INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Systementwicklungsprojekt

Entwicklung eines MASA-Agenten zur Konsolidierung von Sicherheits Policies

Bearbeiter:	Bodensteiner Christoph
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Harald Rölle Helmut Reiser

INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Systementwicklungsprojekt

Entwicklung eines MASA-Agenten zur Konsolidierung von Sicherheits Policies

Bearbeiter:	Bodensteiner Christoph
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Harald Rölle Helmut Reiser

Abgabetermin: 15. 02. 2002

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

München, den 15. Februar 2002

....
(Unterschrift des Kandidaten)

Zusammenfassung

Aufgrund der immer größer werdenden Vernetzung von Computersystemen und der damit einhergehenden Komplexität, vollzog sich ein Wandel des Managments hin zu flexiblen und mächtigeren Ansätzen. Durch den Einsatz von mobilen Agentensystemen wollte man neuen Anforderungen des Managments gerecht werden. Die Verwendung von autonomen mobilen Agenten birgt jedoch hinsichtlich der Sicherheit einige Risiken. Die Definition von Rechten für Agenten und deren Laufzeitumgebungen sowie deren Durchsetzung stellt hierbei einen wichtigen Aspekt dar, um den Sicherheitsrisiken gerecht zu werden um den Missbrauch solcher Managmentstrukturen zu verhindern.
Diese Arbeit beschäftigt sich mit dem Managment von Sicherheitspolicies für das MASA Agentensystem und deren Agenten.
Insbesondere wurde ein mögliches Konzept zur automatisierten Verarbeitung von gewünschten Rechten einzelner Agenten entworfen und prototypisch implementiert. Desweiteren wurden Tools zur Modifikation und Integration von Bestandteilen der Sicherheitspolicy für die MASA-Laufzeitumgebung und deren Agenten erstellt, sowie Möglichkeiten zum Managment von Sicherheitspolicies innerhalb einer MASA Region geschaffen. Die Implementierung erfolgte mittels CORBA/JAVA (JDK1.2).

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Aufgrund der immer größer werdenden Vernetzung von Computersystemen und der damit einhergehenden Komplexität, vollzog sich ein Wandel des Managments hin zu flexiblen und mächtigeren Ansätzen. Durch den Einsatz von mobilen Agentensystemen wollte man neuen Anforderungen des Managments gerecht werden.
Die Verwendung von autonomen mobilen Agenten birgt jedoch hinsichtlich der Sicherheit einige Risiken. Die Definition von Rechten für Agenten und deren Laufzeitumgebungen sowie deren Durchsetzung stellt hierbei einen wichtigen Aspekt dar, um den Sicherheitsrisiken gerecht zu werden um den Missbrauch solcher Managmentstrukturen zu verhindern. Um einen reibungslosen Ablauf im realen Einsatz zu gewährleisten, muß eine schnelle und einfache Konfiguration von Sicherheitseinstellungen möglich sein. Die entwickelten Komponenten sollen hierbei einen kleinen Beitrag zur besseren Managebarkeit von Sicherheitseinstellungen für das MASA System leisten.

1.1 Sicherheitspolicies

Eine Sicherheitspolitik definiert im allgemeinen die Sicherheitsheitsrichtlinien eines Systems [HeAb 94]. Diese besteht einerseits aus einer formalen Definition der Autorisation von Subjekten zum genau spezifizierten Zugriff auf Objekte als auch in der Spezifikation der einzusetzenden Maßnahmen. Dabei sollten die Sicherheitspolicy und die Mechanismen für die Durchsetzung getrennte Implementierungen darstellen. Hierbei gilt auch, daß die Größe und Anzahl der eingesetzten Mechanismen für die Definition und der Durchsetzung der Policies, das Managment überproportional erschwert. Insgesamt gesehen, lassen sich zwei verschiedene Typen von Sicherheits Policies unterscheiden:

Systembestimmte Zugriffspolitik Beim Mandatory Access Control (MAC) wird eine systemglobale Einstellung der Sicherheitsrichtlinien vorgenommen [Fack 01] . Hierbei sind die Regeln zur Zugriffskontrolle fest vorgegeben. Beispiele für diese Art von Sicherheitspolicies wären das Multilevel Security Konzept oder das Konzept eines Ringschutzes. Hierbei wird jede Entität einem bestimmten Sicherheitslevel mit dementsprechenden Rechten zugeordnet.

Benutzerbestimmbare Zugriffspolitik Beim Discretionary Access Control (DAC) können Zugriffe durch die Benutzer beliebig autorisiert werden [Fack 01]. Dabei müssen entsprechende Administrationsrechte vorhanden sein (z.B. Benutzer muss Eigentümer des Objektes sein). Desweiteren besteht hier auch die Möglichkeit der Weitergabe von Rechten an Dritte. Diese Art der Zugriffspolitik sollte aber insgesamt nur eine Ergänzung der systembestimmten Policy darstellen, um diese feingranular erweitern zu können.

Das Java 2 Sicherheitskonzept benutzt eine globale Sicherheitspolitik zur Definition der Rechte, welche innerhalb einer Java-Laufzeitumgebung gelten sollen. Positiv zu bewerten ist die Tatsache, daß bei der Java 2 Security Architecture [Sun00] die Mechanismen zur Durchsetzung der Rechte (Beschreibung des SecurityManager/AccessController siehe Kapitel 2.1.3) unabhängig von der Definition selbiger implementiert worden sind[Fack 01]. Die Sicherheitpolicy des MASA Systems wurde um eine benutzerbestimmbare Zugriffspolitikskomponente erweitert[1], um eine flexiblere und einfachere Handhabung der Rechte einzelner Agenten zu erzielen.

1.2 Die MASA Umgebung

Die Mobile Agent System Architecture ist eine vollständg in Java implementierte Laufzeitumgebung für mobile Agenten nach dem MASIF Standard [GHR 99]. Als Middleware kommt dabei Corba (Orbacus ORB von OOC) zum Einsatz, um eine ortstransparente Kommunikation zwischen Objekten zu ermöglichen. Die Interaktion mit dem System und den beteiligten Komponenten erfolgt dabei durch Java Applets, die von einem in das System integrierten Webserver geladen werden können. Für eine detaillierte Beschreibung des Systems und der verwendeten Konzepte sei auf [Roel 99], [Kemp 98], [GHR 99] verwiesen.

[1] siehe Konzept der Policy Wishlists Kapitel 3.2

1.3 Aufgabenstellung

Für die Mobile Agent System Architecture (MASA) sollte ein Konzept zur automatischen Verarbeitung von Policy Wishlists entwickelt und implementiert werden. Die Implementierung sollte hierbei den Missbrauch des Konzepts der Policy Wishlists (siehe Kapitel 3.1) unterbinden. Hierzu sollten geeignete graphische Benutzerschnittstellen entworfen und implementiert werden. Zu beachten waren auch Anforderungen an die zu entwickelnden Komponenten um eine nahtlose Integration in das MASA System zu gewährleisten.
Desweiteren waren auch Komponenten zum Managment von MASA-Sicherheitspolicies zu entwickeln um eine zentrale und benutzerfreundliche Administration der Sicherheitskonfiguration zu erreichen.

1.4 Aufbau der Arbeit

Innerhalb der Ausarbeitung zum Systementwicklungsprojekt wird anfangs auf Java Sicherheitsarchitektur und deren einzelnen Komponenten eingegangen. Anschließend folgt eine Beschreibung der Masa Sicherheitsarchitektur und des entworfenen Konzepts zur automatischen Verarbeitung der Policy Wishlists. Abschließend werden dann der Entwurf, das Design, die Graphische Benutzerschnittstelle und die Funktionalität des entwickelten Agenten beschrieben.

2 Die Java 1.2 Sicherheitsarchitektur

In diesem Kapitel wird nun der Ausschnitt der Java 1.2 Sicherheitsarchitektur bezüglich der Rechtevergabe und deren Durchsetzung näher erläutert. Dieses Sicherheitskonzept ermöglicht eine differenzierte Zuordnung von Rechten auf Klassenbasis. Hierbei werden Klassen einem 'Schutzbereich' (Protection Domain) zugeordnet, für den jeweils eigene Rechte definiert werden können. Diese Abbildung kann mittels einer 'CodeSource', dem Herkunftsort der Klasse, und möglicher digitaler Signaturen erfolgen. Die Rechte, welche für diesen Bereich gelten, werden mittels einer Menge von einzelnen Permission Objekten repräsentiert welche die kleinste Rechte-Einheit im Java Sicherheitskonzept darstellen. Diese wiederum können als Mengen von homogenen (Permission Collections) und heterogenen Rechten (Permissions) aggregiert werden.
Die Definition der Rechte einer ProtectionDomain erfolgt über die Policy Klasse, die als Schnittstelle zu einer textuellen Policy fungiert. Dadurch wird eine Rechteanpassung ohne Modifikation des Sourcecode möglich. Die Überprüfung der Rechte erfolgt dann durch den SecurityManager/AccessController.[2]

2.1 Komponenten der Sicherheitsarchitektur

Im folgenden Abschnitt wird nun näher auf die zur Sicherheitsüberprüfung wichtigen Klassen eingegangen. Dabei werden zuerst die für die Darstellung und Verwaltung von Rechten nötigen Klassen erläutert, gefolgt von den Klassen, die zur Zuordnung von Rechten zu Code benötigt werden. Abschließend werden die Klassen für die Überprüfung und Durchsetzung der gewährten Rechte erläutert.

2.1.1 Repräsentation von Rechten

Die Klasse `java.security.Permission` und deren Derivate

Die Permission Klassen repräsentieren den Zugriff auf eine System Ressource. Die abstrakte Klasse `java.security.Permission` stellt hierbei die Basis für die entsprechenden spezifischen Permission Objekte dar. Durch eine geeignete Ableitung der Permission Klasse kann man für die verschiedenen Arten von Ressourcen (Dateien, Netzwerk, ...) eigene Rechte definieren. Hierbei spielt vor allem die Methode `implies()` eine gewichtige Rolle, um transitive Beziehungen und Implikationen effizient handzuhaben.

