Diplomarbeit

Konfiguration und Durchsetzung von Sicherheitsspezifikationen

mit Hilfe einer

erweiterten Java-Laufzeitumgebung

Thomas Bühren

21. Dezember 2000

Diplomarbeit an der

Universität Dortmund
Fachbereich Informatik
Lehrstuhl 10 für Software-Technologie

http://ls10-www.cs.uni-dortmund.de

Betreuer

Prof. Dr. Volker Gruhn
Dipl.-Math. Dirk Peters

Thomas Bühren

buehren@peperoni.de
Tel. +49-171-1241058

Zusammenfassung

Java bietet eine konfigurierbare Zugriffskontrolle, mit der ein Anwender oder sein Administrator in einer Sicherheitspolitik festlegen kann, welche Anwendung und welcher Benutzer auf welche Systemressourcen zugreifen darf. Allerdings kann in der Sicherheitspolitik nicht definiert werden, *wie* diese Zugriffe ausgeführt werden, ob dabei also Sicherheitsmechanismen wie Verschlüsselung oder Authentifizierung eingesetzt werden sollen; solche Dienste müssen bereits vom Anwendungs- oder Komponentenentwickler integriert werden.

Dies hat mehrere Nachteile: Möchte der Anwender zusätzliche Sicherheitsmechanismen einsetzen, so ist er auf Anwendungen oder Komponenten angewiesen, von denen seine Sicherheitsziele berücksichtigt werden. Die Entwickler können sich nicht auf die Fachlichkeit konzentrieren, sondern müssen ebenso Sicherheitskonzepte kennen und korrekt anwenden. Die Anwendungskomponenten müssen daher die kryptographischen Schlüssel zur Initialisierung der eingesetzten Algorithmen kennen, was bei fehlerhaften Implementierungen oder unzureichendem Schutz der Schlüssel zu großen Sicherheitslücken führen kann.

Aus diesen Gründen erscheint es vorteilhaft, wenn der Anwender in einer Sicherheitspolitik auch solche Algorithmen vorschreiben könnte und die Java-Laufzeitumgebung diese bei Ressourcenzugriffen unabhängig von der Anwendung ausführen würde.

In dieser Arbeit wird daher eine Lösung zur *Konfiguration und Durchsetzung von Sicherheitsspezifikationen mit Hilfe einer erweiterten Java-Laufzeitumgebung* vorgestellt. Die Lösung besteht zum einen aus einer Erweiterung der Java Virtual Machine, die eine Integration zusätzlicher Funktionalität in bestehende Klassen der Java-Klassenbibliothek ohne deren Veränderung durch *Wrapping* ermöglicht, und zum anderen aus der konfigurierbaren und erweiterbaren Bibliothek *JSEC (Java Security Enforcement and Configuration)*, die bei Ressourcenzugriffen zusätzliche Sicherheitsdienste anwendet, die in einer Sicherheitspolitik spezifiziert werden können.

Diese Bibliothek besitzt eine Architektur, die eine Erweiterung um weitere Ressourcentypen, um neue Sicherheitsdienste und um zusätzliche Bedingungen für die Anwendung dieser Dienste vorsieht und eine Zusammenarbeit solcher, auch unabhängig voneinander entwickelter Bestandteile ermöglicht.

Die Standardimplementierung dieser Bausteine ermöglicht eine Anwendung kryptographischer Algorithmen und eine Protokollierung bei Netzwerk- und Dateizugriffen; die Sicherheitspolitik wird in XML-Dokumenten spezifiziert. Dazu werden Konzepte und Implementierungen der existierenden Java-Sicherheitsbibliotheken JCA, JCE, JSSE, JAAS und des Security Managers eingesetzt und deren Erweiterbarkeit ebenfalls unterstützt.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die Ergebnisse dieser Diplomarbeit ermöglichen es, die von Anwendungen genutzten Sicherheitsdienste unabhängig von der Software-Entwicklung erst zum Zeitpunkt der Installation hinzuzufügen und zu konfigurieren.

Dadurch wird die Anwendungsentwicklung von Sicherheitsaspekten befreit, was eine Verringerung von Aufwand und möglichen Fehlerquellen verspricht. Sicherheitsmerkmale können so auch nachträglich zu Anwendungen hinzugefügt werden, wobei die Benutzung und die Parametrisierung sämtlicher Sicherheitsdienste für jede Installation individuell festgelegt werden können.

Das Java-Sicherheitsmodell wird so erweitert, daß nicht nur die Zugriffskontrolle, sondern auch weitere Sicherheitsmechanismen in einer Sicherheitspolitik definiert werden können und nicht bei der Entwicklung von Anwendungen vorgegeben werden müssen. Beispielsweise kann für Dateien oder Netzverbindungen neben den Zugriffsberechtigungen auch eine Verschlüsselung oder Integritätsprüfung konfiguriert werden.

Dazu wurde eine Bibliothek von Klassen entwickelt, die in Verbindung mit einer für diese Zwecke weiterentwickelten Java Virtual Machine eingesetzt werden kann und die erforderliche Funktionalität bietet. Dabei wurden einerseits existierende Sicherheitsdienste eingebunden, andererseits aber auch neue Funktionalität zu dieser Bibliothek hinzugefügt.

1.1 Motivation

Damit in der Java-Laufzeitumgebung eine Anwendung, eine Komponente oder eine Klasse aus einer Klassenbibliothek auf sicherheitsrelevante Methoden oder Systemressourcen wie das Dateisystem oder Netzwerkverbindungen zu anderen Rechnern zugreifen darf, muß diesem Software-Teil zunächst die Erlaubnis dazu gegeben werden. Dies geschieht durch die Definition einer Sicherheitspolitik, die auch jetzt schon je nach Installation angepaßt werden kann [Sun98e].

