreicht, daß die in PDL verfaßten operationalen Policies durch eine Managementanwendung automatisiert interpretiert und dadurch auch tatsächlich durchgesetzt werden können.

1.2 Managementarchitektur

Hauptmerkmal der Managementarchitektur, die in [Radi 98] für ein policybasiertes Management entwickelt wurde, ist ihr modularer Aufbau. Wie in Abbildung 2 zu sehen ist, besteht diese aus drei Hauptbestandteilen: der **Managementanwendung**, den **Managementagenten** und den **Managementobjekten** (Managed Objects, MOs).
Die Managementanwendung, die ein policybasiertes Management ermöglicht, agiert auf den Managementa-

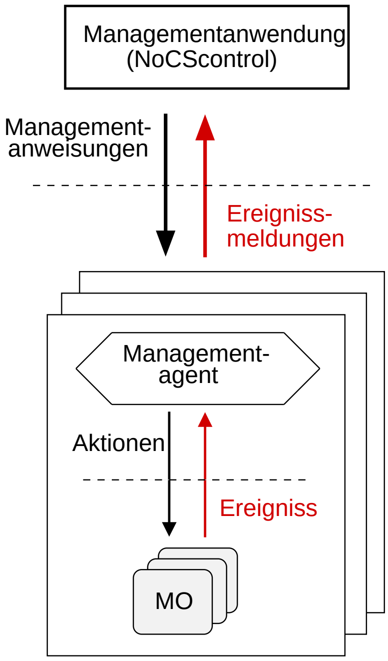

Abbildung 2: Die Managementarchitektur

genten durch Verwendung einer einheitlichen Kommunikationsschnittstelle. Die Managementagenten wiederum agieren auf den MOs über diejenigen Schnittstellen, die diese unterstützen. Die gewünschte Steuerung der MOs wird dadurch erreicht, indem auf Ereignisse der MOs reagiert wird. Tritt ein Ereignis ein, so meldet der entsprechende Managementagent dies an die Managementanwendung weiter. Die Managementanwendung entscheidet nach der Interpretation der verschiedenen Policies über die Reaktion auf dieses Ereignis und veranlaßt den Managementagenten auf das MO einzuwirken (In der Regel wird dem MO eine Aktion, z.B. Benutzen des Druckers, erlaubt oder verboten).
Durch den modularen Aufbau der Gesamtarchitektur ist die Anzahl und Art insbesondere der Managementagenten fast beliebig skalierbar. Dies ist ein großer Vorteil, denn es können, je nach Bedarf und Aufgabe, entsprechende Agenten der Managementanwendung unterstellt und entspechende Policies kreiert werden, ohne großen Aufwand betreiben zu müssen. Am Institut wurden Prototypen solcher Managementagenten entwickelt, die über die Managementanwendung (NoCScontrol) gesteuert werden können. Konkret wurde im Rahmen eines Fortgeschrittenenpraktikums [Demm 98] ein DHCP-Agent und einer Diplomarbeit [Allg 98] ein Switch-Agent implementiert.

1.3 Die Managementanwendung NoCScontrol

Die Managementanwendung NoCScontrol muß verschiedene Aufgaben leisten können, um ein policybasiertes Management von Nomadischen Systemen bewerkstelligen zu können. Solche Teilaufgaben sind unter anderem:

- Kreieren, persistentes Anlegen und Löschen von Policies,
- Lösen von Konflikten zwischen mehreren Policies und
- Durchsetzen von aktivierten operationalen Policies, indem das Zielobjekt beobachtet wird und bei Eintreten von Ereignissen die in der Policy spezifizierten Managementaktionen ausgelöst werden.

Die Teilaufgaben wurden in der konkreten Implementierung (s. [Radi 98]) durch verschiedene Teilkomponenten realisiert, die alle zusammen die Managementanwendung ausmachen. Jede dieser Komponenten ist folgendermaßen aufgebaut:

- Sie ist als **CORBA-Objekt** realisiert und bietet eine bestimmte Funktionalität an, die durch eine wohldefinierte Schnittstelle genutzt werden kann. Sie kann sowohl als Client agieren und dadurch Funktionalitäten anderer Komponenten der Managementanwendung nutzen oder als Server gewisse Funktionalitäten anderen Komponenten anbieten.
- Sie wird in Form eines **MASA-Agenten** implementiert (s. [Kemp 98, GHR 99]).
- Lebensraum eines Agenten ist ein **Agentensystem**. Dieses übernimmt die Aufgaben des Startens, Beendens und der Migration des Agenten auf ein anderes Agentensystem innerhalb der dem Agenten zugänglichen Region.
- Über eine **Agenten-GUI** (Applet), die über das Agentensystem erreichbar ist, können die Operationen ihrer IDL-Schnittstelle aufgerufen werden.

Insgesamt wurden drei Komponenten entworfen, die allesamt, wie erwähnt, als MASA-Agenten implementiert wurden. Im folgenden sind diese drei Agenten und ihre Aufgaben beschrieben (s. auch [Radi 98]):

- Der **Policy Factory Mobile Agent** (PFMA) bietet CORBA-Clients die Möglichkeit strategische, zielorientierte und operationale Policies anzulegen. Handelt es sich um eine operationale Policy, so kann diese durch den Agenten aktiviert und deaktiviert werden.
- Der **Persistence Service Stationary Agent** (PSSA) bietet die Möglichkeit CORBA-Objekte persistent anzulegen, die insbesondere Policies sein können. Seine Dienste werden vom PFMA in Anspruch genommen, um die kreierten Policies persistent abspeichern zu können.
- Der **Enforcement Object Factory Mobile Agent** (EOFMA) kümmert sich um die tatsächliche Durchsetzung der aktiven Policies. Dabei werden durch ihn die entprechenden Managementagenten überwacht. Je Managementagententyp ist ein eigener EOFMA vorgesehen. Seine Dienste werden vom PFMA jedesmal beansprucht, wenn eine operationale Policy aktiviert wird.

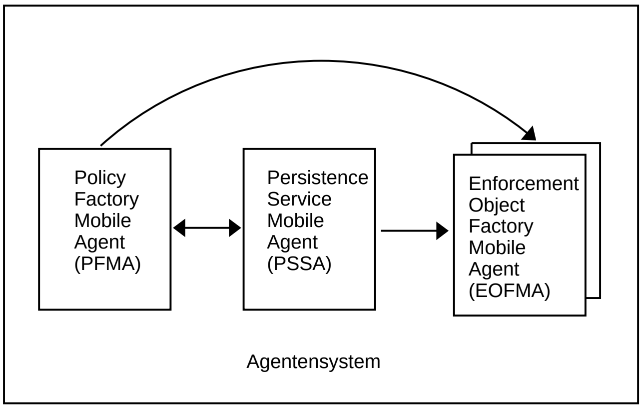

Abbildung 3: Gesamtüberblick über die Komponenten der Managementanwendung NoCScontrol (nach [Radi 98])

Als Agentensystem, welches die Laufzeitumgebung der drei Agenten darstellt, wurde ein hier am Institut entwickeltes System verwendet, das im Rahmen einer Diplomarbeit ([Kemp 98]) entworfen und in weiteren Arbeiten (s. [Gerb 99, GHR 99]) weiterentwickelt wurde. Das System nennt sich *Mobile Agent System Architecture (MASA)*. Über dieses System können verschiedene Dienste in Anspruch genommen werden um Agenten zu verwalten. Zu seinen Aufgaben gehören, neben der Bereitstellung einer Laufzeitumgebung, das Starten und Beenden, das Anhalten und Wiederanfahren der Agenten und die Durchführung der Migration von Agenten zu anderen Agentensystemen, sofern es sich um mobile Agenten handelt. Außerdem stellt es einen Webserver bereit, der es Java-fähigen Webbrowsern ermöglicht, die Applets der einzelnen Agenten zu laden. MASA ist in Java impementiert worden. Als Kommunikationsinfrastruktur für die Agenten wurde CORBA verwendet. Einen Überblick der MASA-Architektur gibt Abbildung 4.

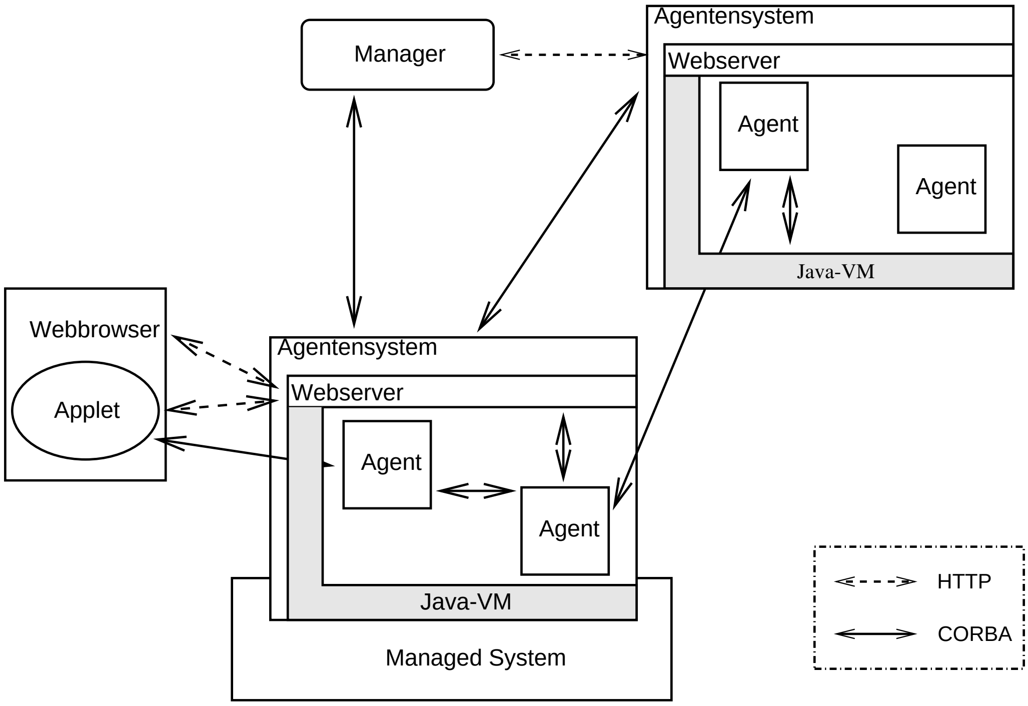

Abbildung 4: Die Architektur der MASA (nach [Kemp 98])