Die Klasse `java.security.PermissionCollection`

Diese Klasse repräsentiert eine homogene Ansammlung von Permission Objekten. Jede Instanz dieser Klasse sollte nur Instanzen von Permission Objekten des gleichen Typs enthalten.

Die Klasse `java.security.Permissions`

Diese Klasse dient zur Darstellung einer Ansammlung von heterogenen Permission Objekten. Normalerweise besteht sie hierbei aus einer Menge von java.security.PermissionCollection Objekten.

2.1.2 Zuordnung von Rechten

Diese Klassen dienen zur Zuordnung von Rechten auf die jeweiligen Klassen.

Die Klasse `java.security.CodeSource`

Ein CodeSource Objekt repräsentiert dabei die Herkunft von Klassen. Zur Unterscheidung zwischen verschiedenen CodeSources wird dabei eine URL verwendet, welche dem Ort entspricht, von dem die Klasse geladen wurde. Optional kann noch eine Menge von Zertifikaten, mit denen die Klassen digital signiert wurden, angegeben werden.

[2] Aus Kompatibilitätsgründen erfolgt hier eine Delegation der Überprüfung vom SecurityManager auf den Accesscontroller (seit JDK1.2)

Die Klasse `java.security.ProtectionDomain`

Eine ProtectionDomain kapselt eine gegebene CodeSource-Instanz und die dazu gehörigen Permissions. Eine solche Instanz repräsentiert somit den Rechtebereich aller Klassen aus gleicher Quelle und einer Menge von Zertifikaten, mit dem der Code dieser Klassen digital signiert wurde. Wobei Klassen, welche die gleichen Rechte besitzen, aber von verschiedenen Herkunftsorten stammen (Differenzierung erfolgt über den ClassLoader), verschiedenen ProtectionDomains angehören. Außerdem kann eine Klasse nur genau einer einzigen ProtectionDomain angehören. Das Setzen der ProtectionDomain geschieht bei der `defineClass()` Methode im ClassLoader. Dadurch ist es aber leider nicht möglich die Rechte einer Klasse dynamisch zu verändern, da die jeweiligen ProtectionDomains intern innerhalb des SecureClassLoaders gecached werden. Nur bei einer Neudefinition von Klassen kann hier eine Modifikation der Rechte einer Klasse erreicht werden. Dieses Problem besteht ab dem JDK1.4 nicht mehr, da dort eine dynamische Rechtevergabe vorgesehen ist.

Die Klasse `java.security.Policy`

Die Sicherheits Policy einer Java Laufzeitumgebung wird durch die Klasse Policy repräsentiert. Sie spezifiziert und liefert die Permissions zurück, welche für eine bestimmte CodeSource definiert worden sind. Obwohl mehrere Instanzen der Policy Klasse möglich sind, kann jedoch zur selben Zeit nur eine einzige aktiv sein. Die Quelle der Policyinformationen, welche durch das Policy Objekt repräsentiert wird, kann implementationsspezifisch erfolgen. In der Standardimplementierung des JDK 1.2 wird hierbei ein in einer speziellen Syntax abgefasstes ASCII Textfile eingelesen, das die Abbildung der CodeSources auf Permissions vornimmt.

```
Eine formale Beschreibung in BNF:

PolicyFile      -> PolicyEntry | PolicyEntry; PolicyFile
PolicyEntry     -> grant {PermissionEntry}; |
                   grant SignerEntry {PermissionEntry} |
                   grant CodebaseEntry {PermissionEntry} |
                   grant SignerEntry, CodebaseEntry {PermissionEntry} |
                   grant CodebaseEntry, SignerEntry {PermissionEntry} |
                   keystore "url"
SignerEntry -> signedby (a comma-separated list of names, which must have signed the code)
CodebaseEntry -> codebase (a string representation of a URL)
PermissionEntry -> OnePermission | OnePermission PermissionEntry
OnePermission -> permission permission_class_name
                 [ "target_name" ] [, "action_list"]
                 [, SignerEntry];
```

An folgendem konkretem Beispiel aus dem MASA Policy File wird nun allen Agenten eine Socket-Permission für beliebige Hosts und Ports ¿ 1024 eingeräumt:

```
grant codeBase "systemresource://de/unimuenchen/informatik/mnm/masa/agent/-"
{
        permission java.net.SocketPermission "*:1024-", "listen, accept, resolve, connect";
}
```

2.1.3 Kontrolle der Rechte

Die Klasse `java.security.AccessController`

Die Kontrolle der Rechte erfolgt ab dem JDK 1.2 über den AccessController. Um abwärtskompatibel zu bleiben wird dieser in Verbindung mit dem SecurityManager eingesetzt, welcher die Kontrolle an den AccessController delegiert. Hierbei handelt es sich um eine Klasse mit statischen Methoden, die anhand des Aufrufkontextes Überprüfungen auf bestimmte Rechte durchführen und gegebenfalls entsprechende Exceptions auswerfen. Dabei gilt grundsätzlich, daß die Klasse mit dem Zugang zu einer schützenden Ressource aktiv auf die Einhaltung der Rechte prüfen muß. Da aber möglicherweise Code verschiedener Herkunft und damit unterschiedlichen ProtectionDomains aktiv sein kein, muß dies bei der Überprüfung der Ressourcen beachtet werden. Um einen Missbrauch zu verhindern, muss also eine Rechteschnittmenge aller Klassen, welche an dem Aufruf beteiligt sind gebildet werden. Insgesamt gesehen, hat damit der AccessController folgende 3 Aufgaben:

- Treffen einer Entscheidung, ob der Zugriff einer kritischen Systemresource erlaubt oder verweigert werden soll
- Privilegierte Codeabschnitte kennzeichnen, um die Ausführung spezieller CodeTeile trotz Sicherheitsrestriktionen zu erlauben
- Informationen über den Aufrufkontext zu liefern um zwischen verschiedenen Kontexten differenzierte Entscheidungen treffen zu können

Durch die beiden letzten Punkte besteht darüberhinaus die Möglichkeit Code als privilegiert zu kennzeichnen, um die Überprüfung des Aufrufstacks auf eine bestimmte Tiefe zu limitieren. Dadurch ist es dann möglich das der priviligierte Code ungeachtet der anderen Klassen auf dem Aufrufstack sicherheitskritische Operationen durchführen kann.
Am folgenden Beispiel (siehe Abbildung 1) soll nun eine konkrete Überprufung (vereinfachte Darstellung) von Rechten dargestellt werden: Hierbei könnte z.B. in der Klasse com.xyz.util eine

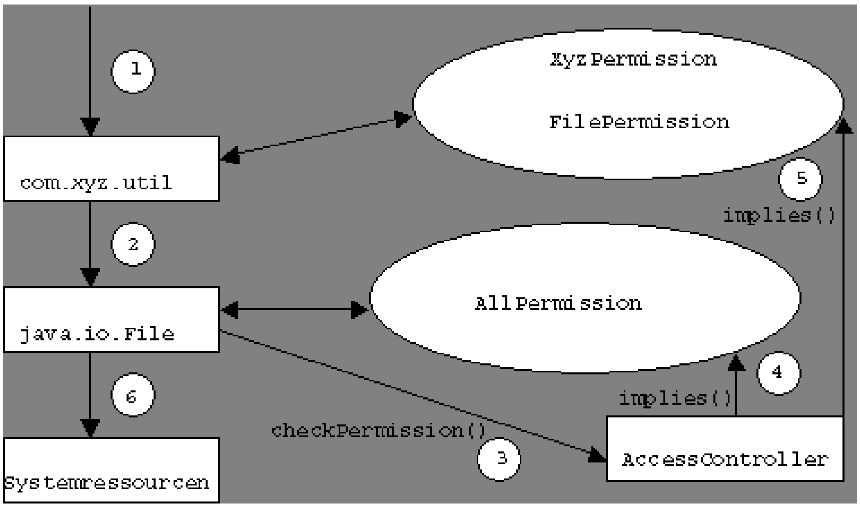

Abbildung 1: Beispiel einer Überprüfung

Methode `deleteFile('filexy')` aufgerufen werden (1)
In dieser Methode könnte beispielsweise versucht werden das angegebene File zu löschen:

```
public void deleteFile(String fileName) {
  ....
  try {
    (new File(fileName)).delete();  \\ (2)
  }
  catch (Exception ex) {
  ..
  }
}
```

In der Systemklasse java.io.File erfolgt nun die Überprüfung der Rechte mittels des AccessControllers:

```
perm = new FilePermission("filexy",delete);
AccessController.checkPermission(perm); \\ (3)
```

Daraufhin überprüft der AccessController alle Ebenen des Aufrufstacks, indem er sich über den ClassLoader der Klassen die zugehörigen ProtectionDomains besorgt. und auf diesen die Methode implies(perm) aufruft (4) (5).
Können nun alle Klassen die bnötigten Rechte aufweisen, kehrt der Aufruf einfach zurück und der Zugriff auf die Systemressource wurde gewährt (6). Anderfalls würde durch den AccessController eine SecurityException ausgeworfen werden und damit der Zugriff verweigert.