Die Ressource, für die eine solche Erlaubnis gelten soll, kann genau bestimmt werden. Zum Beispiel kann eine Zugriffserlaubnis für eine Datei, ein Verzeichnis oder für einen bestimmten Netzwerk-Port auf einem bestimmten Rechner definiert werden [Sun98d]. Durch Angabe einer Herkunftsquelle werden die Klassen festgelegt, für die eine Erlaubnis gilt. Eine Quelle

kann unter anderem ein lokales oder ein entferntes Verzeichnis sein. Zusätzlich zum Herkunftsort kann gefordert werden, daß die Klassen von einem angegebenen Unterzeichner signiert wurden, um deren Herkunft beweisen zu können [Gon98a]; damit wird auch ein Austausch von Klassen gegen namensgleiche Klassen mit fremdem Inhalt verhindert.

Mit diesen Mitteln kann definiert werden, welche Klassen mit welcher Herkunft auf welche Ressourcen zugreifen können. Es kann jedoch nicht festgelegt werden, in welcher Weise der Zugriff geschieht. Sollen vertrauliche Informationen über eine nicht vertrauenswürdige Netzwerkverbindung übertragen werden, so muß der Entwickler der Klassen dafür Sorge tragen, daß entsprechende Maßnahmen wie Verschlüsselung auch eingesetzt werden.

Der Anwender, der die Klassen später einsetzt, muß also darauf vertrauen, daß der Entwickler die Sicherheitsaspekte bedacht und korrekt implementiert hat. Dieses Vertrauen ist aus mehreren Gründen problematisch. Zum einen könnte es sich um Klassen oder Komponenten handeln, die ursprünglich nicht unter Berücksichtigung der für den Anwender wichtigen Schutzziele entwickelt wurden. Ebenso kann sich die Einsatzumgebung von Klassen ändern, beispielsweise wenn eine ursprünglich nur im internen Netz ausgeführte Anwendung auch über öffentliche Netze ausgeführt werden soll. Und schließlich können durch die Notwendigkeit der Berücksichtigung von Sicherheitsmerkmalen in jeder einzelnen Klasse potentiell auch jedesmal neue Sicherheitslücken entstehen.

Die Belastung der Anwendungsentwickler mit sicherheitsbezogenen Details widerspricht zudem mehreren Prinzipien [GJM91] des Software Engineering:

- *Strukturierung.* Eine Unterteilung in die Aspekte der Sicherheit und der Anwendungslogik ist nicht gegeben. So stellen die Sicherheitsaspekte eine lästige und eventuell vernachlässigte Zusatzaufgabe für die Anwendungsentwickler dar.
- *Abstraktion.* Der Anwendungsentwickler kann sich nicht auf die Anwendungslogik konzentrieren und von den Sicherheitsaspekten abstrahieren.
- *Modularität.* Die Module für Anwendungslogik und Sicherheitsaspekte sind direkt gekoppelt, was eine Wiederverwendung der implementierten Anwendungslogik in verschiedenen Sicherheitsszenarien erschwert.
- *Allgemeinheit.* Die Möglichkeit der Parametrisierung von Software ermöglicht deren Einsatz unter verschiedenen Rahmenbedingungen. Diese Allgemeinheit für verschiedene Sicherheitsszenarien muß jedoch vom Anwendungsentwickler geschaffen werden.

Es gibt zwar verschiedene Lösungsansätze, die allgemein einsetzbare und konfigurierbare Sicherheitsdienste zur Verfügung stellen, aber die Nutzung dieser Dienste bleibt weiterhin dem jeweiligen Software-Entwickler überlassen. Dadurch kann insbesondere im Hinblick auf komponentenbasierte Software-Entwicklung und den vielfachen Einsatz zugekaufter Komponenten die Gesamtsicherheit eines Software-Systems nicht gewährleistet werden.

Darum wird die Reichweite der konfigurierbaren Sicherheitspolitik in dieser Arbeit so erweitert, daß nicht nur eine Kontrolle der Zugriffe auf Systemressourcen, sondern auch die Zuschaltung weiterer Sicherheitsmechanismen damit möglich ist.

1.2 Lösungsansatz

In diesem Abschnitt wird zunächst die Grundidee der entwickelten Lösung erläutert. Der Lösungsweg wird in mehrere separat zu lösende Teilprobleme gegliedert und anschließend anderen, vergleichbaren Ansätzen aus der Literatur gegenübergestellt.

1.2.1 Grundidee

Als Grundlage für die Implementierung der beabsichtigten Lösung ist es erforderlich, Zugriffe auf Systemressourcen manipulieren zu können, und zwar ohne Veränderung an den Quelltexten oder an den compilierten Klassendateien der ausgeführten Anwendungsklassen. Durch diese Manipulation kann der übertragene Inhalt verschlüsselt oder signiert werden und zusätzliche Informationen in die Datenströme eingefügt werden, wovon bei der Anwendungsentwicklung völlig abstrahiert werden kann.

Die Anwendungsklassen benutzen nämlich weiterhin die gewöhnlichen, zum Zugriff auf Systemressourcen vorgesehenen Klassen der Java-Klassenbibliothek. Beim Schreiben in eine Systemressource müssen die Daten also nach der Übergabe an die zuständige Java-Klasse dort aufgefangen und modifiziert werden. Analog muß beim Lesen aus einer Systemressource der ursprüngliche Inhalt wieder hergestellt werden, bevor die Java-Klasse die Informationen an die Anwendungsklasse ausliefert.

Die Quelltexte der Java-Klassenbibliothek sind für Forschungszwecke frei verfügbar. Somit können die Klassen, die den Zugriff auf Systemressourcen ermöglichen, mit den nötigen Schritten zur Veränderung und späteren Wiederherstellung der Datenströme erweitert werden. Nach dem Compilieren der Klassenbibliothek kann diese in die Laufzeitumgebung integriert werden. Um die erweiterten Sicherheitsdienste nutzen zu können, müssen Anwendungen in dieser veränderten Laufzeitumgebung ausgeführt werden.