1.4 Die Aufgabenstellung

In diesem Fortgeschrittenenpraktikum sollten die in [Radi 98] entwickelten Policy-Managementagenten jeweils durch eine geeignete Bedienoberfläche erweitert werden. Diese sollten als Java-Applets ([JLS, Flan 96, Naug 96, Sun 98]) unter Verwendung von GUI-Elementen aus der Swing-Bibliothek von Sun Microsystems realisiert werden. Zu beachten waren hier die Anforderungen, die jeder Agent und sein Applet erfüllen muß, um in das Agentensystem MASA integriert werden zu können (vgl. [Kemp 98, Gerb 99, GHR 99]).

2 Die Policy-Managementagenten

2.1 Der PolicyFactoryMobileAgent

Auf dem PolicyFactoryMobileAgent (PFMA) werden sowohl die *Policy Factory* realisiert, als auch die von dieser verwalteten *Policy Objects* instanziiert. Alle Komponenten wurden mit CORBA-Schnittstellen versehen und bieten somit ihre Funktionalität über eine Schnittstelle CORBA-Clients an (z.B. fungiert das Applet des PFMA als CORBA-Client). Während der Entwicklung der Oberfläche für den PFMA stellte sich dadurch die Frage, wie man am besten diese Funktionalität in einer einzigen Oberfläche unterbringen kann. Grundsätzlich sollte der Benutzer des Agentenapplets die Möglichkeit haben, alle drei Policy-Typen (s. Kapitel 1.1) zu kreieren. Um den Überblick über die erstellten Policies zu behalten, sollte eine strukturierte Darstellung aller bereits erstellten Policies geboten werden. Schließlich sollten die Attribute (Eigenschaften) der Policies nachträglich veränderbar sein, insbesondere das Aktivieren und Deaktivieren von operationalen Policies.

2.1.1 Die IDL-Schnittstellen

Policy Factory Im folgenden ist die IDL-Spezifikation der Policy Factory angegeben, die die Grundlage für das weitere Vorgehen bildete:

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

Es wurden alle angebotenen Operationen ausgenutzt und implementiert. Ihre Bedeutung sei hier kurz erläutert (s. auch [Radi 98]):

- *StrategicPolicy* **createStratPol** *(in string PolDescr)* Eine strategische Policy wird erzeugt. Als Parameter wird die in Freitext verfaßte Policy-Beschreibung übergeben.
- *GoalorientedPolicy* **createGoalPol** *(in string pName, in PolicyModality pModality, in string pSubject, in string pSDomain, in string pTarget, in string pTDomain, in string pAction, in string pConstraint)* Eine zielorientierte Policy wird erzeugt. Übergeben werden eine Reihe von Parametern, die über eine Eingabemaske abgefragt wurden.
- *OperationalPolicy* **createOperPol** *(in string PolDescr)* Eine operationale Policy wird erzeugt. Die übergebende Policy-Beschreibung muß in PDL verfaßt worden sein.
- *boolean* **destroyPolicy** *(in string aPolicyID)* Die Policy mit der angegebenen ID wird gelöscht.
- *PolicyObject* **getPolicy** *(in string policyID)* Liefert die Policy mit der angegeben ID. Zu beachten ist hier, daß nur ein Objekt vom Typ *PolicyObject* zurückgeliefert wird. Diese Funktion dient vor allem dazu, um einzelne Policy-Attribute abzufragen.
- *PolicyObjectlist* **listAllPolicies** *()* Liefert eine Liste von *PolicyObjects* zurück. Ausgehend von den IDs kann dann mit **getPolicy** *(policyID)* das entsprechende *PolicyOject* geholt werden.

Die IDL-Schnittstellen der Policy Objects Da die Policies in drei Typen unterteilt sind skap1.1, gestaltete sich auch deren Schnittstellenspezifikation entsprechend. Es wurde zwischen einer Reihe von Operationen unterschieden, die alle Typen unterstützen müssen, und solchen, die typbezogen einzustufen waren, und somit nur von dem entsprechenden Policy-Typ bereitgestellt zu werden brauchten.
Zunächst haben alle Policies eine Reihe von gemeinsamen Operationen, die durch folgendes IDL-Interface spezifiziert werden:

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

Je nach Typ haben die Policies zusätzliche, spezifische Operationen, die durch folgende IDL-Schnittstellen spezifiziert werden:

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

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

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

Zu beachten ist, daß die meisten angebotenen Operationen nur einen lesenden Zugriff auf die entsprechenden Policies erlauben (alle die mit get beginnen). Lediglich die drei Operationen, die allen Policy-Typen gemeinsam sind, erlauben das Setzen von Policy-Attributen:

- void **setParentID** (in string PolID)
- boolean **addChildID** (in string PolID)
- boolean **removeChildID** (in string PolID)

Sie dienen vor allem dazu, die Beziehungen der einzelnen Policies zueinander festzulegen, und sollen den Policy-Entwicklungsprozeß unterstützen.
Die operationalen Policies bieten zudem noch zwei wichtige Operationen an, mit denen diese aktiviert und deaktiviert werden können.

- void **activate** (in string EOFactoryID, in string agentInitCh)

- void **deactivate** ()

Für die Entwicklung der Oberfläche gestaltete sich besonders die Tatsache, daß eine Spezifikationstrennung zwischen den einzelnen Policy-Typen getroffen wurde, als kompliziert. Es mußten jeweils verschiedene Darstellungsformen für jeden einzelnen Typen gefunden und umgesetzt werden. Auch der Aufwand für die interne Verwaltung und den Zugriff auf die einzelnen Policy-Objekte wurde dadurch größer. Zudem bot die IDL-Spezifikation der Policy Factory keine Operation, um einen direkten Zugriff auf eine Policy zu ermöglichen. Es war jedesmal eine vorherige Abfrage des Typs notwendig.

2.1.2 Benutzeroberfläche

Grundlegender Aufbau Die Benutzeroberfläche (s. Abbildung 5) ist vertikal in zwei Bereiche aufgeteilt. Im linken Bereich werden mittels eines Baumes alle bereits kreierten Policies dargestellt. Hierbei wird eine strukturierte Darstellung vorgenommen, indem die Policies nach ihrem Typ getrennt im Baum eingehängt werden. Die Wurzel (Root) des Baumes trägt die Aufschrift *Policies* an der drei Unterknoten mit den Beschriftungen *Strategic*, *Goaloriented* und *Operational* angehängt sind. Sie werden fest vom Programm vorgegeben und können vom Benutzer auch nicht verändert werden. Unter diesen drei Knoten werden dann die Policies gemäß ihres Typs eingehängt. Dabei wird die Policy durch ein Endblatt (Leaf) mit der Aufschrift ihrer eindeutigen ID im Baum visualisiert. Im rechten Bereich der Oberfläche werden dem Benutzer je nach Modus, verschiedene