Der Java ClassLoader

Die Aufgabe eines ClassLoaders besteht im allgemeinen im Laden des Bytecodes und dem Bereitstellen von Klassendefinitionen, welche ungefähr dem Aufgabenbereich eines Linkers entspricht. Desweiteren ist er im Bezug zur Sicherheitsarchitektur für folgende Aufgaben verantwortlich [Oaks 98] :

Separation der Namensbereiche (Abbildung von Klassen auf Namespaces)

Zusammenarbeit mit dem SecurityManager respektive dem AccessController

Separation von lokalen und entfernten Klassen bezüglich der Sicherheitsrestriktionen

Im Sicherheitskonzept von Java spielt deshalb der ClassLoader eine eminent wichtige Rolle. Durch die enge Zusammenarbeit mit dem SecurityManager und dem AccessController können diese auf Informationen wie z.B. den Ort und den Signaturen der Klasse zugreifen und differenzierte Entscheidungen bezüglich der Rechte treffen.

2.1.4 Zusammenfassung

Mit der Java 2 Sicherheitsarchitektur wurden Möglichkeiten zur Durchsetzung einer differenzierten und feingranularen Sicherheitspolitik geschaffen. Durch die Verwendung einer einzigen systemglobalen Sicherheitspolitik steht aber auch nur der Schutz der Ausführungsumgebung im Vordergrund. Ein großer Mangel der Sicherheitsarchitektur besteht dabei in der fehlenden Unterstützung von sich dynamisch ändernden Rechten. Somit können anwendungsspezifische Sicherheitspolitiken, welche eine feingranulare und dynamische Adressierung der Subjekte, Objekte und der Zugriffsrechte erfordern, nicht definiert werden. Abhilfe besteht in den neueren Java Sicherheitsarchitekturen des JDK1.3/1.4 welche neuere Konzepte, wie rollenbasierte und dynamische Rechtevergabe, unterstützen. Für nähere Informationen bezüglich der Java Sicherheitsarchitektur sei auf die Online Dokumentation von SUN Microsystems verwiesen [Sun00].

3 Sicherheitspolicies und deren Durchsetzung innerhalb von MASA

Im folgenden Kapitel wird nun die Definition und die Durchsetzung einer Sicherheitspolicy näher beschrieben.

3.1 Überblick über die MASA Sicherheitspolicy Architektur

Das Sicherheitskonzept zur Definition und Durchsetzung von Rechten innerhalb der MASA Laufzeitumgebung wird nun im folgenden auszugsweise näher erläutert. Die Basis für die Vergabe und Durchsetzung von Rechten besteht in der zweifelsfreien Authentifikation der beteiligten Subjekte. Die Identifikation der Agentensysteme und Agenten wird durch eine eindeutige Namensvergabe sichergestellt. Die Authentifikation erfolgt dabei mittels digitaler Signaturen und Zertifikate.

Die Sicherung der Endsystemschnittstelle erfolgt bei MASA durch die Java 2 Platform Security Architecture [Oaks 98]. Hierbei werden die Konzepte der Definition von Rechten mittels eines Policy Files und der Policy Wishlist, welche die gewünschten Rechte einer Agentengattung spezifiziert, eingesetzt. Eine Agenteninstanz ist nun die konkrete Instanziierung des durch den Agentengattungs Code definierten Agenten. Bei der Erzeugung einer Agenteninstanz wird der Java Code und weitere spezifizierende Dateien, welche innerhalb eines signierten JAR Files zusammengefasst sind, vom AgentCodeRepository angefordert und der Agent dementsprechend erzeugt. Hierbei wird im auch ein eindeutiger Name zugewiesen, um zwischen multiplen Instanzen einer Agentengattung entsprechend differenzieren zu können.
Bei der Erzeugung einer Agenteninstanz auf einem Agentensystem wird durch den AgentClassLoader die sogenannte ProtectionDomain der Agenten Klassen definiert, welche die Rechte desselbigen spezifiziert.
Die geltenden Rechte einer Agenten Instanz setzen sich somit aus relevanten Teilen der System Policy und den Einträgen der Policy Wishlist der Agentengattung zusammen. Die Definition der Rechte geschieht mittels der Standard Java 2 Policy und einer Liste von einzelnen Permissions (Wishlist). Für eine detaillierte Beschreibung des Sicherheitskonzepts sei auf [Roel 99] verwiesen.

3.2 Konzept der Filterung von Agenten Policy Wishlists

Um eine automatische Behandlung von Rechten einzelner Agentengattungen zu ermöglichen wurde eine neue Komponente in die bestehende Sicherheitsarchitektur integriert. Hierbei werden die Policy Wishlists der Agenten mittels Substitutionen modifiziert, bevor die enthaltenen Rechte zur geltenden Policy hinzugefügt werden.
Konkret wurde dies mittels einer Filter Komponente erreicht, die innerhalb des AgentClassLoaders eingefügt wurde. Diese wählt beim Laden der Policy Wishlist aus dem CodeRepository des jeweiligen Agenten einen entsprechenden Satz von Substitutionsregeln aus einem FilterRepository aus und wendet diese Regeln auf die Liste der gewünschten Permissions des Agentens an. Bei einer geeigneten Wahl von Substitionsregeln kann somit eine differenzierte automatische Behandlung von Wishlists erfolgen. Die Auswahl der FilterSets basiert dabei auf der CodeSource der jeweiligen Agenten Gattung. Da diese für jede Agentengattung unterschiedlich sein muss, lässt sich somit eine eindeutige Abbildung auf die entsprechenden FilterSets realisieren.
Da die Agenteninstanzen, wie eingangs erwähnt, eindeutige Namen besitzen ließe sich dieses Konzept auch auf die einzelnen Objekte übertragen. Hierfür müsste jedoch eine Änderung der Java Sicherheitsarchitektur von Klassen auf Objekt basierte Mechanismen erfolgen, was meiner Kenntnis nach erst für zukünftige Versionen des JDK vorgesehen ist [3] Desweiteren lässt sich auch ein sog. Default Filter definieren, welcher im Falle eines nicht vorhandenen Filters greift [4].

[3] JDK 1.4 wird keine Objektbasierten Sicherheitsmechanismen unterstützen
[4] Hier könnte bespielsweise ein Filter definiert sein, der die restriktivste Policy garantiert

3.3 Analyse des Konzepts und der Implementierung

Die Implementierung zur automatischen Modifikation von Agenten Policy Wishlists, weist in seinem Konzept einige Vor- und Nachteile auf. Aus diesem Grunde, sollen die Stärken und Schwächen hier aufgezeigt und verdeutlicht werden.

Die Stärken des Konzepts liegen vor allem in seiner Mächtigkeit,Flexibilität und Skalierbarkeit. Durch die Möglichkeiten der Definition von Ersetzungsfolgen mittels regulärer Ausdrücke [5] lassen sich komplexeste Modifikationen der Agenten Policy Wishlists realisieren. Durch die äußerst einfache Struktur der Policy Wishlists ist die Definition solcher Ausdrücke relativ leicht zu bewerkstelligen.

Der Aufbau der Wishlists besteht aus einer Enumeration von einzelnen Permissions mit folgender Syntax:

```
permission_class_name ::= FilePermission|SocketPermission|XYZPermission|...
permissionentry ::= permission permission_class_name [ "target_name" ]
                    [, "action_list" ][, SignerEntry];
wishlist ::= permissionentry*
```

So sind neben einfachen Filterregeln für bestimmte Arten von Permissions auch Modifikationen von Parameterlisten möglich.
Somit können Veränderungen der Sicherheitspolicy von Agenten schnell und flexibel umgesetzt werden. Diese Vorgehensweise bietet somit auch dementsprechend Vorteile bei der Migration und einen möglichen Wechsel der Policy Sprache oder bei der Einführung von benutzerspezifischen Permissions. Hierbei kann durch eine Neudefinition der Filter- bzw Modifikationsregeln eine Änderung innerhalb des Java Codes vermieden werden. Ungeachtet der großen Vorteile besitzt dieses Konzept jedoch gravierende Mängel.
Am schwersten wiegt die Dezentralisierung und Desintegration der geltenden Policy. Durch die ohnenhin schon zweigeteilte Policy[6] kommt nun eine weitere Komponente hinzu.
Desweiteren ist die Definition solcher Filtereinträge komplex und fehleranfällig, da auch abstrakte Regeln für unbekannte Policies definiert werden müssen. Trotz aller Vorteile, wiegen die Dezentralisierung und Desintegration der Policies schwer und reduzieren die Praxistauglichkeit dieser Lösung.
Um die Mängel dieser Lösung zu reduzieren, musste bei der Entwicklung des PolicyEditor Agenten und dessen Oberfläche hoher Wert auf die Integration der Policy Einstellungen gelegt werden.
Dazu wurden die gesammten Sicherseinstellungen, welche für einen Agenten gelten, innerhalb einer Komponente zusammengefasst. Dies führt zu einer übersichtlichen und einfachen Möglichkeit die Sicherheitspolicy eines Agenten zu bearbeiten.

[5]zur Beschreibung dieser Ausdrücke wird die Perl 5 Syntax verwendet

[6]die resultierende Policy eines Agenten setzt sich aus der Summe von SystemPolicy und AgentPolicy Einträgen zusammen

4 Der MasaPolicyEditor Agent und seine Komponenten

Der Aufgabenbereich des MasaPolicyEditor Agenten liegt in der Manipulation und dem Management von Sicherheits Policies. Hierbei fungiert der Agent als Schnittstelle zum Laufzeitsystem um entsprechende Aktionen anzustossen und persistent durchzusetzen.
Bei der Entwicklung des Agenten wurde besonders Wert auf eine hohe Funktionaliät und eine geeignete Benutzerschnittstelle gelegt. Die Konzeption der graphische Benutzerschnittstellen hatte eine möglichst hohe Integration von Komponenten, welche an der Defnition und Durchsetzung von Sicherheitspolicies beteiligt sind, als Ziel. Zudem sollten Änderungen bzgl. der Sicherheitsarchitektur[7] ohne große Modifikationen im Quellcode einhergehen.