Nachteil dieser Lösung ist die Tatsache, daß für jedes neu ausgelieferte Java-Update zunächst die Quelltexte verfügbar sein müssen und darin die ursprünglich vorgenommenen Änderungen erneut durchgeführt werden müssen. Dieses Vorgehen ist aufwendig, zeitraubend und fehleranfällig. Die Portierung auf andere Betriebssysteme bringt zusätzlich neue Varianten der Quelltexte und somit zusätzlichen Anpassungsaufwand mit sich.

Aus den genannten Gründen wird in dieser Arbeit das Ziel verfolgt, die neuen Sicherheitsautomatismen separat zur Java-Laufzeitumgebung als Drittprodukt installieren zu können und dabei weitgehend unabhängig von der eingesetzten Java-Version zu sein. Vorteile dieser Lö-

sung wären eine Installation ohne Austausch der kompletten Laufzeitumgebung und eine erheblich einfachere Portierung zu neuen Java-Versionen.

Die geplanten Erweiterungen betreffen jedoch viele verschiedene Bereiche der Java-Klassenbibliothek und die Virtual Machine. Da die äußeren Eingriffsmöglichkeiten durch den Schutz vor bösartigen Klassen durch verschiedene Java-Sicherheitsmechanismen eingeschränkt sind, können die gewünschten Erweiterungen nicht vollständig ohne Änderungen an den Quelltexten der Java-Laufzeitumgebung durchgeführt werden.

Durch Einsatz einer weiterentwickelten Java Virtual Machine können die oben angesprochenen Klassen aus der Java-Klassenbibliothek, die zum Zugriff auf Systemressourcen dienen, trotz der umfangreichen neuen Funktionalität unverändert bleiben. Dieser Ansatz bringt eine Unabhängigkeit der Änderungen von der Java-Klassenbibliothek, die häufiger als die Virtual Machine in neuen Releases modifiziert wird.

Ausgangspunkt der Überlegungen ist der Lademechanismus: Klassen werden in Java bei ihrer ersten Benutzung durch einen Class Loader in den Arbeitsspeicher geladen [Gon98a]. Stellt das Java-System fest, daß eine benötigte Klasse noch nicht im Arbeitsspeicher vorhanden ist, so wird der qualifizierte Name der benötigten Klasse an den Class Loader übergeben, der die Klasse dann beispielsweise aus einem im Klassenpfad angegebenen Verzeichnis oder aus der Java-Klassenbibliothek lädt.

Hier bietet sich ein zentraler Ansatzpunkt, um die Funktionalität einzelner Klassen zu verändern: Der Ladevorgang wird innerhalb der Java Virtual Machine so modifiziert, daß statt der Klassen aus der Java-Klassenbibliothek spezielle andere Klassen geladen werden, die dieselben Methoden zur Verfügung stellen wie die Java-Klassen, aber zusätzlichen Code enthalten, um die Datenströme wie gewünscht zu manipulieren. Diese alternativen Klassen können sich ihrerseits wiederum auf den eigentlich ersetzten Klassen abstützen, um die dann veränderten Daten mit den üblichen Java-Methoden wie gewohnt verarbeiten zu lassen.

Da die neuen Klassen sich wie eine Hülle um die ursprünglichen Klassen legen und alle Methodenaufrufe abfangen können, werden sie im folgenden auch als *Wrapper* bezeichnet. Dieser Wrapping-Mechanismus ist allgemein genug, um weitreichende Änderungen an der Java-Klassenbibliothek ohne eine Änderung an deren Quelltexten durchführen zu können. Die Wrapper umschließen die zu verändernden Klassen und sind so konfigurierbar, daß sie bei Methodenaufrufen für Zugriffe auf definierbare Systemressourcen zusätzliche Sicherheitsmechanismen zwischenschalten.

Es gibt jedoch eine Einschränkung der Reichweite dieser Vorgehensweise. So können Anwendungsteile, die aus Native Code [Sun97] bestehen und sich nicht auf der Java-Klassenbibliothek abstützen, nicht mit erweiterten Sicherheitsdiensten ausgestattet werden. Die originären Java-Sicherheitsmechanismen bleiben bei Klassen, die eigenen Native Code benutzen, aber ebenfalls unwirksam [Ven99]. Außerdem kann in der Sicherheitspolitik genau festgelegt werden, welche Klassen auf Native Code zugreifen dürfen und welche nicht

[Sun98d]. Darum wird dieser Aspekt in der vorliegenden Arbeit bewußt unberücksichtigt gelassen.

1.2.2 Zerlegung in Teilprobleme

Der Wrapping-Mechanismus stellt die erste Teilaufgabe dar und erlaubt die Umhüllung beliebiger Klassen mit eigenen und so die Einfügung zusätzlicher Funktionalität. Daher ist eine weitere Teilaufgabe die Entwicklung eben dieser Wrapper-Klassen für alle Java-Klassen, die einen Zugriff auf Systemressourcen anbieten. So sind Wrapper zum Zugriff auf Datei- und Netzwerkressourcen zu entwickeln, um die übertragenen Daten zu verschlüsseln oder zu signieren und zusätzliche Informationen für weitere Sicherheitsmechanismen im bereits genutzten Datenstrom über das Netzwerk zu übertragen.

Die von den Wrappern zusätzlich benötigte Funktionalität wird als dritte Teilaufgabe in einer eigenständigen Klassenbibliothek separat entwickelt. Diese Klassenbibliothek bündelt einerseits Teile der vorhandenen Java-Sicherheitsbibliotheken [Sun98c, Sun98e, Sun99a, Sun99b, Sun00a, Sun00c, Sun00d], wo dies zur Vereinfachung der Benutzung in den Wrappern sinnvoll erscheint. Andererseits werden darin zusätzliche Dienste zur Verfügung gestellt, die noch nicht in Form von nutzbaren Bibliotheken vorhanden sind. Ein Beispiel dafür ist die Verwaltung und das Abfragen aller Einstellmöglichkeiten der Wrapper und deren Sicherheitsmechanismen.