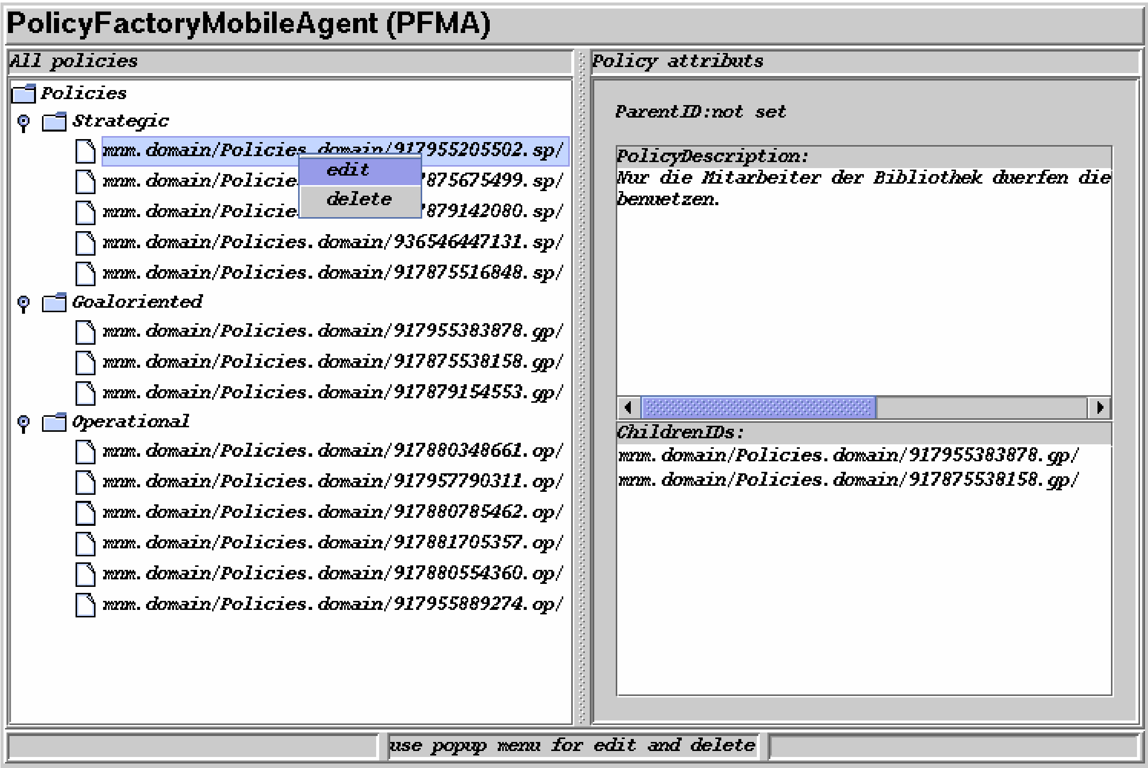

Abbildung 5: Die Benutzeroberfläche des PFMAs

Teiloberflächen (JPanels) bereitgestellt (s.u.). Von diesen aus kann er z.B. eine neue Policy erstellen oder alle Attribute einer selektierten Policy auf einen Blick betrachten.
Welcher Modus gerade aktiv ist, bestimmt der Benutzer interaktiv mit der Maus, indem er einen Knoten oder ein Blatt, des im linken Bereich der Oberfläche dargestellten Policy-Baumes, selektiert oder dort über ein Popup-Menü eine Operation aufruft (s. Abbildung 5). Dabei gilt:

- wird einer der Knoten *Policies*, *Strategic*, *Goaloriented* und *Operational* selektiert, so tritt kein Ereignis ein;

- wird ein *Endblatt*, das eine Policy representiert, selektiert, so werden im rechten Bereich der Oberfläche alle Attribute der entsprechenden Policy dargestellt.

Alle Elemente des Policy-Baumes (Wurzel, Knoten und Blätter) offerieren über ein Popup-Menü weitere Operationen. Diese unterscheiden sich je nach Art des Elements. Generell werden drei Arten unterschieden:

- Die *Wurzel* des Baumes mit der Aufschrift *Policies*. Von hier aus kann die Operation **reload** aufgerufen werden. Dies bewirkt, daß der Policy-Baum neu aufgebaut wird, indem eine neue Anfrage an die Policy Factory (**listAllPolicies** ()) gerichtet wird, und der aktuelle Zustand erfragt wird. Im Normalfall ist das nicht notwendig, denn intern werden alle neu erstellten und zerstörten Policies verwaltet und der Baum danach aktualisiert, so daß immer der aktuelle Zustand zu sehen ist.

- Die drei *Strukturknoten* mit den Aufschriften *Strategic*, *Goaloriented* und *Operational*. Hier steht die Operation **new** zur Verfügung. Wird diese aufgerufen, so hat der Benutzer die Möglichkeit, eine neue Policy zu erstellen. Je nachdem, ob *Strategic*, *Goaloriented* oder *Operational* ausgewählt wurde, erscheint im rechten Teil der Benutzeroberfläche die entsprechende Maske, mit dessen Hilfe eine Policy des gewählten Typs kreiert werden kann.

- Ein *Endblatt*, das eine Policy representiert. Hier stehen die Operationen **edit** und **delete** zur Verfügung, unabhängig vom Typ der Policy. **edit** zeigt im rechten Bereich der Oberfläche das entsprechende Formular, mit dem alle editierbaren Attribute der entsprechenden Policy verändert werden können. Die Operation **delete** zerstört (nach einer Abfrage) die selektierte Policy.

2.1.3 Kreieren einer Policy

Strategische Policy Um eine strategische Policy zu kreieren, selektiert man im Policy-Baum den Knoten *Strategic*, drückt die rechte Maustaste und wählt im erscheinenden Popup-Menü die Operation **new** aus. Im rechten Teil des PFMA-Applets wird die Eingabemaske (s. Abbildung 6) sichtbar.
Im Eingabefeld mit der Aufschrift “PolicyDescription” muß die Beschreibung der Policy eingegeben werden.

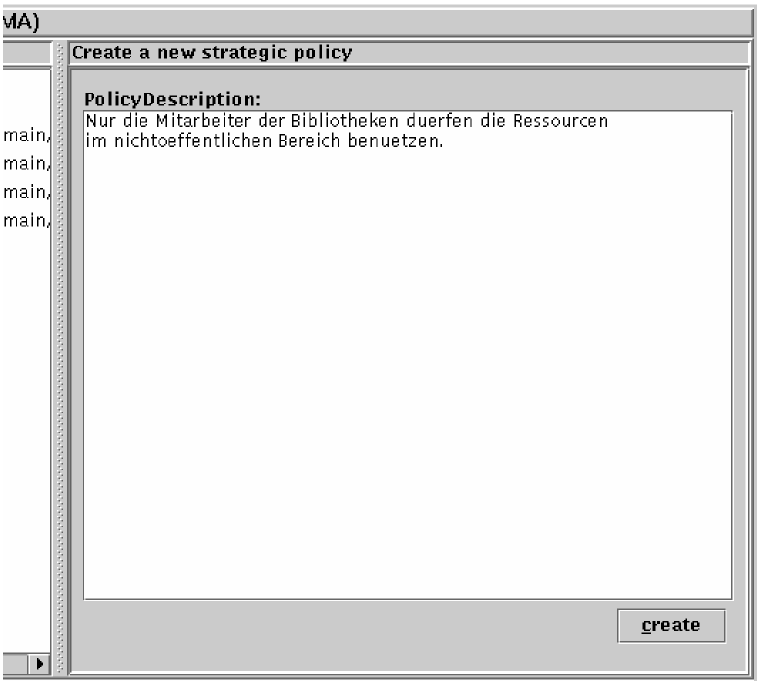

Abbildung 6: Eingabemaske zum Kreieren einer strategischen Policy

Der Text ist nicht an eine Syntax gebunden und wird auch nicht interpretiert. Durch Betätigen des Knopfes **create** wird die Policy erzeugt. Gelingt dies nicht, so wird eine Fehlermeldung zurückgeliefert.

Zielorientierte Policy Um eine zielorientierte Policy zu kreieren, selektiert man im Policy-Baum den Knoten *Goalorientated*, drückt die rechte Maustaste und wählt im erscheinenden Popup-Menü die Operation **new** aus. Im rechten Teil des PFMA-Applets wird die Eingabemaske, wie sie in Abbildung 7 zu sehen ist, sichtbar. Es stehen insgesamt sieben Eingabefelder und eine Auswahlmöglichkeit zur Verfügung, deren

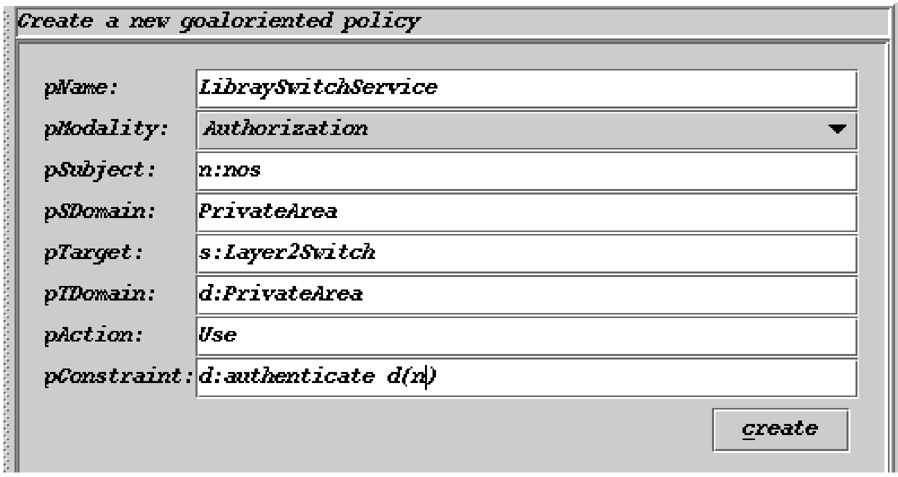

Abbildung 7: Eingabemaske zum Kreieren einer zielorientierten Policy

Bedeutung in folgender Tabelle kurz zusammen gefaßt ist (vgl. auch [Radi 98]).

Name	Bedeutung
pName	Name der Zielorientierten Policy
pModality	Modalität der Policy (Authorization oder Obligation)
pSubject	Typ derjenigen Objekte, die als Subjekte der Policy auftreten
pSDomain	Domäne der Subjekte
pTarget	Typ derjenigen Objekte, die als Zielobjekte der Policy auftreten
pTDomain	Domäne der Zielobjekte
pAction	Aktionen der Policy, die von den Subjekten aufgerufen werden dürfen oder müssen.
pConstraint	Bedingungen, die vor der Anwendung einer Policy erfüllt sein müssen

Durch Betätigen des Knopfes **create** wird eine neue zielorientierte Policy erstellt. Dabei wird die Plausibilität der eingegebenen Inhalte nicht überprüft!