4.1 Funktionalitätsbeschreibung des Agenten

Zur Erläuterung der Funktionalität des Agenten folgt eine kurze Beschreibung der Corba Schnittstellen.

```
IDL-Spezifikation des MasaPolicyEditor Agenten:

module PolicyEditor {
        typedef sequence<octet>           OctetString;
        typedef sequence<OctetString>     OctetStrings;
        typedef string                    ASName;
        typedef sequence<ASName>          ASNameList;
        exception AgentFileNotFound {};
        exception PolicyUnavailable {};
  interface PolicyEditor : agent::Migration
  {
    agentSystem::AgentKindList getAvailAgentKinds();
    ASNameList listPlaces();
    CfMAF::Name getCurrentSystem();
    void migrateToSystem(in short index);
    OctetString getAgentKindPolicyWlist(in agentSystem::AgentKind agentKind);
    OctetString getAgentKindInitialProps(in agentSystem::AgentKind agentKind);
    OctetString getAgentSystemPolicy();

    agentSystem::FilterList getAgentFilterSetConfig(
                            in agentSystem::AgentKind agentKind);

    void writeAgentFilterSetConfig(
                            in agentSystem::AgentKind agentKind,
                            in agentSystem::SubstitutionSet content);

    OctetString getSysPolicy (in CfMAF::Name asystem) raises (PolicyUnavailable);

    void writeSysPolicy(in OctetString content);
    void refreshSystemPolicy();
    void gatherPolicies(in CfMAF::NameList asystems);

    void distributePolicies(in CfMAF::NameList asystems,
                            in OctetString mpolicy);

    void filterPolicies(    in CfMAF::NameList asystems,
```

[7] Beispielsweise durch Einführung anwendungsspezifischer Permissions oder Migration auf JDK 1.3

```
                              in agentSystem::SubstitutionSet filters);

    boolean sysPolicyAvailable (in CfMAF::Name asystem);
  };
};
```

Zusätzlich zu der Funktion als Schnittstelle zum System und zur Benutzerschnittstelle, erfüllt der Agent vor allem drei Aufgaben:

- Sammeln von Agentensystem Policies
  Um System Policies von entfernten Systemen anzuzeigen und editieren zu können, müssen sie vorher durch den Agenten gesammelt und intern gespeichert werden. Nachdem der Agent die relevanten Systeme aufgesucht und deren Policies abgerufen hat, kehrt der Agent zum Ausgangsystem zurück.

- Verteilen von Agentensystem Policies
  Hier kann eine eine eigens definierte (beispielsweise die lokal installierte) Policy auf den spezifizierten Systemen verteilt und in Kraft gesetzt werden. Nachdem der Agent die Systeme sequentiell abarbeitet und dort die entsprechende Policy abspeichert und danach ein Policy.refresh() aufruft, kehrt er wieder zum ursprünglichen System zurück.

- Anwenden von Substitutionen mittels regulärer Ausdrücke
  Hier kann mittels einer speziellen Obfläche eine Menge von Substitutions-Regeln aufgestellt werden und auf den ausgewählten Systemen zur Anwendung kommen.

Die Semantik der einzelnen IDL Operationen soll nun näher beschrieben werden:

- `agentSystem::AgentKindList getAvailAgentKinds();`
  Gibt die Menge aller auf dem System installierten (aller auf dem System direkt startbaren) Agentengattungen zurück.

- `ASNameList listPlaces();`
  Gibt die Menge aller erreichbarer Agentensysteme zurück. Diese Funktion dient zur Anzeige aller verfügbaren Agentensysteme, welche durch den Agenten besucht werden können.

- `CfMAF::Name getCurrentSystem();`
  Hilfsfunktion zur Anzeige des aktuellen Agentensystems.

- `void migrateToSystem(in short index);`
  Funktion zur Migration des Agenten auf ein neues Agentensystem.

- `OctetString getAgentKindPolicyWlist(in agentSystem::AgentKind agentKind);`
  Liefert die PolicyWishlist einer Agentengattung.

- `OctetString getAgentKindInitialProps( in agentSystem::AgentKind agentKind);`
  Liefert die Properties einer Agentengattung.

- `OctetString getAgentSystemPolicy();`
  Liefert die System Policy des aktuellen Laufzeitsystems.

- `agentSystem::FilterList getAgentFilterSetConfig(in agentSystem::AgentKind agentKind);`
  Liefert die aktuelle Filter Konfiguration einer Agentengattung. Falls kein spezifischer Filter für eine Agentengattung definiert worden ist, wird hier der DefaultFilter zurückgegeben. Die Datenstruktur FilterList beinhaltet neben den Substitutionsregeln auch eine Angabe über den Typ des Filters. Durch den Aufruf des PermissionManagers wird diese Methode aufgerufen und die aktuelle Policy Konfiguration des Agenten damit angezeigt.

- `void writeAgentFilterSetConfig(in agentSystem::AgentKind agentKind,in agentSystem::SubstitutionSet content);`
Speichert eine Filter Konfiguration persistent auf dem aktuellen Laufzeitsystem ab. Dadurch können die durch den PermissionManager spezifizierten Filtersätze auf dem System gesichert werden.

- `OctetString getSysPolicy (in CfMAF::Name asystem) raises (PolicyUnavailable);`
Liefert die angegebene System Policy zurück. Falls die Policy nicht im Agenten abgespeichert ist (Die System Policies müssen in einem vorhergehenden Schritt gesammelt worden sein) wird eine PolicyUnavailable Exception ausgeworfen. Nach einer Gather Policy Operation des Policy Managers kann dann die System Policy eines entfernten Systems angezeigt und modifiziert werden.

- `void writeSysPolicy(in OctetString content);`
Speichert die übergebene Policy persistent im aktuellen Laufzeitsystem ab.

- `void refreshSystemPolicy();`
Aktualisiert die Policy des aktuellen Agentensystems. Dies führt zu einem Aufruf Policy.refresh(), welcher in der aktuellen Version ein Wiedereinlesen der SystemPolicy zur Folge hat. Diese Funktion wird sowohl nach dem Verteilen einer globalen Policy auf den angegeben Systemen als auch direkt durch den Policy Selektor aufgerufen.

- `void gatherPolicies(in CfMAF::NameList asystems);`
Agentenfunktion zum Sammeln von System Policies, welche in einer NameList spezifiziert werden. Dieser Schritt ist notwendig falls beispielsweise durch den Policy Manager System Policies von entfernten Systemen modifziert werden sollen.

- `void distributePolicies(in CfMAF::NameList asystems,in OctetString mpolicy);`
Verteilt die übergebene System Policy auf die in der NameList spezifizierten Agentensysteme. Angestossen durch den Policy Manager kann dadurch eine Policy rasch auf allen verfügbaren Systemen verteilt und in Kraft gesetzt werden.

- `void filterPolicies(in CfMAF::NameList asystems,in agentSystem::SubstitutionSet filters);`
Wendet die übergebenen Substitutionen auf die in der NameList spezifizierten Agentensysteme an. Dadurch können gleichartige Modifikationen an mehrerern System Policies mittels regulärer Substitutionen innerhalb einer Operation (Die direkte Manipulation, mittels des Masa Policy Editors, würde ein vorheriges Einsammeln der relevanten Policies nötig machen) durchgeführt werden.

- `boolean sysPolicyAvailable (in CfMAF::Name asystem);`
Funktion zur Überprüfung, ob eine bestimmte Agentensystem Policy in dem Agenten abgespeichert wurde. Dadurch werden innerhalb der Policy Manager Komponente alle direkt verfügbaren System Policies angezeigt.

4.2 Funktionalitätsbeschreibung der Benutzerschnittstelle

Auf die Gestaltung der Benutzeroberflächen wurde bei der Entwicklung großen Wert gelegt, um den Ansprüchen einer Managmentanwendung gerecht zu werden. So wurde versucht zusammengehörige Teile innerhalb einer Komponente zu integrieren um dem Benutzer den jeweils bestmöglichsten Überblick zu verschaffen. In den folgenden Bereichen sollen die einzelnen Benutzerobflächen nun näher beschrieben werden.

Die Oberfläche des Applets wurde dabei in zwei Bereiche aufgeteilt, um den einzelnen Aufgabenbereichen des Agenten Rechnung zu tragen.

4.2.1 Policy Selector Komponente

Diese Oberfläche und deren Subkomponenten sind für das Anzeigen und Modifizieren der Sicherheitspolicies von einzelnen Agentengattungen und der lokalen[8] System Policy verantwortlich. Die Oberfläche des Policy Selector (Abbildung 2) gliedert sich dabei in drei Bereiche. Im oberen Bereich läßt sich der Agent auf ein erreichbares System migrieren. Der zentrale Anzeigenbereich zeigt alle auf dem System installierten Agentengattungen. Auf der rechten Seite lassen sich die Komponenten zum Anzeigen/Bearbeiten der Agentengattungs- bzw. System Policy Konfiguration aufrufen. Da die Oberfläche leider nicht gänzlich intuitiv bedient werden kann, sollen hier die