Diese Klassenbibliothek steht zum einen den Wrappern zur Verfügung, um die Sicherheitsdienste in der gewünschten Form durch nachträgliche Konfiguration einsetzen zu können. Zum anderen kann die Klassenbibliothek aber auch bei der Entwicklung von Anwendungen genutzt werden, wenn die zusätzlichen Dienste in herkömmlicher Weise durch explizite Programmierung genutzt werden sollen.

Zusammenfassend wird die Gesamtlösung in diese drei Teilbereiche aufgeteilt:

- Der *Wrapping-Mechanismus* ermöglicht wie beschrieben den Austausch einzelner Klassen, um deren Verhalten zu verändern.
- In einer *Klassenbibliothek* werden alle Klassen zusammengefaßt, die für die Konfigurierbarkeit des Systems und für zusätzlich entwickelte Sicherheitsdienste benötigt werden.
- Die *Wrapper-Klassen* hüllen Teile der Java-Klassenbibliothek ein und fügen die zusätzlich benötigte Funktionalität hinzu. Sie werden über den Wrapping-Mechanismus in das System eingefügt und benutzen die entwickelte Klassenbibliothek für ihre Funktionalität.

Somit entstehen neben der Lösung des Gesamtproblems auch zwei separat wiederverwendbare Teile, die in anderen Kontexten eingesetzt werden können. Insbesondere bietet sich die Möglichkeit, weitere Wrapper zu implementieren, die andere Aufgaben haben als die im Rahmen dieser Arbeit entwickelten.

1.2.3 Abgrenzung zu ähnlichen Lösungsansätzen

In der Literatur wurden mehrere Lösungen vorgestellt, die sich aufgrund verschiedener Aspekte mit dem hier vorgestellten Ansatz vergleichen lassen. An der TU Dresden wurde eine Plattform zur Bereitstellung multiliteraler Sicherheit in verteilten Anwendungen geschaffen [PSW98]. Hierbei kann die Benutzung von Verschlüsselungs- oder Signierungsalgorithmen zur Sicherstellung der Vertraulichkeit oder Integrität von über Netzwerke übertragenen Daten ebenfalls durch den Anwender in einer Sicherheitspolitik nachträglich zugeschaltet werden. Zusätzlich kann der gewünschte Grad von Sicherheit vorgegeben werden, der sich beispielsweise in der automatischen Auswahl von Algorithmen niederschlägt. Bei dieser Lösung wird insbesondere Wert auf die Unabhängigkeit der beteiligten Parteien in verteilten Anwendungen gelegt. Darum werden die letztendlich benutzten Algorithmen und Parameter erst beim Verbindungsaufbau unter den Systemen selbst ausgehandelt. In diesen Punkten geht die genannte Lösung über den Rahmen dieser Diplomarbeit hinaus. Das wichtigste Differenzierungsmerkmal ist jedoch die dazu notwendige Benutzung spezieller Klassenbibliotheken bei der Anwendungsentwicklung. So können zwar die Algorithmen – teils automatisch – nachträglich ausgewählt werden, aber der Entwickler einer Anwendung muß diese Möglichkeit explizit vorgesehen haben. Als zukünftiger Anknüpfungspunkt wäre eine Verschmelzung der Ergebnisse beider Arbeiten denkbar. So könnten die dort entwickelten Bibliotheken als Basisdienst in der Klassenbibliothek dieser Arbeit benutzt werden. Wegen größerer Übersichtlichkeit und zur Vermeidung zusätzlicher Einarbeitungszeit werden im Rahmen dieser Arbeit jedoch die von Sun ursprünglich angebotenen Sicherheitsbibliotheken eingesetzt.

Die Abgrenzung durch nachträgliche, automatische Einbindung von Sicherheitsdiensten gilt auch für diese Sun-Bibliotheken. Der Java Authentication and Authorization Service (JAAS) beispielsweise bietet seinem Namen entsprechend Methoden für Benutzerauthentifizierung und -autorisierung. JAAS muß jedoch recht aufwendig in Anwendungen integriert werden [Sun00a]. In dieser Arbeit soll JAAS unabhängig von der Anwendungsentwicklung eingesetzt werden können. Gleiches gilt für die Java Cryptography Architecture (JCA) [Sun99a], Java Cryptography Extension (JCE) [Sun00d] und die Java Secure Socket Extension (JSSE) [Sun00c].

Damit wird also das in Java implementierte Sicherheitsmodell so verändert, daß für Zugriffe auf Systemressourcen anhand einer Sicherheitspolitik nicht nur über die Erlaubnis entschieden wird, sondern ebenso über weitere anzuwendende Sicherheitsmechanismen. Eine detaillierte Untersuchung der Eigenschaften des veränderten Sicherheitsmodells würde über den Rahmen dieser Arbeit hinausgehen, könnte aber auch die Analyse zur Durchsetzbarkeit von Sicherheitspolitiken aus [Sch98] enthalten.

Ähnlich wie in dieser Arbeit wurde auch zuvor bereits mit Veränderungen an Virtual Machine und Class Loader experimentiert, beispielsweise durch Dirk Balfanz und Li Gong [BG97]. Dabei war jedoch die Zielsetzung eine andere, nämlich die gleichzeitige Ausführung mehrerer Anwendungen innerhalb einer Java Virtual Machine ohne gegenseitige Beeinflussung. Dies war aufgrund weitreichender Auswirkungen nicht ohne massive Veränderung der Java-Umgebung möglich. Ein Teil der Lösung dabei war die Einführung eigener Class Loader für

die verschiedenen gleichzeitig laufenden Anwendungen. So entstehen getrennte Namensräume und die Anwendungen können Klassen gleichen Namens aber unterschiedlicher Inhalte benutzen. In dieser Arbeit werden Änderungen am Lademechanismus vorgenommen, die über die Möglichkeiten normaler Class Loader hinausgehen.