Operationale Policy Um eine operationale Policy zu kreieren, selektiert man im Policy-Baum den Knoten *Operational*, drückt die rechte Maustaste und wählt im erscheinenden Popup-Menü die Operation **new** aus. Im rechten Teil des PFMA-Applets wird die Eingabemaske, wie sie in Abbildung 8 zu sehen ist, sichtbar.
Im Eingabefeld mit der Aufschrift "PolicyDescription in PDL" kann der Benutzer die Beschreibung der Policy eingeben. Dabei muß er sich an die PDL-Syntax halten (vgl. [Radi 98]). Das Programm stellt standardmäßig nur einen groben Syntax-Rahmen zur Verfügung. Der Benutzer muß diesen vervollständigen. Um zu überprüfen, ob die eingegebene Policy-Beschreibung der PDL-Syntax auch entspricht, kann man einen Syntax-Check durchführen lassen. Dies erreicht man, indem man den Knopf **check PDL** betätigt. Daraufhin wird die Syntax der eingegebenen Policy-Description überprüft und das Ergebnis in einem eigenen Fenster angezeigt. Wurde ein Fehler gefunden, so wird seine Position im Text angegeben. Durch Betätigen des Knopfes **create** wird eine neue operationale Policy erstellt. Diese ist in diesem Augenblick noch nicht aktiviert. Dies kann jedoch, wie im folgenden Kapitel 2.1.4 beschieben wird, über die Operation **edit** der Policy erreicht werden.

2.1.4 Ändern der Attribute einer Policy

Das Ändern von Policy-Attributen ist jederzeit möglich. Hierfür muß zuerst die Policy im Policy-Baum selektiert und im Popup-Menü die Operation **edit** ausgewählt werden. Daraufhin erscheint im rechten Teil des PFMA-Applets die Eingabemaske, die in Abbildung 9 zu sehen ist. Wie in Kapitel 2.1.1 bereits erwähnt, sind nur wenige Policy-Attribute änderbar. Diese Eingabemaske bietet zunächst für alle Policies die Möglichkeit

- die ParentID und

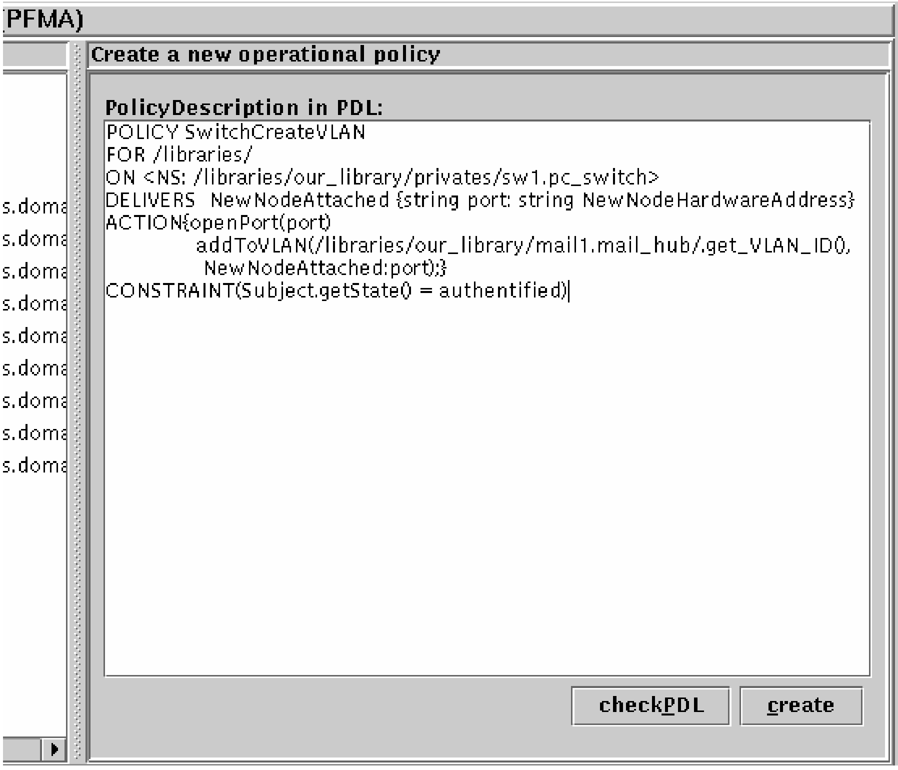

Abbildung 8: Eingabemaske zum Kreieren einer operationalen Policy

- eine ChildID

zu setzen. In den entsprechenden Auswahlboxen sind alle *PolicyIDs* verfügbar, von denen eine ausgewählt wer-

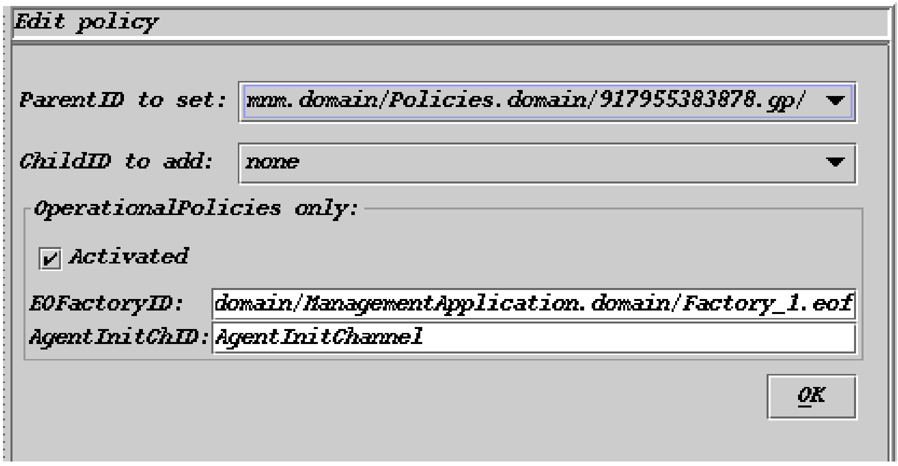

Abbildung 9: Eingabemaske zur Modifizierung von Policy-Attributen

den kann. War bereits eine *ParentID* für die aktuelle Policy gesetzt, so wird diese durch den neu eingestellten Wert ersetzt (Eine Policy darf nur e i n e Policy als Vater haben). Dagegen wird die eingestellte *ChildID* in der Liste der *Children Policies* eingetragen (Eine Policy darf mehrere *Children Policies* haben).
Operationale Policies können außerdem über diese Eingabemaske aktiviert und deaktiviert werden. Hierfür stehen diesen drei zusätzliche Oberflächenkomponenten zur Verfügung: die Checkbox mit der Aufschrift "Activated" und die zwei Eingabefelder "EOFactoryID" und "AgentInitChID". Die Checkbox "Activated" zeigt beim Erscheinen der Maske den aktuellen Zustand der Policy. Ist diese markiert, so ist die Policy aktiv und umgekehrt. Möchte man eine nicht aktive Policy aktivieren, so wählt man die Auswahlbox an. Dadurch werden die zwei Eingabefelder für eine Benutzereingabe freigegeben. Um die Policy zu aktivieren, sind zusätzlich

zwei Parameter erforderlich, die in den entsprechenden Eingabefeldern einzugeben sind. Der erste Parameter **EOFactoryID** gibt die eindeutige ObjectID des EnforcementObectFactoryMobileAgents (EOFMA) an, auf den das entsprechende EnforcementObject zu dieser Policy instanziiert wird. Diese ID wird beim Starten eines EOFMAs direkt als Konstruktorparameter eingegeben und sollte für den Fall der Policy-Aktivierung gemerkt werden. Der zweite Parameter **AgentInitChID** bezeichnet den eindeutigen Namen des MASA-Eventchannels, an den sich alle Agenten beim Starten und Beenden am MASA-Agentensystem an- bzw. abmelden müssen. Dieser Name ist zur Zeit **AgentInitChannel** und sollte bekannt sein. Das Programm belegt die beiden Felder mit Standardwerten vor, die der Benutzer entsprechend ändern kann.
Die Einstellungen werden erst durch Betätigen des Knopfes **OK** übernommen. Ein anschließend erscheinendes Dialogfeld gibt Auskunft über den Erfolg oder Mißerfolg der durchgeführten Änderungen.

2.1.5 Anzeigen der Attribute einer Policy

Das Anzeigen der Attribute einer Policy wird dadurch erreicht, indem man im Policy-Baum die entsprechende Policy selektiert. Da es drei verschiedene Policy-Typen gibt, die sowohl gemeinsame als auch typspezifische Attribute aufweisen, konnte auch keine einheitliche Darstellungsweise erreicht werden. Es wurden drei verschiedene Oberflächen entwickelt, jeweils eine für jeden Policy-Typ, die in den Abbildungen 10, 11 und 12 dargestellt sind. Angezeigt werden alle Attribute, auf die über die IDL-Schnittstellen der *PolicyObjects* zugegriffen werden kann s. Kapitel 2.1.1 .