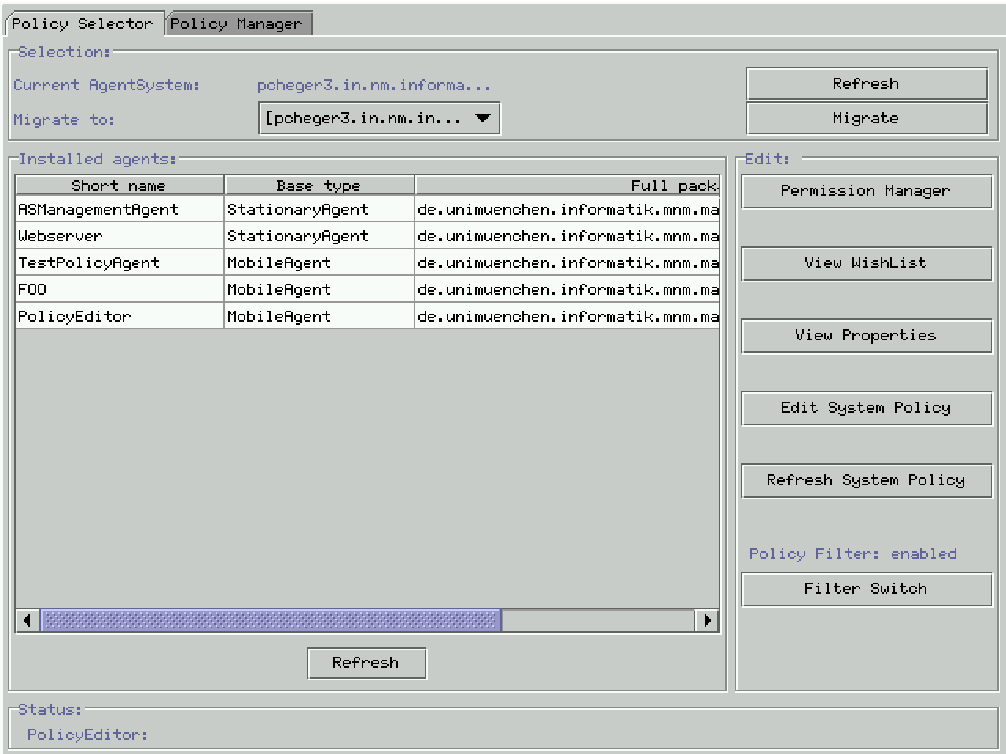

Abbildung 2: Policy Selektor

einzelnen Funktionalitäten kurz beschrieben werden.

[8]lokal im Sinne des Aufenthaltsortes des Agenten

4.2.2 Permission Manager

Eine zentrale Komponente für das Rechtemanagment von Agentengattungen bildet der Permission Manager (Abbildung 3). In seiner Oberfläche wird die komplette resultierende Policy (inklusive der globalen Permissions) eines Agenten dargestellt. Dabei werden alle beteiligten Komponenten innerhalb eines Frames angezeigt.
Dies erfolgt durch eine konzeptionelle Teilung des Frames in drei Bereiche:
Im oberen Bereich erfolgt die Eingabe, bzw Übernahme der CodeBase der Agentengattung. Diese bestimmt die Anzeige der Permissions, indem die System Policy nach relevanten Einträgen durchsucht und angezeigt wird. Ebenso wird das lokale CodeRepository nach equivalenten Einträgen durchsucht und die entsprechende Policy Wishlist angezeigt.
Die Anzeige erfolgt dabei durch drei TextAreas, welche den linken Bereich der Komponente bilden und jeweils die System Permissions, die gefilterte (siehe Jar Permissions) und ungefilterte (siehe Wishlist Permissions) Policy Wishlist der Agentengattung darstellen. Durch mehrmaliges Drücken des 'Detail' Buttons können verschiedene Detail-Informationen ein und wieder ausgeblendet werden. Desweiteren besteht hier die Möglichkeit die lokale System Policy zu bearbeiten um beispielsweise dem Agenten neue Rechte zu gewähren.
Der rechte Bereich (Abbildung 3) wird mit dem Filter Managment der Agenten Gattung ausgefüllt. Hier wird je nach Konfiguration ein spezifischer (möglicherweise leerer) oder der Default-Filter angezeigt, falls für diese Agenten Gattung kein Filter definiert worden ist. Die Anzeige des Filtertyps erfolgt dabei durch das Label FilterSet (Specific/Default). Dieses kann dann durch die entsprechenden Buttons definiert werden um beim Speichern des Filters den entsprechenden Filter zu Setzen.
Durch den Button 'Add' kann eine weitere Filterregel zum bestehenden Filterset hinzugefügt werden. Die Definition einer neuen Filterregel erfolgt dabei mittels einer weiteren Komponente, in welcher der Name der Regel, ein regulärer Matchingausdruck und ein Substitutionsausdruck eingegeben werden muss. Hier kann der Ausdruck mittels einer Testeingabe in der angrenzenden TextArea auch grob überprüft werden.
Die Filter werden dann in einem speziellen Verzeichnis innerhalb des MASA Systems, spezifiziert durch eine Umgebungsvariable, als XML Dateien abgespeichert. Hierbei wird durch eine Abbildung der CodeBase der Agentengattung auf einen Dateinamen, eine eindeutige Filterzuordnung erreicht.
Der Button 'Filter Line' stellt hierbei eine kleine Hilfe zur Erstellung von Matchingausdrücken dar, um eine Permission mittels eines vorkommenden Wortes zu matchen. Die restliche Benutzerführung sollte intuitiv erfolgen können.

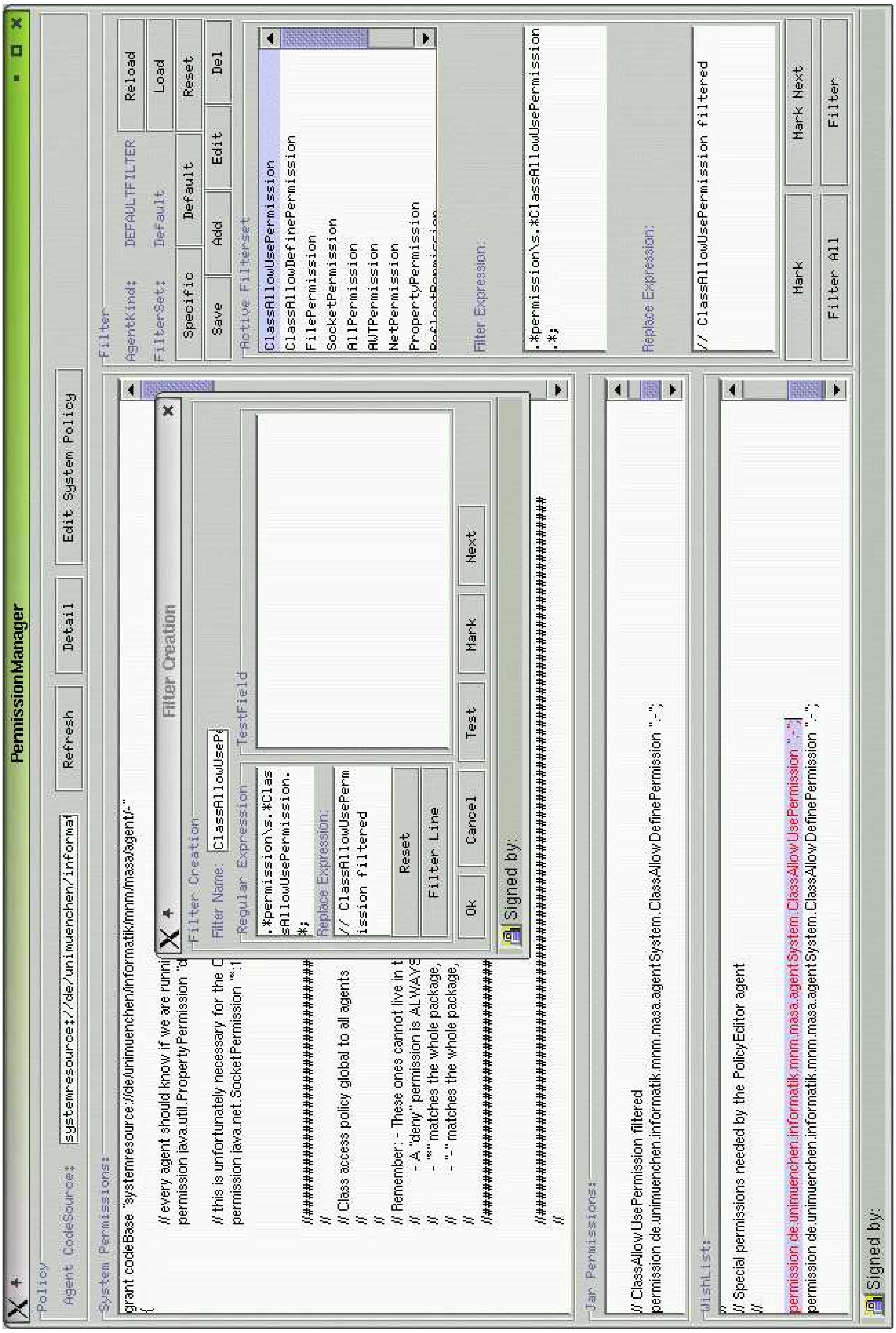

Abbildung 3: Permission Manager

4.2.3 Policy Editor

Um eine komfortable Bearbeitung zu ermöglichen, wurde ein kleiner Editor (Abbildung 4) mit den üblichen Funktionalitäten wie mehrstufiges Undo/Redo, Suchen/Ersetzen mit regulären Ausdrücken und Cut/Copy/Paste entwickelt. Dadurch können die Policy Files innerhalb der Anwendung schnell und komfortabel modifiziert werden.
Das alleinige Speichern der Policy bewirkt jedoch keine Erneuerung der Policy Konfiguration. Diese muss manuell innerhalb der Policy Selector (Abbildung 2) Oberfläche mittels eines 'Refresh System Policy' erneuert werden. Um einen neuen Eintrag zu definieren dient der 'new' Menueintrag. Dieser erzeugt am Ende des System Policy Files einen Rahmen für einen neuen Eintrag mit leerer CodeBase.