In [LB98] wird vorgeschlagen, zur nachträglichen Modifikation bestehender Klassen deren Bytecode von einem speziellen Class Loader bearbeiten zu lassen. Damit wäre im Gegensatz zur hier vorgestellten Lösung eine rein Java-basierte Entwicklung ohne Eingriffe in die Java Virtual Machine möglich. Ein Nachteil ist jedoch die komplizierte Definition der einzufügenden Funktionalität auf Bytecode-Ebene; bei der hier vorgestellten Lösung können Wrapper als Java-Klassen entwickelt und vom herkömmlichen Compiler zu Bytecode verarbeitet werden. Zudem wird die Java-Klassenbibliothek nicht von einem Class Loader geladen, so daß deren Ersetzung damit nicht möglich wäre, was aber für die Ziele dieser Arbeit notwendig ist.

Dieselbe Einschränkung betrifft auch die in [WBD97] vorgeschlagene Lösung mit einem speziellen Class Loader, der unterschiedlichen Programmteilen verschiedene Versionen einer Klasse zur Verfügung stellt. Mit einem solchen Namespace Management wird ähnlich wie in dieser Arbeit zusätzliche Funktionalität zu Systemklassen hinzugefügt. An unterschiedliche aufrufende Klassen werden dabei verschiedene Versionen einer bestimmten Klasse herausgegeben, die jeweils andere Funktionalität enthalten. Ein Nachteil dieser Lösung ist die nach dem Laden starre Verbindung der Funktionalität mit den aufrufenden Klassen. So ist die Entscheidung über eine Veränderung der Ressourcenzugriffe nur zum Zeitpunkt des Ladens möglich, was zwar einen Laufzeitvorteil bietet, aber die Konfigurationsmöglichkeiten einschränkt. Im Gegensatz dazu werden Klassen bei der hier vorgestellten Lösung im Laufzeitsystem global durch andere ersetzt und die Parametrisierung der Sicherheitsmechanismen bei jedem Öffnen einer Ressource durchgeführt. Damit existiert auch weiterhin genau eine Version jeder Klasse. So ist sichergestellt, daß die in [LB98] vorgeschlagenen und in JDK 1.2 implementierten Regeln zur Typsicherheit zwischen mehreren Namespaces eingehalten werden, die einen Einsatz der Lösung aus [WBD97] mittlerweile ausschließen. Der Klassenaustausch wird in dieser Arbeit innerhalb der Virtual Machine vor dem Aufruf des Lademechanismus durchgeführt, so daß auch Klassen der Java-Bibliothek ersetzt werden können, die nicht von den herkömmlichen Class Loadern geladen werden.

Weiterentwickelte Java Virtual Machines kommen auch in kommerziellen Systemen zum Einsatz. So wurden für den Enterprise JavaBeans Application Server Gemstone/J[1] die Quelltexte der Hotspot Virtual Machine [DG98] lizenziert, um die proprietäre Persistent Cache Architektur (PCA) integrieren zu können, die Objekte in einem zentralen Speicher für mehrere, parallel betriebene Java Virtual Machines verfügbar macht und zusätzlich eine automatische Persistenz dieser Objekte bietet.

Da Enterprise JavaBeans [MS01] in einem Container betrieben werden und mit Beans auf einem anderen Server keine direkte Kommunikation stattfindet, könnte eine Verschlüsselung

[1] http://www.gemstone.com/

dieser Netzwerkzugriffe auch vom Container nach einer eingestellten Konfiguration veranlaßt werden. Da die Beans neben der Nutzung des Containers aber auch die normale Java-Klassenbibliothek einsetzen und somit auch eigene Netzwerkverbindungen aufbauen und auf andere Systemressourcen wie Dateien zugreifen können, die nicht vom Container verwaltet werden, bietet sich auch für Enterprise JavaBeans der Einsatz der hier entwickelten Mechanismen an. Da auch ein System mit Enterprise JavaBeans eine Java Virtual Machine voraussetzt, sollte deren Austausch kein Problem darstellen. Ausnahmen sind Systeme wie das angesprochene Gemstone/J[1], das bereits eine veränderte Virtual Machine enthält, deren geänderte Quelltexte erneut erweitert werden müßten.

Bei der Betrachtung von Java im Kontext der Sicherheit werden große Anstrengungen unternommen, um Sicherheitslücken in Java zu finden oder zu schließen. Es gab und gibt wahrscheinlich immer noch zahlreiche Angriffsmöglichkeiten, um die in Java vorhandenen Sicherheitsmechanismen zu umgehen oder Fehler im Java-System auszunutzen [Gon99, MF99, KNN98, Pfl97]. Gegenmaßnahmen sind neben der Behebung von Fehlern oder dem Einbau zusätzlicher Laufzeitüberprüfungen auch verbesserte Bytecode Verifier, um die Klassen vor ihrer Ausführung auf Verletzungen der Java-Konventionen zu überprüfen [Hag98]. Solche Überlegungen tragen jedoch nicht zur Lösung des in dieser Arbeit behandelten Problems bei und sind somit nicht Gegenstand der Untersuchungen.

Ebenso wird keine Optimierung durch den Einsatz besonders schneller oder sicherer Algorithmen zur Verschlüsselung oder zur Verteilung von dazu notwendigen Schlüsselpaaren angestrebt [RC97, Sch96, GW90]. Vielmehr sollen verschiedene Algorithmen variabel eingesetzt werden können, ohne daß diese zum Zeitpunkt der Anwendungsentwicklung berücksichtigt werden müßten.