Von Interesse ist insbesondere für operationale Policies, ob gerade Zielobjekte (*TargetObjects*) existieren,

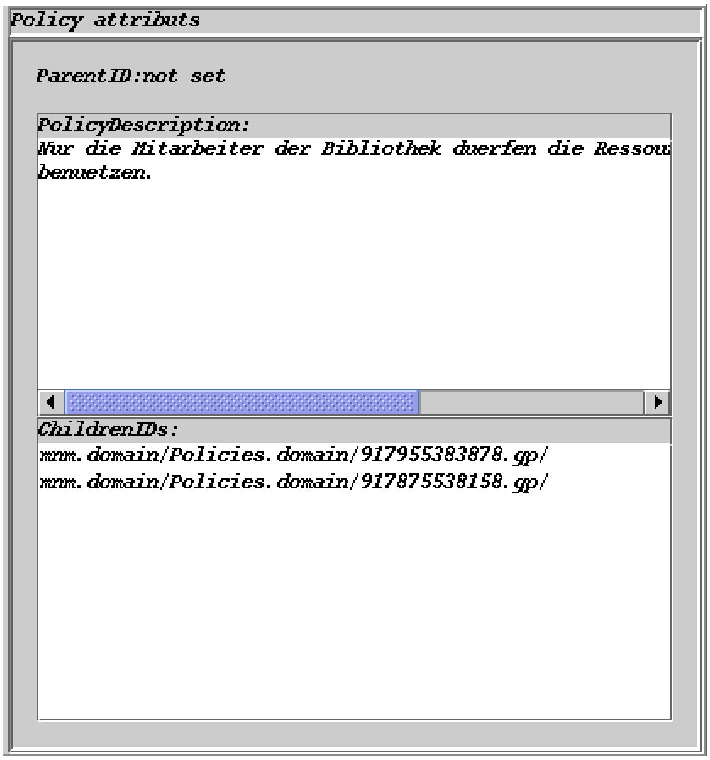

Abbildung 10: Oberflächen zur Anzeige der Attribute einer strategischen Policy

die von dieser Policy betroffen sind. Dieses wird in der entsprechenden Oberfläche (s. Abbildung 12) durch einen Baum visualisiert, der mit der Überschrift "TargetObjects" gekennzeichnet ist. In diesem werden alle aktuell von der Policy betroffenen Zielobjekte durch einen eigenen Knoten mit einer fortlaufenden Nummerierung (1,2,...) dargestellt. Durch Selektieren des Knotens mit der Maus können die zwei Werte **EnfObjID** und **AgentID** ausgelesen werden, durch die solch ein *TargetObject* charakterisiert wird (s. [Radi 98]). Dabei gibt EnfObjID den eindeutigen Identifikator des entsprechenden *EnforcementObjects*, das die Policy durchsetzt, und AgentID den eindeutigen Identifikator des Managementagenten (innerhalb des MASA-Agentensystems), der von der Policy betroffen ist, an.

Abbildung 11: Oberfläche zur Anzeige der Attribute einer zielorientierten Policy

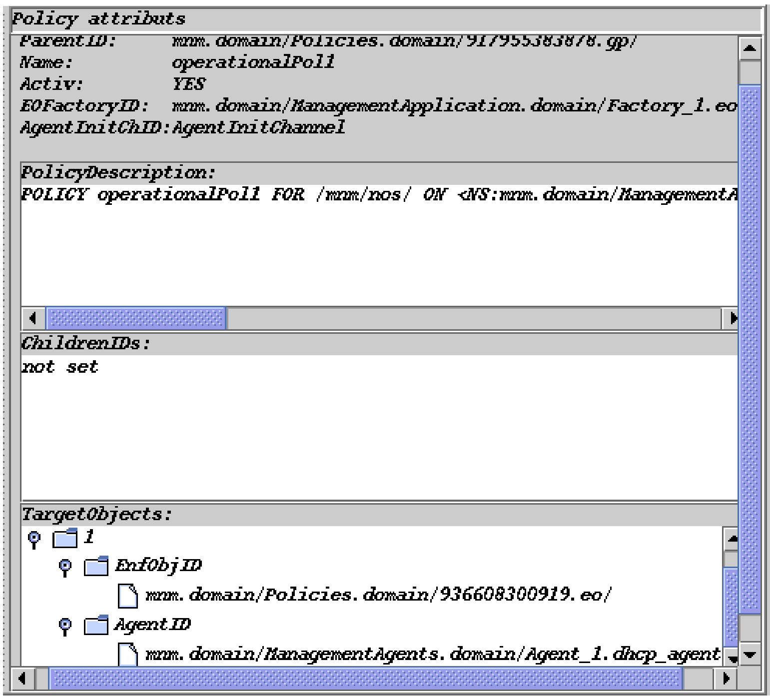

Abbildung 12: Oberfläche zur Anzeige der Attribute einer operationalen Policy

2.2 Der PersistenceServiceStationaryAgent

Der PersistenceServiceStationaryAgent (PSSA) realisiert einen "Persistence Service", der Methoden zum Speichern, Wiederherstellen, Einfügen und Löschen von persistenten Objekten bietet. Hierbei kann der Service sowohl für die persistente Speicherung von Policies im speziellen, als auch von CORBA-Objekten im allgemeinen genutzt werden. Er ist als CORBA-Objekt reallisiert worden und stellt seine Funktionalität über Schnittstellen zur Verfügung.
Den Großteil seiner Funktionalität machen sich die beiden anderen Agenten zunutze, um Policies vor dem Beenden persistent abspeichern und nach dem Start wiederherstellen zu können.
Für die Implementierung der Oberfläche wäre von Interesse, wenn eine allgemeine Verwaltung von Datenbanken (*PersistentObjects* werden in Datenbanken gespeichert und verwaltet) möglich wäre. Es sollten dann Aufgaben wie Erstellen einer neuen Datenbank, Laden, Manipulieren und Umbenennen von bereits existierenden Datenbanken möglich sein.

2.2.1 Die IDL-Schnittstellen

Im folgenden ist die IDL-Spezifikation des *PersistenceServices* zu sehen:

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

Es werden insgesamt 13 Operationen angeboten, von denen die drei Operationen:

- **addElement** (in PersistentObject newObject, in string userName)
- **getAllObjects** (in string userName), und
- **getElement** (in string PID, in string userName)

nicht vom Applet aus aufrufbar sind. Die restlichen zehn können vom Applet aufgerufen werden, wobei mit

- **connect** und **disconnect** eine Verbindung zum PSSA auf- bzw. abgebaut wird.
- **createNewDB** eine neue Datenbank angelegt wird.
- **restore** eine bereits existierende Datenbank wiederhergestellt und geladen wird
- **store** die aktuelle Datenbank unter einem anderen Namen persistent gespeichert wird
- **listAllID** alle IDs der persistenten Objekte der aktuellen Datenbank erhalten werden.
- **deleteElement** das angegebene Element aus der aktuellen Datenbank entfernt wird.

2.2.2 Benutzeroberfläche

Die Benutzeroberfläche des PSSAs (s. Abbildung 13) gliedert sich in drei Bereiche: dem Funktionsauswahl-, dem Aktions- und dem Informationsbereich. Im Funktionsauswahlbereich (rechter Teil der Oberfläche) kann der Benutzer eine Reihe von Operationen auslösen, die in den folgenden Absätzen näher beschrieben werden. Je nach ausgewählter Funktion erscheint im Aktions-Bereich (linker Teil) die entsprechende Maske (JPanel), von der der Benutzer weitere Aktionen ausführen kann. Im Informations-Bereich (unterer Teil) werden eine Reihe von Meldungen eingeblendet, die den aktuellen Status bzw. Wert einiger Variablen des PSSAs wiedergeben. Im einzelnen wird der Name des aktuellen Benutzers, der Name der aktuellen Datenbank und die Tatsache, ob bereits eine Verbindung zu einer Datenbank besteht oder nicht (DB allocated) angezeigt.

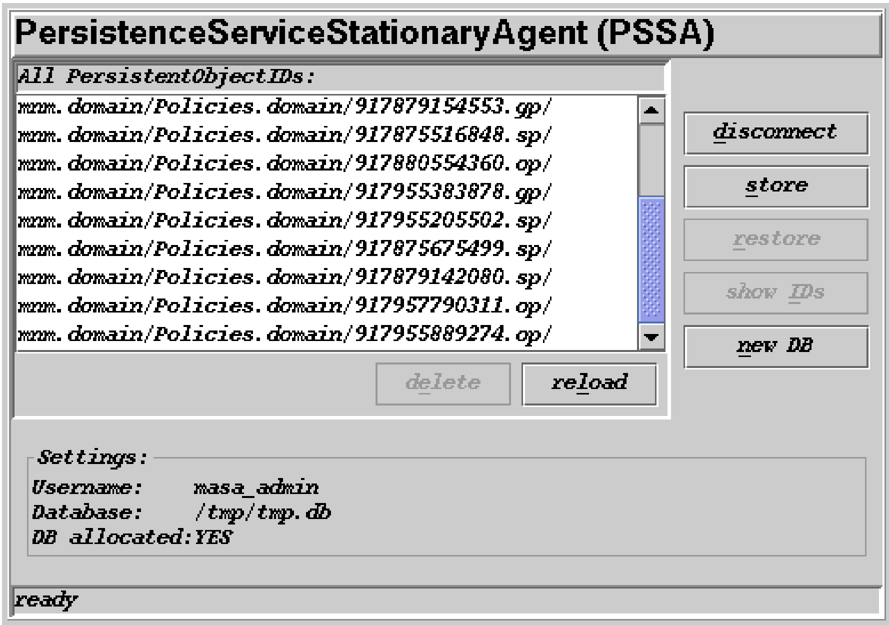

Abbildung 13: Die Benutzeroberfläche des PSSAs

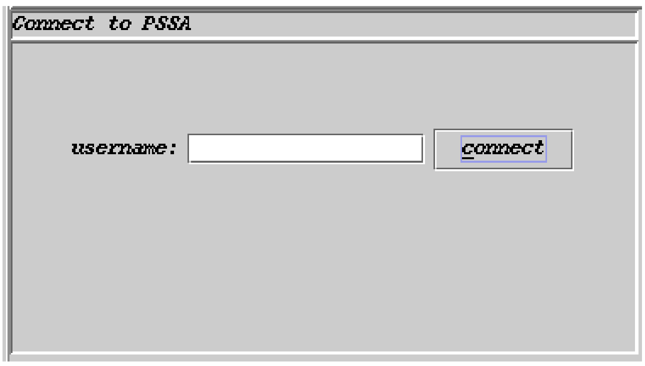

Abbildung 14: Anmeldemaske des PSSAs

Eine Verbindung zum PSSA aufbauen (connect) Ist der Benutzer gerade nicht mit dem PSSA verbunden, so wird standardmäßig die Maske, die in Abbildung 14 zu sehen ist, eingeblendet. Von dieser aus kann man wieder eine Verbindung aufbauen. Hierzu wird nur ein Benutzername erwartet, der im Feld mit der Aufschrift "username" eingegeben werden muß. Anschließend wird durch Betätigen des Knopfes **connect** ein Verbindungsversuch gestartet. Das eingegebene Kennwort wird vom PSSA derzeit nicht auf Zugriffsberechtigung überprüft. Es dient lediglich der internen Benutzerverwaltung und -unterscheidung. Sind mehrere Benutzer beim PSSA angemeldet, so ist insbesondere die restore-Operation nicht mehr möglich (Ein-Datenbank-Betrieb). Die Verbindung kann jederzeit abgebrochen werden, indem man den Knopf **disconnect** (s. Abbildung 13) betätigt.