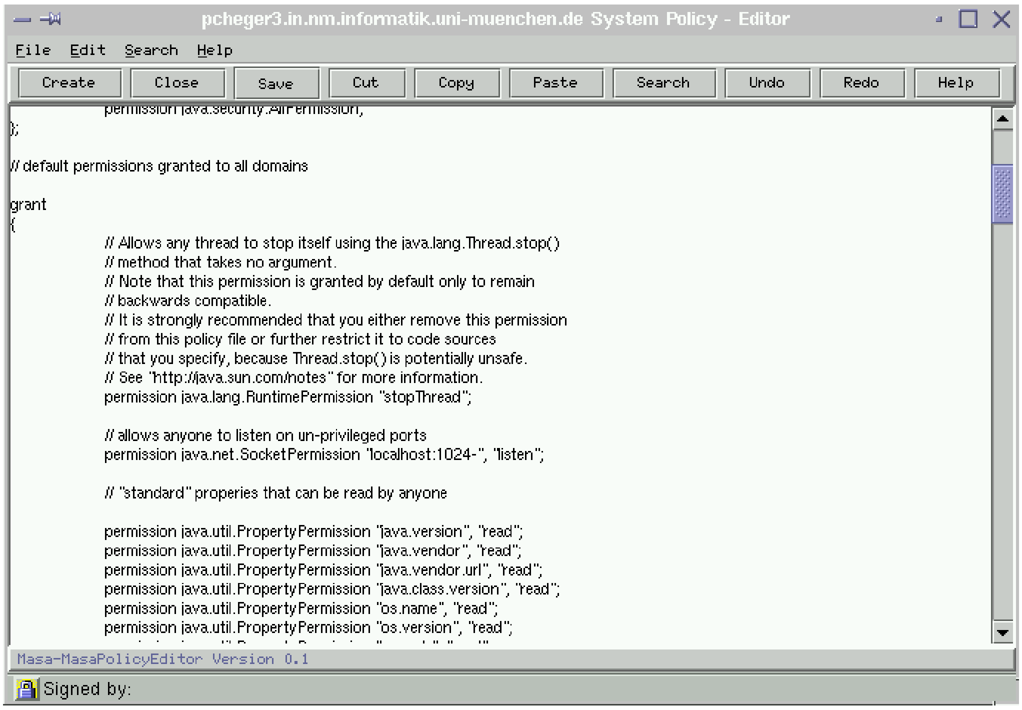

Abbildung 4: Masa Policy Editor

4.2.4 Policy Manager Komponente

Die Policy Manager Komponente (Abbildung 5) bildet die zweite Hauptkomponente des Applets. Hierin sollte das Managment von System Policies erfolgen. Die dreigeteilte Oberfläche besteht aus einem Bereich zur Anzeige aller erreichbaren Agentensysteme, einem Bereich zur Definition einer globalen System Policy und einer Sektion zum Anstossen der einzelnen Aktionen, welche durch den Agenten ausgeführt werden können.
Die Spezifizierung des zu behandelnden Systems und der auszuführenden Aktion erfolgt dabei direkt innerhalb der Anzeige durch das Markieren von Checkboxes. Dabei sind folgende Aktionen möglich:

- GET - Retrieve Policies
  Hier werden die Agentensysteme markiert, welche durch den Agenten besucht werden, um die System Policy des Systems zu erhalten. Da der Agent nur Zugriff auf das lokale Agentensystem besitzt, muss dieser Schritt vor dem Editieren einer speziellen System Policy ausgeführt werden.

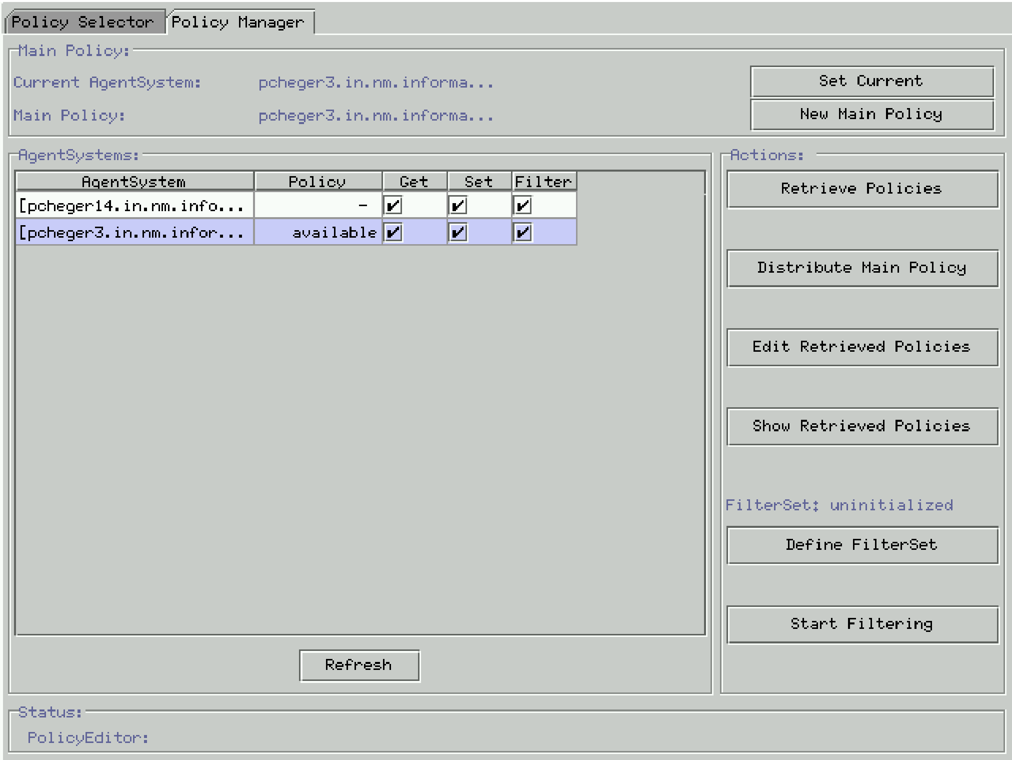

Abbildung 5: Policy Manager

- SET - Distribute Main Policy
  Mit dieser Aktion können die spezifizierten Systeme mit einer neuen System Policy ausgestattet werden. Hierbei wird die vorher spezifizierte 'Main Policy' auf den mit SET markierten System abgespeichert und anschließend ein Policy.refresh() ausgeführt.

- FILTER - Start Filtering
  Mit dieser Aktion können eigens definierte Substitutionen mittels regulärer Ausdrücke auf den sepzifizierten System ausgeführt werden. Bei einer geschickten Wahl der regulären Ausdrücke kann somit eine individuelle Manipulation der jeweiligen System Policies mit einer Aktion erfolgen. Die Definition der regulären Substitutionsausdrücke erfolgt in einer speziellen Komponente, welche im Anschluss näher beschrieben wird.

Durch den Button 'Edit Retrieved Policies' kann eine vorher eingeholte System Policy editiert werden. Durch die Spalte 'Policy (- / Available)' wird der aktuelle Status der eingeholten System Policies angezeigt. Falls nun eine entfernte System Policy im PolicyEditor abgespeichert wird, migriert der Agent auf das entsprechende System und speichert dort die modifizierte Policy ab und führt einen Policy.refresh() auf diesem System aus.

4.2.5 System Filter Komponente

Die System Filter Komponente (Abbildung 6) dient zur Definition von regulären Substitutionen, die auf System Policies ausgeführt werden sollen. Durch die Aktion 'Define FilterSet' wird eine Komponente zum Erstellen und Testen von Substitutionen geöffnet. Die Oberfläche ist dabei in zwei Bereiche aufgeteilt, wobei der linke Teil zum Managment der Ausdrücke dient und der rechte Teil als Testfeld der definierten Substitutionen fungiert. Die Definition der regulären Ausdrücke erfolgt hierbei analog zur Permission Manager (Abbildung 3) Komponente.

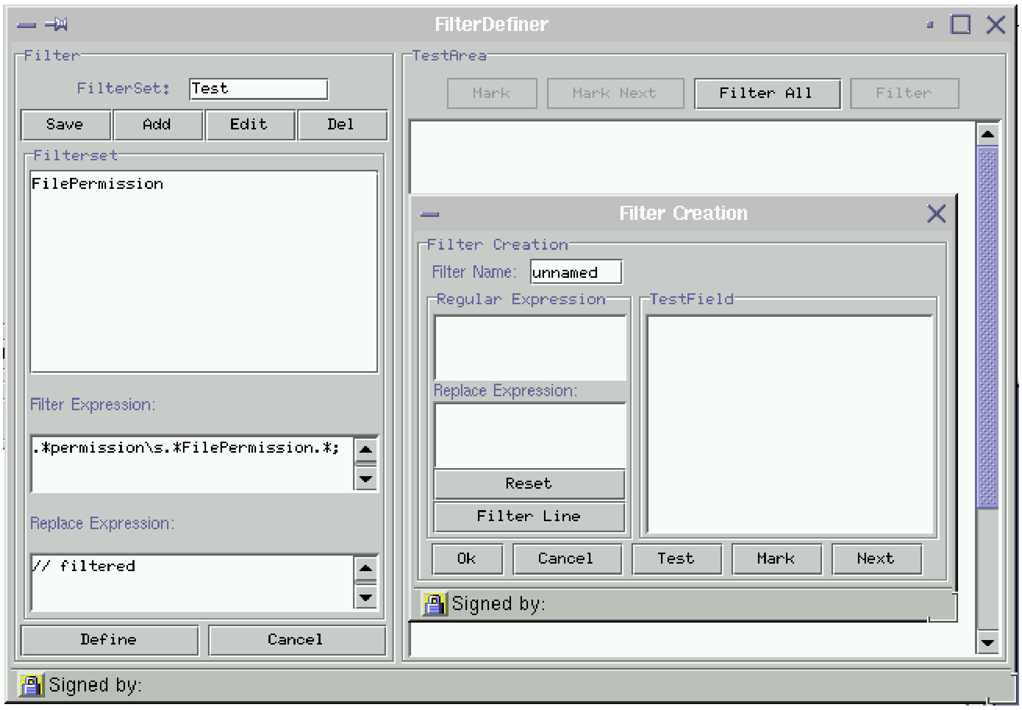

Abbildung 6: Die System Filter Komponente

5 Entwurf und Implementierung

Die Einführung eines Filters mittels Substitutionen durch reguläre Ausdrücke, erlaubt die automatische Verarbeitung der Agenten Policy Wishlists. Bei der Entwicklung der Komponente spielten vor allem eine möglichst geringe Modifikation des bestehenden Systems als auch die Optionalität der Filter-Funktionalität eine große Rolle.