Eine ähnliche Zielsetzung verfolgt auch die IP Security Protocol Working Group (IPSEC), die ein Sicherheitsprotokoll in der Netzwerkschicht entwickelt hat [Cal98]. Damit können Authentifizierung, Zugriffskontrolle, Vertraulichkeit und Integrität ebenfalls ohne Veränderung der Anwendungen hergestellt werden [IBM99]. In einer Sicherheitspolitik können zu verschiedenen Verbindungsendpunkten die gewünschten Verschlüsselungs- und Signierungsalgorithmen eingestellt werden, die durch einen speziellen IP Stack auf alle übertragenen Daten angewendet werden. Dabei kann der Einsatz der Algorithmen jedoch nicht gezielt auf bestimmte Benutzer, Anwendungen oder Anwendungsteile eingeschränkt werden. Ein weiterer Nachteil dieser Lösung ist die unumgängliche Erweiterung des Vertrauensbereichs auf das Betriebssystem und den IP Stack. So werden kritische Daten aus dem Anwendungsprozeß unverschlüsselt und unsigniert an das Betriebssystem und von dort an die Netzwerktreiber übergeben. Hier bieten sich zusätzliche Angriffsmöglichkeiten, beispielsweise für Trojaner. Im Gegensatz dazu verlassen die kritischen Daten bei einer Benutzung der Sicherheitsalgorithmen innerhalb des Java-Systems den Betriebssystemprozeß der Java Virtual Machine nicht unbehandelt, was den Vertrauensbereich auf die Java-Umgebung einschränkt.

Auch in Java gibt es bereits Klassen, die als Wrapper bezeichnet werden; dies sind Klassen, die primitive Typen einschließen und als Objekt repräsentieren [GJS00]. Auch Klassen, die mit Benutzung des Java Native Interface [Sun97] zur Einhüllung plattformabhängiger Pro-

gramme dienen und damit Altsystemen eine Java-Schnittstelle geben, werden als Wrapper bezeichnet. Die Wrapper-Klassen dieser Arbeit umschließen dagegen andere Java-Klassen und werden ohne Programmänderungen von einer Anwendung statt der gewrappten Klassen genutzt.

Der Begriff Wrapper wird auch für weitere, teilweise verwandte Entwicklungen verwendet. In [JH98] wurde eine sichere Laufzeitumgebung für plattformspezifische Anwendungen vorgestellt, die als Sandbox bezeichnet wird. Es wird versucht, die in Java vorhandene Sicherheit bei der Ausführung unbekannter Anwendungen nachzuempfinden, indem statt der Anwendung ein Wrapper aufgerufen wird. Dieser aktiviert in seinem Betriebssystemprozeß zunächst einen Benutzer mit so wenig Rechten wie möglich und ruft unter diesem Benutzer das eigentliche Programm auf. Auch diese Lösung stützt sich auf bestehenden Sicherheitsmechanismen ab, was aber dazu führt, daß nur Dateisystemzugriffe kontrolliert werden können und Netzwerkzugriffe nicht, da Betriebsysteme dies üblicherweise nicht abhängig vom Benutzer bieten.

Dies umgeht [GWT96], indem ein Wrapping der Betriebssystemaufrufe durchgeführt wird, so daß alle Funktionen des Betriebssystems kontrolliert werden können, ähnlich wie es auch durch diese Arbeit mit der Java-Klassenbibliothek möglich ist. Dazu wird das UNIX-spezifische Process Tracing eingesetzt, um alle System Calls abzufangen, gegebenenfalls zu modifizieren und dann weiterzuleiten. Nachteil dieser Lösung ist die Beschränkung auf UNIX und das nötige Abfangen aller, auch der nicht zu modifizierenden Betriebssystemaufrufe. Die vorliegende Arbeit ist auf Java beschränkt, damit aber auf allen davon unterstützten Plattformen einsetzbar. Durch die konsequente Nutzung der Objektorientierung werden die Wrapper nur bei Methodenaufrufen in Klassen aktiv, für die Modifikationen vorgesehen wurden.

Noch enger verwandt mit der Idee dieser Arbeit ist der Ansatz aus [FBF99], bei dem ebenfalls Betriebssystemaufrufe abgefangen und gegebenenfalls modifiziert werden. Auch dort ist die Zielsetzung, bei der Entwicklung von Software-Komponenten auf die Programmierung von Sicherheitsmechanismen verzichten und diese nachträglich in einer Sicherheitspolitik definieren zu können. Der Ansatz bietet die Möglichkeit, mit einer Wrapper-Beschreibungssprache vordefinierte Aktionen wie Zugriffskontrollen oder Schreiben von Protokolldateien festzulegen, die unter definierbaren Aktivierungsbedingungen ausgeführt werden. Diese proprietäre Definitionssprache wurde eingeführt, um übliche Aktionen ohne Programmierung eigener Wrapper durchführen zu können, da deren Entwicklung durch die Betriebssystemnähe kompliziert und fehleranfällig ist. Im Gegensatz dazu können anwendungsspezifische Wrapper für den Wrapping-Mechanismus dieser Arbeit als herkömmliche Java-Klassen entwickelt werden, was die Implementierung vereinfacht; Wrapper für übliche Aufgaben werden im Rahmen dieser Arbeit bereitgestellt. Die Aktivierungsbedingungen und deren Auswirkungen werden in Konfigurationsdateien festgelegt.

1.3 Aufbau der Arbeit

Die Einleitung in diesem ersten Kapitel der Arbeit enthielt neben einer Motivation zur Veränderung der Java-Laufzeitumgebung auch eine Vorstellung des grundlegenden Lösungsansatzes sowie dessen Zerlegung in Teilprobleme und einen Vergleich mit ähnlichen Ansätzen.

Kapitel 2 stellt die Grundlagen und Ziele der Sicherheit informationstechnischer Systeme sowie die in Java implementierten Sicherheitskonzepte vor. Darauf aufbauend legt die Anforderungsdefinition in Kapitel 3 fest, welche Sicherheitsziele in dieser Arbeit verfolgt und welche Mechanismen zusätzlich zu den für Java verfügbaren implementiert werden sollen.