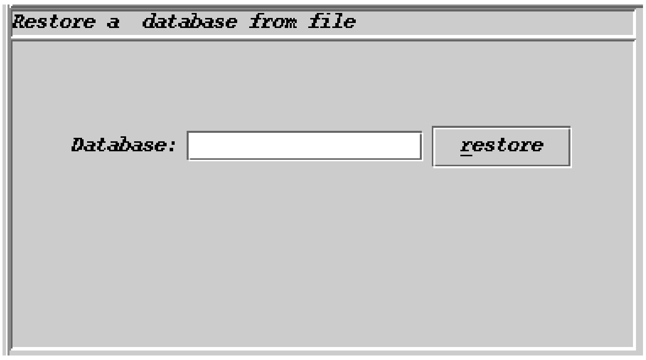

Abbildung 15: Maske zur Wiederherstellung einer Datenbank

Eine Datenbank wiederherstellen (restore) Voraussetzung für das Ausführen der Operation "restore" ist eine Verbindung zum PSSA (s. Kapitel 2.2.2). Weiterhin darf diese Operation nicht ausgeführt werden, wenn ein weiterer Benutzer eine Verbindung zum PSSA aufrecht erhält. In der Applet-Implementierung wird dies dadurch verhindert, daß der Knopf **restore** in diesen Fällen nicht für den Benutzer zur Auswahl steht (disabled). Sind diese beiden Voraussetzungen erfüllt, so erscheint durch Betätigen des Knopf **restore** (s. Abbildung 13) die Maske, die in Abbildung 15 zu sehen ist. Im Eingabefeld "Database" muß der vollständige Pfad zu einer Datei angegeben werden, die bereits vom PSSA erstellt worden ist. Wenn z.B. im Verzeichnis **/tmp** die Datenbank mit dem Namen **tmp.db** existiert, so würde die Eingabe wie folgt lauten : **/tmp/tmp.db**. Durch Betätigen des Knopfes **restore** (innerhalb der Maske) wird die angegebene Datenbank eingelesen.

Aktuelle Datenbank speichern (store) Um die Operation "store" durchführen zu können, muß erstens eine Verbindung zum PSSA aufgebaut und zweitens eine Datenbank mittels "restore" (s.o.) bereits geladen worden sein. Erst dann ist auch der entsprechende Knopf "store" von der Oberfläche des PSSAs (s. Abbildung 13) aus für den Benutzer auswählbar.
Möchte man die aktuelle Datenbank abspeichern, so betätigt man den Knopf **store**, woraufhin die in Abbildung 16 dargestellte Maske dem Benutzer zur Verfügung steht. Im Eingabefeld "Database" wird der vollständige Pfad einschließlich Dateiname erwartet, unter dem die Datenbank gespeichert werden soll (z.B. **/tmp/mnmpolicies.db**). Das Programm gibt standardmäßig den Pfad der aktuellen Datenbank vor, so daß mit Betätigen des Knopfes **store** (innerhalb der gleichen Maske) eventuelle Änderungen an der Datenbank gespeichert werden können. Gibt man einen anderen Namen an, so wird eine Kopie der aktuellen Datenbank unter dem angegebene Pfad angelegt.

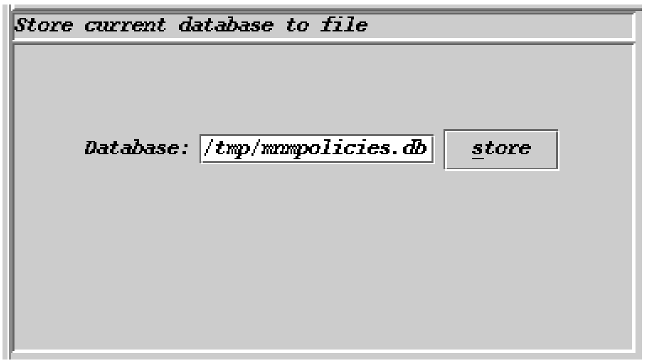

Abbildung 16: Maske zur Speicherung einer Datenbank

Anlegen einer neuen Datenbank (new DB) Einzige Voraussetzung, um eine neue Datenbank zu erstellen, ist eine bereits aufgenommene Verbindung zum PSSA. Ist dies der Fall, so steht dem Benutzer jederzeit und unabhängig von der Anzahl der Benutzer, die eine Verbindung zum PSSA haben, die Möglichkeit zur Verfügung, eine neue Datenbank zu erstellen. Hierfür wird der Knopf **new DB** (s. Abbildung 13) betätigt. In der Eingabemaske (s. Abbildung 17) wird von dem Benutzer erwartet, einen vollständigen Pfad einschließlich

Dateinamen im Eingabefeld “Filename” einzugeben (z.B. **/tmp/policies/mnmpolicies.db**). Durch Betätigen des Knopfes **create** wird die neue Datenbank erstellt, aber nicht geladen! Mittels “restore” (s.o.) kann diese dann, sofern zu diesem Zeitpunkt erlaubt, geladen werden.

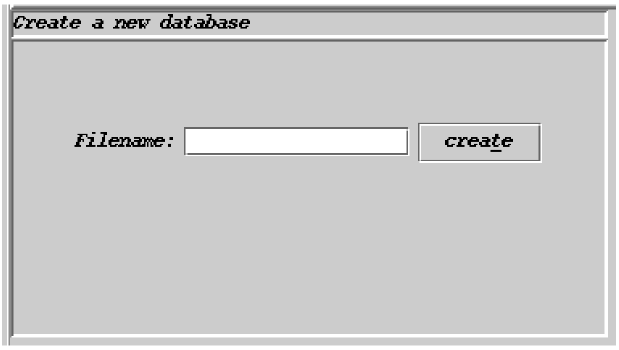

Abbildung 17: Maske zum Anlegen einer neuen Datenbank

Anzeigen der Elemente einer Datenbank (show IDs) Eine aktive Verbindung zum PSSA und eine mittels “restore” geladene Datenbank sind die zwei Voraussetzungen, um die IDs aller in einer solchen Datenbank enthaltenen Objekte anzeigen zu lassen. Sind diese zwei Vorgänge getätigt worden, so wird automatisch die Oberfläche, die in Abbildung 18 zu sehen ist, dem Benutzer präsentiert. Dort werden alle IDs der *Persistent Objects* der aktuellen Datenbank in einer Auswahlliste angezeigt. Durch Selektion einer ID und anschließendem Betätigen des Knopfes **delete** (s. Abbildung 18) kann der Benutzer ein ganzes Objekt aus der Datenbank entfernen. Durch Betätigen von “reload”, kann der Benutzer manuell eine Aktuallisierung der Darstellung erreichen.

Abbildung 18: Anzeigen aller IDs

Der Startvorgang Aus Gründen der Flexibilität und einfacher Bedienbarkeit für nicht versierte Benutzer wurden einige Abläufe während des Startvorgangs des Applets automatisiert. So versucht das Applet beim Starten (in der init-Methode) eine Verbindung zum PSSA aufzubauen und sich mit dem Benutzernamen **masa_admin** anzumelden. Wurde dies erfolgreich abgeschlossen, so ruft es die Operation **restore** *(/tmp/tmp.db,masa_admin)* des PSSAs auf, womit der PSSA aufgefordert wird, die Datenbank **tmp.db** im Verzeichnis **/tmp/** wiederherzustellen und zu laden. Traten keine Fehler auf, so kann der Benutzer gleich mit dieser Datenbank arbeiten. Möchte man diesen Effekt nicht, so reicht es, die Datei **tmp.db** aus **/tmp/** zu entfernen. Das Applet fordert dann den Benutzer auf, eine Datenbank auszuwählen.
Im allgemeinen gilt für das erfolgreiche Arbeiten mit dem PSSA folgende Reihenfolge:

1. Unter einem Benutzernamen beim PSSA anmelden.
2. Eine Datenbank mittels “restore” wiederherzustellen und zu laden.
3. Erst dann mit der Datenbank arbeiten.

2.3 Der EnforcementObjectFactoryMobileAgent

Auf dem EnforcementObjectFactoryMobileAgent (EOFMA) werden sowohl die *EnforcementObjectFactory* als auch die *EnforcementObjects* instanziiert. Da auch sie als CORBA-Objekte realisiert wurden, bieten sie über Schnittstellen Clients ihre Funktionalität an. Ein Teil dieser Funktionalität, ist dazu gedacht, dem PFMA (s. Kapitel 2.1) das Aktivieren, Deaktivieren und Durchsetzen von operationalen Policies zu ermöglichen. Für die Entwicklung der Oberfläche stellte sich somit die Frage, welche der angebotenen IDL-Operationen für einen Benutzer von Interesse sind, bzw. welche dieser ausführen darf. Aus diesen Überlegungen heraus entstand eine Oberfläche, die nur ein Monitoring des EOFMAs erlaubt. Dies bedeutet, daß nur ein Auslesen von bestimmten Werten ermöglicht wird. Ein direktes Einwirken auf den EOFMA ist nur über den PFMA möglich.