5.1 Der Policy Wishlist Filter

Konkret wurde die Filterkomponente zwischen dem 'AgentClassLoader' und dem 'SimplePolicyParser' eingefügt. Dabei wird dem 'SimplePolicyParser', welcher für die Rückgabe von Permissions aus der geparsten Agenten Policy Wishlist verantwortlich ist, eine gefilterte Policy Wishlist übergeben. Die Filterung erfolgt dabei mittels eines vom 'AgentClassloader' bestimmten Filters (basierend auf der CodeBase des Agenten) welcher vom 'FilterRepository' angefordert wird. Falls kein entsprechender Filter vorhanden ist wird der Default FilterSatz angewendet. Die Anwendung eines FilterSatzes besteht dabei in einer sequentielle Folge von Ersetzungen, die durch die 'Filter' Klasse mittels des 'Substituter' ausgeführt werden.
Eine Ersetzung wird dabei durch einen 'SubstitutionEntry', welcher aus einem Namen, einem regulären Matchingausdruck und einem Ersetzungsausdruck besteht, repräsentiert.
Alle Zugriffe auf Filtersätze erfolgen dabei durch ein zentrales FilterRepository, um die persistente Repräsentation eines FilterSatzes leicht austauschen oder modifizieren zu können. Um eine plattformunabhängige Speicherung der FilterSätze zu erreichen, werden diese als XML-Daten innerhalb des MASA Systems gesichert. Das Parsen der XML Daten wird dabei mittels eines SAX - Parsers, welcher mit dem Apache XERCES API[Xerces] entwickelt wurde, vorgenommen.
Das folgende Diagramm zeigt hierbei einen Ausschnitt des Systems, wobei die gestrichelten Linien funktionale Abhängigkeiten beschreiben. Dabei bezeichnet eine funktionale Abhängigkeit den Aufruf von Methoden auf der Ziel Klasse.

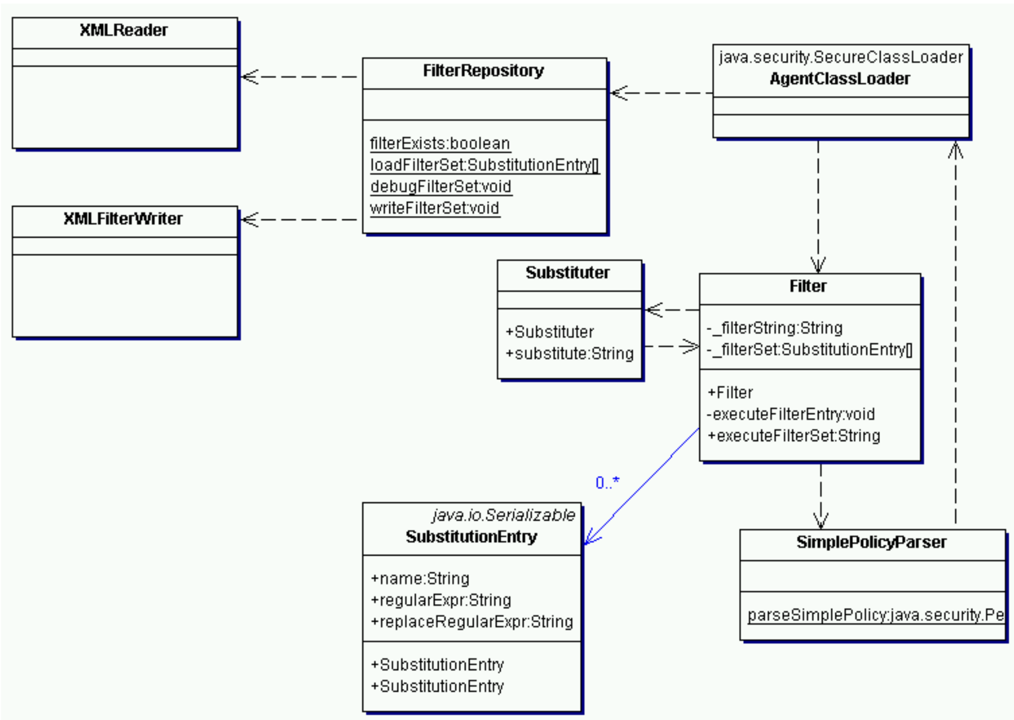

Abbildung 7: Ausschnitt der Policy Wishlist Filter Implementierung

5.2 Der Policy Editor Agent

Der Agent wurde als Automat implementiert, wobei ein Attribut des Agentenobjektes den internen Zustand repräsentiert. Dieses Attribut wird bei der Migration einer Agenteninstanz ausgewertet um die entsprechenden Aktionen auszuführen. So kann der Agent in einem von vier Zuständen sein, welche durch die Policy Manager Komponente gesetzt werden können. Durch den Aufruf der Methoden gatherPolicies(agentenSystems), distributePolicies(agentenSystems) und filterPolicies(agentenSystems) wird das Zustands Attribut dementsprechend gesetzt und die entsprechende Aktion angestossen. In dem Agenten werden dann die für die Aktionen nötigen Informationen (beispielsweise die zu verteilende Policy) gespeichert und beim Migrieren des Agenten serialisiert. Das Zustandsdiagramm (Abbildung 8) veranschaulicht nochmal die Funktionsweise des Agenten,

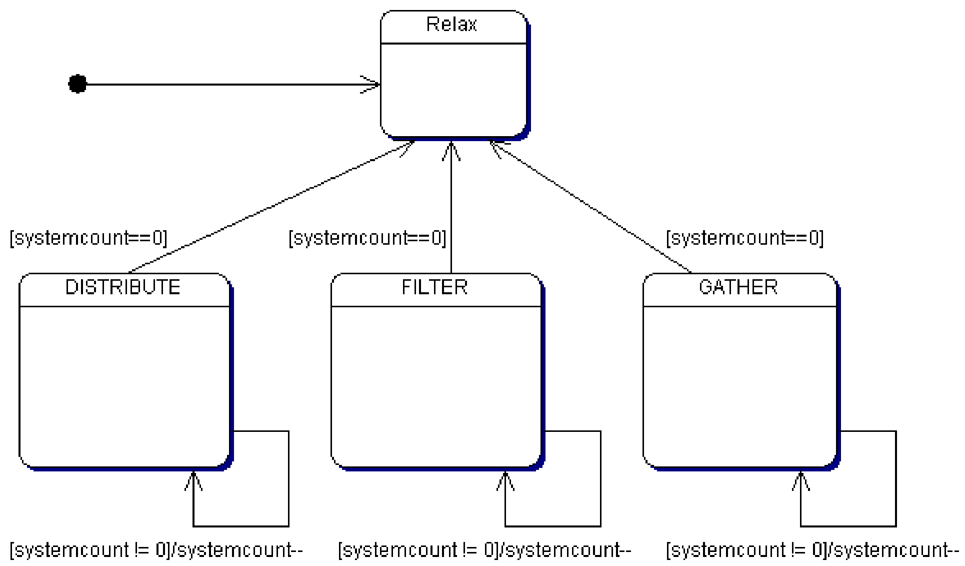

Abbildung 8: UML Zustandsdiagramm des Agenten

wobei das Attribut `systemcount` die Anzahl der noch zu besuchenden Systeme repräsentiert.

5.3 Die Graphische Benutzeroberfläche

Da die einzelnen Komponenten der graphischen Benutzeroberfläche durch ihre Spezifität kaum wiederverwendet werden können, wurden sie als kompakte funktionelle Einheiten entwickelt. Um eine nahtlose Integration in das MASA Look and Feel zu gewährleisten und eine ansprechende Benutzerführung zu ermöglichen, wurde auf die Verwendung eines Tools zur visuellen Komposition verzichtet. Das folgende Diagramm (Abbildung 9) zeigt die einzelnen Klassen der graphischen Benutzeroberfläche und deren funktionellen Abhängigkeiten (gestrichelte Linien). Die zentrale Kom-