Kapitel 4 stellt den entwickelten Wrapping-Mechanismus vor, mit dem Teile der Klassenbibliothek ersetzt werden können. Die Bibliothek von Sicherheitsmechanismen aus Kapitel 5 enthält die Funktionalität zur Erfüllung der festgelegten Anforderungen und wird von dazugehörigen Wrapper-Klassen für eine Modifikation des Verhaltens bei Zugriffen auf Systemressourcen eingesetzt.

Die gesamte Lösung wird in Kapitel 6 anhand einer bestehenden Client-Server-Anwendung validiert, die ursprünglich nicht unter Beachtung von Sicherheitsanforderungen entwickelt wurde. Im Fazit in Kapitel 7 werden die gesammelten Erfahrungen zusammengefaßt und als Ausblick mögliche Weiterentwicklungen angeregt.

Der Anhang enthält Spezifikationen und Konfigurationsdateien als Dokumentation der entwickelten Software sowie eine Liste aller an der Java-Umgebung vorgenommenen Änderungen als Grundlage für die Portierung zu anderen Java-Versionen. Danach folgen Glossar und Literaturverzeichnis.

1.3.1 Typographische Konventionen

Kursivschrift wird erstens bei der Einführung von Begriffen verwendet, die damit hervorgehoben und im nachfolgenden Text erneut verwendet werden, und zweitens als Betonung der entscheidenden Ausdrücke bei Aufzählungen.

Quelltexte, Kommandozeilenbefehle und Konfigurationsdateien sowie Bezeichner aus diesen werden in der Schriftart `Courier` dargestellt, worauf bei mehrfacher Wiederholung jedoch verzichtet wird.

Bei der ersten Benutzung eines im Glossar aufgeführten Begriffs wird mit einem →Pfeil darauf hingewiesen. Für Literaturhinweise werden eckige Klammern verwendet, zum Beispiel [BCC99], deren Quellenangaben im Literaturverzeichnis zu finden sind.

2 Grundlagen

In diesem Kapitel werden die Grundlagen für eine Umsetzung des vorgestellten Lösungsansatzes beschrieben. Zunächst wird ein Überblick zur Sicherheit bei der elektronischen Datenverarbeitung gegeben, wobei von konkreten Implementierungen weitgehend abstrahiert wird. Anschließend werden in 2.2 die in Java implementierten Sicherheitskonzepte und die von Sun zur Verfügung gestellten Sicherheitsbibliotheken beschrieben.

Die in Java nicht implementierten Möglichkeiten werden zusammen mit den Sicherheitsgrundlagen bei der Anforderungsdefinition im danach folgenden Kapitel 3 berücksichtigt, während die vorhandenen Java-Sicherheitsmechanismen anschließend bei der Implementierung eingesetzt und dazu teilweise modifiziert werden.

2.1 Sicherheit informationstechnischer Systeme

Mögliche Schutzmechanismen zur Abwendung von Gefahren für die Sicherheit informationstechnischer Systeme lassen sich in drei Klassen aufteilen [Lip90]:

- *Rechtlicher Schutz* bietet einen Abschreckungseffekt durch Androhung einer Bestrafung.
- *Organisatorischer Schutz* bezeichnet Maßnahmen wie Brandschutz, eine physikalische Zugangskontrolle zu Rechnern [Wec90] oder eine sichere Verlegung von Netzwerkleitungen als Schutz vor physikalisch-technischen Schäden sowie eine Ausbildung und Sensibilisierung der Benutzer.
- *Software-technischer Schutz* umfaßt alle Schutzmechanismen innerhalb eines Datenverarbeitungssystems, beispielsweise die des Betriebssystems, wie Zugriffskontrolle oder Authentifizierung sowie die Protokollierung der Aktivitäten von Benutzern oder Programmen. Darunter fallen auch Verschlüsselung und weitere Mechanismen, die von Anwendungen eingesetzt werden.

Bei der Betrachtung der Sicherheit eines informationstechnischen Systems müssen Aspekte aller Bereiche beachtet werden. Da diese Arbeit in die Klasse des Software-technischen Schutzes fällt, wird hier aber nur auf solche Gesichtspunkte eingegangen.

2.1.1 Schutzziele

Die Sicherheit informationstechnischer Systeme erfordert die Erhaltung der drei Eigenschaften *Vertraulichkeit*, *Integrität* und *Verfügbarkeit* [Pfl97]. In diesem Abschnitt werden diese erläutert sowie mögliche Schutzmechanismen vorgestellt.

2.1.1.1 Vertraulichkeit

Vertraulichkeit bedingt, daß bestimmte Informationen nicht an Unbefugte gelangen können. Welche Informationen dies sind, hängt von den gestellten Sicherheitsanforderungen ab; so kann die Identität eines Benutzers durch Pseudonymisierung geschützt werden und die Information über die bloße Existenz einer Kommunikationsbeziehung durch Mix-Konzepte [Cha81]. In dieser Arbeit wird die Vertraulichkeit oder Geheimhaltung von Daten behandelt, so daß nur befugte Nutzer oder Prozesse diese lesen können. Die einfachste Möglichkeit, ungewollte Zugriffe auf Daten zu vermeiden, ist die Speicherung an einer Stelle, die nur den berechtigten Personen bekannt ist; dies bietet allerdings nur mangelhaften Schutz.

Einen berechenbareren Schutz bietet die Einführung einer Zugriffskontrolle, bei der die Software überwacht, daß nur solche Benutzer auf Daten zugreifen können, für die der Zugriff erlaubt wurde. Solche Zugriffskontrollmechanismen sind üblicherweise auf Dateisystemebene in Betriebssystemen integriert. Nachteil ist, daß die Daten unter Umgehung des Betriebssystems oder durch Ausnutzung seiner Fehler trotzdem gelesen werden können. Auch ein Schutz gegen Abhören einer Netzwerkübertragung ist so nicht möglich.