2.3.1 Die IDL-Schnittstellen

Die EnforcementObjectFactory Im folgenden ist die IDL-Spezifikation der *EnforcementObjectFactory* angegeben:

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

Es werden insgesamt vier Operationen angeboten, von denen die beiden ersten – zum Erstellen bzw. Zerstören von *EnforcementObjects* – nicht über das Applet des EOFMAs aufrufbar sind. Sie sind auch für den Benutzer nicht direkt von Bedeutung, da sie indirekt über den PFMA (s. Kapitel 2.1) durch Aktivieren und Deaktivieren von operationalen Policies aufgerufen werden können. Interessanter für die Oberflächengestalltung sind die zwei letzten Operationen. Mit dem Aufruf von **listAllEnfObj** () ist es möglich, eine Liste aller *EnforcementObjects* zu erhalten, die momentan auf dem Agenten existieren. Mit dem Aufruf von **getEnfObj** (in string EnfObjID) kann dann auf ein konkretes *EnforcementObject* zugegriffen und sein Zustand ausgelesen werden.

Die EnforcementObjects Ein *EnforcementObject* (EO) ist durch folgendes IDL-Interface charakterisiert:

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

Die beiden Operationen **startMonitoring ()** und **stopMonitoring ()** beinflussen das Verhalten des EOs direkt, indem sie das EO veranlassen, sich an den *Eventchannel* des Zielobjekts (*Targetobject*) zu binden bzw. zu trennen. Sie werden intern innerhalb des EOFMAs aufgerufen. Ihre Implementierung in der Bedienoberfläche des Agenten war somit wenig sinnvoll und wurde auch nicht durchgeführt.
Die restlichen sieben Operationen erlauben das Auslesen von verschiedenen Attributen eines EOs und sind geeignet, um einen Überblick über das Objekt zu erhalten. Sie alle wurden implementiert.

2.3.2 Benutzeroberfläche

Die Benutzeroberfläche des EOFMAs (s. Abbildung 19) ist horizontal in zwei Bereiche aufgeteilt. Im oberen Bereich werden mittels eines Baumes alle *EnforcementObjects*, die momentan im Agenten instanziiert sind, dargestellt. Die Wurzel des Baumes trägt die Aufschrift “activ EOs”, unter der alle EOs als Blätter eingehängt werden. Zur Indentifizierung der EOs wird ihre eindeutige ID verwendet. Sind zum aktuellen Zeitpunkt keine EOs auf dem EOFMA instanziiert, so ist nur die Wurzel sichtbar.
Durch Selektieren eines EOs aus dem Baum erscheinen im unteren Bereich der Oberfläche alle Werte, die

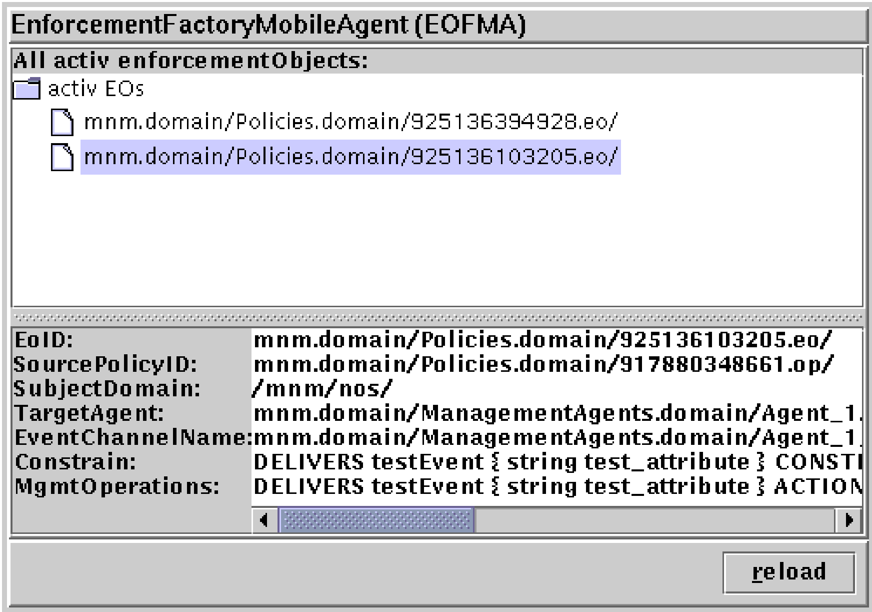

Abbildung 19: Die Benutzeroberfläche des EOFMAs

über seine IDL-Schnittstelle ausgelesen werden können (vgl Kapitel 2.3.1). Dabei bedeuten (s. [Radi 98]):

- **EoID**: Die ID des selektierten *EnforcementObjects*.
- **SourcePolicyID**: Die eindeutige ID der operationalen Policy, die zum Kreieren des EOs geführt hat.
- **SubjectDomain**: Die Heimatdomäne des von der Policy betroffenen Subjekts.
- **TargetAgent**: Die ID des Managementagenten, der vom EO beobachtet wird.
- **EventChannelName**: Der Name des Eventchannels, an dem das EO sich momentan gebunden hat.
- **Constrain**: Das Constraint gemäß der PDL-Syntax aus der Policy-Definition.
- **MgmtOperations**: Die Managementoperationen, die beim Auftreten eines Events auszuführen sind (in PDL-Syntax).

Während der Initialisierung des Applets wird durch eine Anfrage an den EOFMA der aktuelle Zustand erfragt und der EO-Baum aufgebaut. Durch den Knopf **reload** (s. Abbildung 19) kann dann zu einem späteren Zeitpunkt immer wieder eine neue Anfrage und somit eine Aktualisierung der Darstellung veranlaßt werden.

3 Quellcode Änderungen

Gegenüber dem Quellcode von [Radi 98] wurden einige Änderungen und Erweiterungen vorgenommen, die notwendig erschienen. Im folgenden werden diese aufgezählt.

3.1 In PolicyFactoryMobileAgent.java

Diese Klasse wurde um einen zweiten, parameterlosen Konstruktor erweitert. Wird dieser zur Erzeugung der Klasse benutzt, so werden alle Variablen, die in folgender Tabelle zu sehen sind, mit den angegebenen Defaultwerten vorbelegt.

Variable	**Wert**
ObjectID	mnm.domain/ManagementApplication.domain/PolFactory_1.pf
persistenceServiceName	mnm.domain/ManagementApplication.domain/PersService_1.ps
persistentDatabaseName	**/tmp/tmp.db**
policyDomainPrefix	mnm.domain/Policies.domain/

Der Konstruktor sollte nur zu Testzwecken aufgerufen werden, um ein schnelleres Starten des Agenten über das AgentSystemApplet des Agentensystems, zu erreichen. Im anderen Konstruktor können dann individuelle Werte für diese Variablen eingestellt werden.

```
public PolicyFactoryMobileAgent() {
    String oID = "mnm.domain/ManagementApplication.domain/PolFactory_1.pf";
    this.ObjectID = oID;
    String pDP = "mnm.domain/Policies.domain/";
    this.policyDomainPrefix = pDP;
    String pDBName = "/tmp/tmp.db";
    this.persistentDatabaseName = pDBName;
    String pSName = "mnm.domain/ManagementApplication.domain/PersService_1.ps";
    this.persistenceServiceName = pSName;
    this.local_policyObjectDatabase = new java.util.Hashtable();
    this._PersistenceService = null;
    this.prev_started = false;
}
```

3.2 In persistence.idl

Das IDL-Interface des PersistenceServices (s. Kapitel 2.2.1) wurde um die Operation **createNewDB** erweitert.

```
void createNewDB (in string fileName, in string userName)
  raises (UserNotFound, StoreFailure);
```

Durch ihren Aufruf wird eine neue Datenbank unter dem angegebenen Namen “fileName” angelegt. Sie kann nur dann ausgeführt werden, wenn der mit “userName” angegebene Benutzer auch tatsächlich eine aktive Verbindung zum PSSA hat.
Sie wird in *PersistenceServiceStationaryAgent.java* implementiert.

3.3 In PersistenceServiceStationaryAgent.java

Diese Klasse wurde um eine Methode und einen zweiten, parameterlosen Konstruktor erweitert.
Die Methode **runDomainBuilder()** startet den DomainBuilder, der bisher manuell aufgerufen werden mußte, bevor alle drei Agenten gestartet werden konnten.

```
public void runDomainBuilder (){
    DomainBuilder dBuilder = new DomainBuilder ();
    dBuilder.run ();
}
```

Der parameterlose Konstruktor belegt die Variable ObjectID mit einem Defaultwert vor, wie in nachfolgender Tabelle zu sehen. Gleichzeitig wird in diesem die Methode **runDomainBuilder ()** aufgerufen. Da der PSSA auf jeden Fall als erster der drei Agenten gestartet werden muß, wird mit dem Aufruf der Methode an dieser

Stelle sichergestellt, daß der DomainBuilder bereits gestartet wurde. Ein manuelles Starten des DomainBuilders entfällt somit.