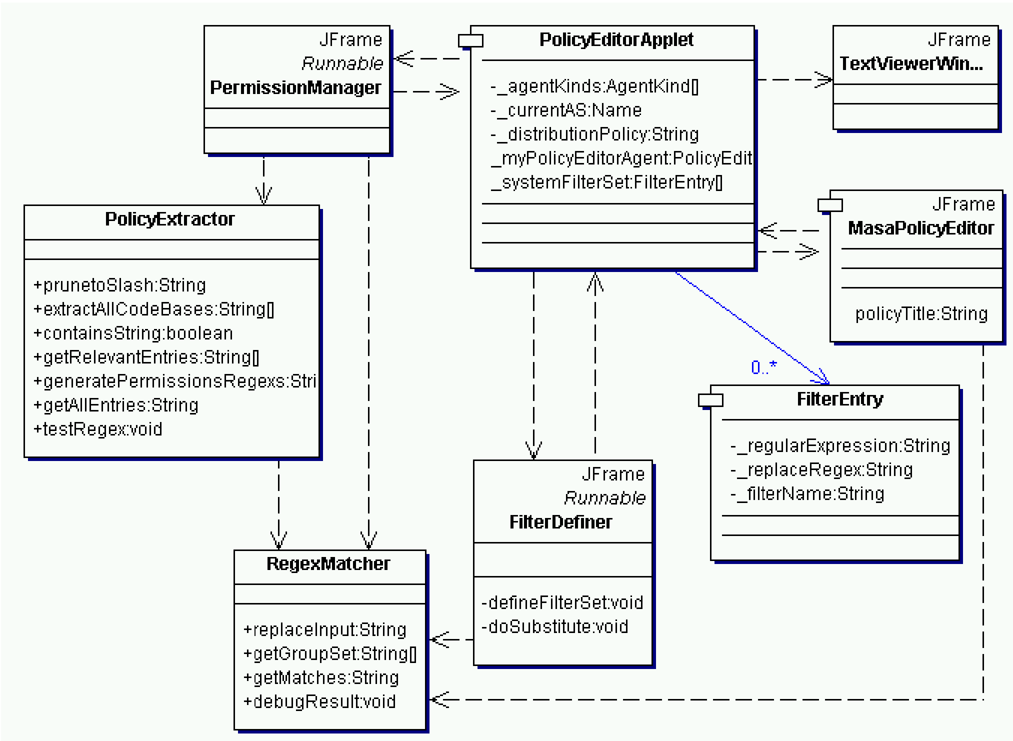

Abbildung 9: Funktionale Übersicht der Graphischen Benutzerschnittstelle

ponente bildet die 'PolicyEditorApplet' Klasse. Alle separaten Komponenten lassen sich hieraus aufrufen und anzeigen. Die Komponenten sind dabei Ableitungen der JFrame Klasse und komplett integrierte Einheiten welche Methoden auf dem übergebenen Applet Objekt aufrufen um mit dem System oder dem Agenten zu interagieren. Der wichtigste Bestandteil besteht dabei in der PermissionManager Komponente (Abbildung 3), welche für die Security Konfiguration einer Agentengattung verantwortlich ist. Hierbei wird mittels der 'PolicyExtractor' Klasse die aktuelle System Policy nach relevanten Einträgen durchsucht und in der Komponente angezeigt. Für Operationen mit regulären Ausdrücken ist hierbei die Klasse 'RegexMatcher' verantwortlich.

5.4 Verwendete Bibliotheken

Zur Unterstützung regulärer Ausdrücke kam die Jakarta-ORO[Oro] Bibliothek zum Einsatz, welche sich durch eine hohe Funktionalität und Performanz auszeichnet. Sie unterstüzt die Perl5- und AWK-Syntax für reguläre Ausdrücke[9]. Für die XML Unterstützung wurde das Apache Xerces Framework[Xerces] benutzt. Hier wurde ein SAX Parser für die plattformunabhängige Speicherung der Filter Definitionen entwickelt. Beide Bibliotheken stehen unter der Apache Software License, die bis auf eine Kennzeichnung der Bibliotheksverwendung im Quellcode und in der Binärversion keinerlei Einschränkungen aufweist.

[9]siehe Anhang: Überblick über die unterstützten Ausdrücke der Perl 5 Syntax

6 Zusammenfassung und Ausblick

Mit dieser Arbeit wurden für die Mobile Agent System Architecture Komponenten zum Managment der Sicherheitspolicies von Agenten und deren Laufzeitsystemen entwickelt. Hierbei wurde das Konzept der Agent Policy Wishlists um eine Filterkomponente erweitert und in das bestehende System integriert. Hierfür wurden eigene Graphische Benutzerschnittstellen entworfen und implementiert um ein komfortables und effizientes Arbeiten zu ermöglichen.

6.1 Weitere Arbeiten

Die Möglichkeit zur dynamischen Anpassung von Sicherheitspolicies während der Laufzeit eines Agenten konnte jedoch mit den zur Verfügung stehenden Mitteln (JDK1.2) [Sun00] nicht erreicht werden. So können Anpassung nur zur Laufzeit des Systems erfolgen. Die Modifikationen an den Sicherheitseinstellungen einer Agentengattung wirken sich demnach nur nach einem erneuten Starten des Agenten aus.
Um eine dynamische Änderung der Agenten Sicherheitseinstellungen zu erreichen, ist eine Migration auf das JDK1.4 oder höher unumgänglich, da innerhalb des JDK1.2 die ProtectionDomains intern durch den SecureClassloader gecached und somit nicht aktualisiert werden können. Durch das Wegfallen des internen Caches könnte eine dynamische Policy ohne größere Änderungen durchgesetzt werden.

A Erster Anhang: Perl5 Regular Expressions

Here we summarize the syntax of Perl5 regular expressions, all of which is supported by the OROMatcher TM Perl5 classes. However, for a definitive reference, you should consult the perlre man page that accompanies the Perl5 distribution and also the book Programming Perl, 2nd Edition from O'Reilly & Associates. We need to point out here that for efficiency reasons the character set operator [...] is limited to work on only ASCII characters (Unicode characters 0 through 255). Other than that restriction, all Unicode characters should be useable in the package's regular expressions.

```
Alternatives separated by |
Quantified atoms
  {n,m} Match at least n but not more than m times.
  {n,}  Match at least n times.
  {n}   Match exactly n times.
  *     Match 0 or more times.
  +     Match 1 or more times.
  ?   Match 0 or 1 times.

Atoms
  regular expression within parentheses
  a . matches everything except \n
  a ^ is a null token matching the beginning of a string or line
  (the position right after a newline or right before the beginning of a string)

  a $ is a null token matching the end of a string or line
  (i.e., the position right before a newline or right after the end of a string)

  Character classes (e.g., [abcd]) and ranges (e.g. [a-z])
    Special backslashed characters work within a character class
    (except for backreferences and boundaries).
    \b is backspace inside a character class

  Special backslashed characters
  \b  null token matching a word boundary (\w on one side and \W on the other)
  \B    null token matching a boundary that isn't a word boundary
  \A    Match only at beginning of string
  \Z  Match only at end of string (or before newline at the end)
  \n  newline
  \r  carriage return
  \t  tab
  \f  formfeed
  \d  digit [0-9]
  \D  non-digit [^0-9]
  \w  word character [0-9a-z_A-Z]
  \W  a non-word character [^0-9a-z_A-Z]
  \s  a whitespace character [ \t\n\r\f]
  \S  a non-whitespace character [^ \t\n\r\f]
  \xnn  hexadecimal representation of character
  \cD   matches the corresponding control character
  \nn or \nnn
        octal representation of character unless a backreference.
  \1, \2, \3, etc.
        match whatever the first, second, third, etc. parenthesized group matched.
```

```
       This is called a backreference. If there is no corresponding group,
       the number is interpreted as an octal representation of a character.
  \0   matches null character
       Any other backslashed character matches itself
```

Expressions within parentheses are matched as subpattern groups and saved for use by certain methods.

By default, a quantified subpattern is greedy. In other words it matches as many times as possible without causing the rest of the pattern not to match. To change the quantifiers to match the minimum number of times possible, without causing the rest of the pattern not to match, you may use a ? right after the quantifier.

```
  *?      Match 0 or more times
  +?      Match 1 or more times
  ??      Match 0 or 1 time
  {n}?    Match exactly n times
  {n,}?   Match at least n times
  {n,m}?  Match at least n but not more than m times
```

```
Perl5 extended regular expressions are fully supported.

(?#text)        An embedded comment causing text to be ignored.

(?:regexp)      Groups things like "()" but doesn't cause the group match to be saved.

(?=regexp)      A zero-width positive lookahead assertion. For example, \w+(?=\s)
                matches a word followed by whitespace,
                without including whitespace in the MatchResult.

(?!regexp)      A zero-width negative lookahead assertion. For example foo(?!bar)
                matches any occurrence of "foo" that isn't followed by "bar".
                Remember that this is a zero-width assertion, which means
                that a(?!b)d will match ad because a is followed by a character
                that is not b (the d) and a d follows the zero-width assertion.

(?imsx)         One or more embedded pattern-match modifiers. i enables case
                insensitivity, m enables multiline treatment of the input,
                s enables single line treatment of the input, and x enables
                extended whitespace comments.
```

B Zweiter Anhang: Apache Software License

```
/* ====================================================================
 * The Apache Software License, Version 1.1
 *
 * Copyright (c) 2000 The Apache Software Foundation.  All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. The end-user documentation included with the redistribution,
 *    if any, must include the following acknowledgment:
 *       "This product includes software developed by the
 *        Apache Software Foundation (http://www.apache.org/)."
 *    Alternately, this acknowledgment may appear in the software itself,
 *    if and wherever such third-party acknowledgments normally appear.
 *
 * 4. The names "Apache" and "Apache Software Foundation", "Jakarta-Oro"
 *    must not be used to endorse or promote products derived from this
 *    software without prior written permission. For written
 *    permission, please contact apache@apache.org.
 *
 * 5. Products derived from this software may not be called "Apache"
 *    or "Jakarta-Oro", nor may "Apache" or "Jakarta-Oro" appear in their
 *    name, without prior written permission of the Apache Software Foundation.
 *
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED.  IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
 * USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 * ====================================================================
 *
 * This software consists of voluntary contributions made by many
 * individuals on behalf of the Apache Software Foundation.  For more
 * information on the Apache Software Foundation, please see
 * <http://www.apache.org/>.
```

Literaturverzeichnis

[Fack 01] FACKELMANN, D.: *Rechtekonzept für die Mobile Agent System Architecture.* Diplomarbeit, Ludwig-Maximilians-Universität München, April 2001.

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

[HeAb 94] HEGERING, H.-G. und S. ABECK: *Integrated Network and System Management.* Addison–Wesley, 1994.

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

[Oaks 98] OAKS, SCOTT: *Java Security.* The Java Series. O'Reilly, 1998.

[Oro] *Apache Jakarta ORO.*

[Roel 99] RÖLLE, H.: *Authentisierung und Autorisierung für das Java/CORBA–Agentensystem MASA.* Diplomarbeit, Technische Universität München, August 1999.

[Sun00] *JavaTM 2 Platform, Standard Edition, v1.2.2 API Specification.* API Documentation, SUN, 2000.

[Xerces] *Apache XML Xerces Java Parser.*