Variable	Wert
ObjectID	mnm.domain/ManagementApplication.domain/PersService_1.ps

```
public PersistenceServiceStationaryAgent() {
  runDomainBuilder();
  String oID = "mnm.domain/ManagementApplication.domain/PersService_1.ps";
  this.ObjectID = oID;
  this.PersistentObjectDB = null;
  this.databaseName = "not set";
  this.usersList = new java.util.Vector(1,1);
  this.database_allocated = false;
}
```

3.4 In EnforcementObjectFactoryMobileAgent.java

Diese Klasse wurde um einen zweiten, parameterlosen Konstruktor erweitert. Wird dieser zur Erzeugung der Klasse benutzt, so werden alle Variablen, die in folgender Tabelle zu sehen sind, mit Defaultwerten vorbelegt:

Variable	Wert
ObjectID	mnm.domain/ManagementApplication.domain/Factory_1.eof
persistenceServiceName	mnm.domain/ManagementApplication.domain/PersService_1.ps
persistentDatabaseName	/tmp/tmp.db
managementAgentType	.dhcp_agent
enforcementObjectDomainPrefix	mnm.domain/Policies.domain/
nomadicSystemDomain	mnm.domain/ManagedClients.domain/

Er sollte ebenfalls nur zu Testzwecken aufgerufen werden.

```
public EnforcementObjectFactoryMobileAgent() {
    String oID = "mnm.domain/ManagementApplication.domain/Factory_1.eof";
    this.ObjectID = oID;
    String eODP = "mnm.domain/Policies.domain/";
    this.enforcementObjectDomainPrefix = eODP;
    String nSD = "mnm.domain/ManagedClients.domain/";
    this.nomadicSystemDomain = nSD;
    String pSName = "mnm.domain/ManagementApplication.domain/PersService_1.ps";
    this.persistenceServiceName = pSName;
    String pDBName = "/tmp/tmp.db";
    this.persistentDatabaseName = pDBName;
    String mAT = ".dhcp_agent";
    this.managementAgentType = mAT;
    this.local_enforcementObjectDatabase = new java.util.Hashtable();
    this.prev_started = false;
}
```

A README-Auszug

Im folgenden, ist ein Auszug aus dem README zu sehen, das das Starten der Policyagenten über das Applet des Agentensystems MASA beschreibt.

```
No enviroment variable has to be set. (normally)

The steps 1-6 have to be done only in case the AgentSystem,
AgentInitChannel and the NamingService are NOT running:

1. change to the masa-directory
(the same as specified in AGENT_SYSTEM_CODE_BASE)

2. change to the masa-bin directory
   >cd bin/

3. start the osagent in an own shell
   >./masaScript start osagent

   Look at the shell. You must see something like this:
        Bound to the following UDP port  : 14383
        Bound to the following interfaces:
        Address: 129.187.214.2  Subnet: 255.255.255.0  Broadcast: 129.187.214.255

4. start the NamingService in an own shell
   >./masaScript start naming
   Look at the shell. You must see something like this:
        NamingWebServerinitWebServer(): ior length404
        NamingWebServer.waitForRequests(): localhost:sunheger2.nm.informatik.uni-muenchen.de/129.187.214.2
        configured host: sunheger2
        NamingWebServer.waitForRequests(): Wait for requests, listening at port 5383.

   Remember the port!(5383)

5. start the AgentInitChannel in an own shell
   >./masaScript start channel

        NamingWebServer.getIORWWW() get port...done.
        NamingWebServer.getIORWWW(): IOR:000000000000002849444c3a6f72672e6f6d672...
        Created event channel :com.visigenic.vbroker.services.CosEvent.Channel@1dc60e4c
        AgentChannel.main():Channel is up and bound to the name AgentInitChannel

   Remember the name of the Channel! (AgentInitChannel).
   If you want to activate an operational Policy, then you have to now this name.

6. start the AgentSystemt in an own shell
   >./masaScript start system

7.  open a new shell and start hotjava
   >hotjava
   or with log
   >hotjava -log /dev/tty

8. set the URL to:
        http//:hostnameWereAgentSystemIsRunning:NamingWebServerPort
        //(see step4)
   example :  http//:sunheger2:5383
   Now you sould see the masa-AgentSystemApplet (after a view seconds (~>60))

9.Start the PersistenceServiceMobileAgent (PSMA)
   -select the option "Create agent" from the masa-AgentSystemApplet,
   -choose PersistenceService from the Chooseagent-combobox,
   -enter in the table below in the field "Values":
```

```
 mnm.domain/ManagementApplication.domain/PersService_1.ps

(NOTE: You can replace PersService_1.ps with any name you like.
      In this case you have to pass your selection in the constructor
      of PolicyFactoryMobileAgent (step11) and
      EnforcementObjectFactoryMobileAgent (step 10).
 )

-press the RETURN-key!!! veryyy importand,
-and press "create agent".
The PersistenceServiceMobileAgent is now created
and you can access its webpage. To make this do:
     -select the option "Manage agents" from the masa-AgentSystemApplet,
     -select in the table PersistenceServiceMobileAgent
     -press "Show agent's webpage"
if you see the webpage:
     -you can restore a policy database.
     in the Mopa-directory you can find an exampledatabase
     "tmp.db" and restore it.

10.Start the EnforcementObjectFactoryMobileAgent (EOFMA)
   -select the option "Create agent" from the masa-AgentSystemApplet,
   -choose EnforcementObjectFactory  from the Chooseagent-combobox
   -enter in the table below in the field "Values":

        mnm.domain/ManagementApplication.domain/Factory_1.eof
        mnm.domain/ManagementApplication.domain/PersService_1.ps //same as in step 9
        /tmp/tmp.db  //or an other database-name
        dhcp_agent   //or an other agenttyp
        mnm.domain/Policies.domain/
        mnm.domain/ManagedClients.domain/

   (NOTE: You can replace Factory_1.eof in
         "mnm.domain/ManagementApplication.domain/Factory_1.eof"
         with any name you like.
         If you want to activate an operational Policy,
         then you have to now this name.
    )

   -press the RETURN-key!!! veryyy importand
   -and press "create agent"
   the EnforcementObjectFactoryMobileAgent is now created
   and you can access its webpage.
   To make this do:
        -select the option "Manage agents" from the masa-AgentSystemApplet,
        -select in the table EnforcementObjectFactoryMobileAgent,
        -press "Show agent's webpage".

11.Start the  PolicyFactoryMobileAgent (PFMA)
   -select the option Create agent from the masa-AgentSystemApplet,
   -choose PolicyFactory  from the Chooseagent-combobox,
   -enter in the table below in field "Values":

        mnm.domain/ManagementApplication.domain/PolFactory_1.pf
        mnm.domain/ManagementApplication.domain/PersService_1.ps
        /tmp/tmp.db  //or an other database-name
        mnm.domain/Policies.domain/

   (NOTES:
        a)You can replace PolFactory_1.pf in
        "mnm.domain/ManagementApplication.domain/PolFactory_1.pf"
        with any name you like.
```

```
        b)"mnm.domain/ManagementApplication.domain/PersService_1.ps"
        must be the same as in step 9
        c)the database "/tmp/tmp.db" (or your selection )
        must be restored at this moment by the PSMA.
    )

   -press the RETURN-key!!! veryyy importand,
   -and press "create agent".
   The PolicyFactoryMobileAgent is now created and you can
   access its webpage.
   To make this do:
        -select the option "Manage agents" from the masa-AgentSystemApplet,
        -select in the table PolicyFactoryMobileAgent,
        -press Show agent's webpage or doubleclick

13.Start the JMAPI-DB-Server by  typing the following command
   from the MoPA-directory:

   >gmake STARTJM

   For adding new Nomadic Nodes to the JMAPI-database
   (mac_address and domain) use the Applet by Christian Schiller
   by  typing the following command from the MoPA-directory

   >gmake JMAPPLET

   (Note:At this moment(31.03.1999) the JMAPI-DB-Server does not work.)


That's all.
```

Literatur

[Allg 98] ALLGEYER, P.: *Entwicklung und Implementierung einer Managementschnittstelle für einen PC-basierten Switch/Router.* Diploma Thesis, Technische Universität München, August 1998.

[Demm 98] DEMMEL, S.: *Implementierung eines portierbaren Java-DHCP-Servers mit einer CORBA-Managementschnittstelle.* Fortgeschrittenenpraktikum, Ludwig-Maximilians-Universität München, September 1998.

[Flan 96] FLANAGAN, D.: *Java in a Nutshell.* O´Reilly & Associates, Inc., 1996.

[Gerb 99] GERBER, S.: *Entwicklung einer Benutzerschnittstelle mit Java/CORBA für die Mobile Agent System Architecture (MASA).* Systementwicklungsprojekt, Technische Universität München, April 1999.

[GHR 99] GRUSCHKE, B., S. HEILBRONNER und H. REISER: *Mobile Agent System Architecture — Eine Plattform für flexibles IT–Management.* Technischer Bericht 9902, München, August 1999.

[Heil 98a] HEILBRONNER, S.: *Requirements for Policy-based Management of Nomadic Computing Systems.* In: SETHI, A. S. (Herausgeber): *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 98)*, Newark, DE, USA, Oktober 1998. .

[JLS] GOSLING, J., B. JOY und G. STEELE: *The Java Language Specification.* Addison–Wesley, 1996.

[Kemp 98] KEMPTER, B.: *Entwurf eines Java/CORBA-basierten Mobilen Agenten.* Diploma Thesis, Technische Universität München, August 1998.

[Naug 96] NAUGHTON, P.: *Java Handbook.* Osborne McGraw-Hill, 1996.

[Radi 98] RADISIC, I.: *Konzeption eines policy-basierten Konfigurationsmanagements für nomadische Systeme in Intranets.* Diploma Thesis, Ludwig-Maximilians-Universität München, August 1998.

[Sun 98] SUN MICROSYSTEMS, INC.: *Java Development Kit 1.2 Documentation*, 1998.