Hier bietet eine Verschlüsselung von Daten den Vorteil, daß der gespeicherte oder übertragene Datenstrom zwar abgehört werden kann, aber in einer Form vorliegt, die einen Rückschluß auf die tatsächlichen Nutzdaten unter normalen Umständen nicht zuläßt. Wird jedoch der zur Entschlüsselung notwendige Schlüssel entwendet, so ist dies damit möglich. Daher muß der Sicherung von Schlüsseln besondere Beachtung geschenkt werden, und diese müssen beispielsweise durch einen Zugriffskontrollmechanismus geschützt werden. Daneben ist es jedoch auch möglich, die Nutzdaten aus einem verschlüsselten Datenstrom durch Raten oder Ausprobieren zu entziffern. In Abschnitt 2.1.3 werden verschiedene Verschlüsselungskonzepte erläutert.

Beim Schutz der Vertraulichkeit müssen Angriffe vor allem verhindert werden. Die Erkennung unerlaubten Lesens ist nicht nur schwierig, sondern in den meisten Fällen auch nutzlos. Ein Problem besteht neben der Einrichtung eines wirksamen Schutzes auch in der Definition der zugriffsberechtigten Parteien; einige Modelle dazu werden in Abschnitt 2.1.2 vorgestellt.

2.1.1.1.1 Covert Channels

Ein bemerkenswerter Aspekt ist der Verlust von vertraulichen Daten über Kanäle, die ursprünglich gar nicht zur Übertragung von Daten vorgesehen sind und daher bei einer Sicherheitsbetrachtung leicht übersehen werden. Solche *Covert Channels* [NCS93] treten an vielen unerwarteten Stellen auf und könnten von findigen Angreifern ausgenutzt werden. Beispiels-

weise könnte ein Sender periodisch auf eine bestimmte Stelle der Festplatte zugreifen, während der Empfänger dies an einer anderen Stelle und damit an einer anderen Position des Schreib- und Lesekopfes versucht. Mit dieser Methode können einzelne Bits übertragen werden, indem der Sender je nach Bitwert auf seine Position zugreift oder nicht; an der Verzögerung, die bei seinem Festplattenzugriff durch die Bewegung des Kopfes auftritt, kann der Empfänger den gesendeten Bitwert messen, wobei Einflüsse durch andere Programme noch geeignet gefiltert werden müssen.

Dieses Beispiel verdeutlicht, daß es unwahrscheinlich ist, alle möglichen Covert Channels eines Systems beachten zu können. Allerdings muß der Sender in solchen Szenarien zunächst unter Überwindung von Sicherheitsmechanismen in das System hineingebracht werden. Ist dies möglich, so kann mit großer Wahrscheinlichkeit statt des nur aufwendig nutzbaren Covert Channel auch ein offener Kanal mißbraucht werden, um die Daten zu übertragen. Offensichtlich auch aus diesem Grund wird den Convert Channels in Java keine Beachtung geschenkt [Gon99].

In dieser Arbeit liegt der Fokus ebenfalls nicht auf der Erkennung einer Nutzung von Covert Channels durch böswillige Komponentenentwickler, sondern auf der nachträglichen Absicherung offener Kanäle, deren Nutzung zunächst rein fachlich implementiert werden kann.

2.1.1.2 Integrität

Die Integrität bzw. Unversehrtheit von Daten bedingt, daß Änderungen daran nur von Befugten in einer erlaubten Weise vorgenommen werden können, wobei neben dem tatsächlichen Ändern auch Schreiben, Löschen und Erzeugen von Daten zu beachten sind. Dies kann wie der Schutz der Vertraulichkeit ebenfalls durch eine Kontrolle der Zugriffe geschützt werden.

Nach einer Integritätsverletzung ist es dagegen einfacher, diese später festzustellen. Dazu kann mit einer Einweg-Hash-Funktion der Hash-Wert von Daten berechnet und zusätzlich gespeichert oder übertragen werden. Nach dem Lesen bzw. Empfang der Daten wird der Hash-Wert erneut berechnet, der mit dem vorherigen übereinstimmt, solange die Daten korrekt übermittelt wurden. Weitere Gesichtspunkte dazu folgen in 2.1.3.1.

Mit der Kontrolle des Hash-Werts kann eine Integritätsverletzung nicht verhindert, aber im Nachhinein festgestellt werden. Dann können die ursprünglichen Daten aus einem Backup wiederhergestellt oder bei einer Netzwerkverbindung erneut übertragen werden.

Eine Kontrolle, ob Daten in erlaubter Weise verändert wurden, ist anwendungsabhängig und wird in vielen Implementierungen nicht besonders beachtet [Pfl97]. Dies kann zu Problemen führen, wenn die verarbeitenden Programme die Daten falsch interpretieren, und Folgefehler können zu unsicheren Zuständen des Systems mit größeren Sicherheitslücken oder zu Abstürzen und damit einer Einschränkung der Verfügbarkeit des Systems führen.

2.1.1.3 Verfügbarkeit

Verfügbarkeit eines Systems bedeutet, daß dieses zur Erfüllung seiner Aufgaben tatsächlich genutzt werden kann; das Gegenteil wird als *Denial of Service* bezeichnet. Neben direkten Angriffen auf die Verfügbarkeit, beispielsweise durch die Störung eines Netzwerks, können auch andere Attacken ein Denial of Service verursachen, etwa die Überflutung eines Netzwerks oder eines Rechners mit großen Datenmengen, was zu einer Einschränkung der Leistung und im schlimmsten Fall zu einem Ausfall des Systems führen kann. Ebenso kann eine Verletzung der Integrität bei der späteren Verarbeitung durch ein Programm wie beschrieben zu Fehlern während der Ausführung und damit eventuell zu einem Zustand führen, in dem dieses Programm oder andere Systemteile nicht mehr verfügbar sind.

Ein Schutz vor Denial-of-Service-Attacken ist schwierig und zur Zeit in den meisten Systemen nicht implementiert [Pfl97]. Vorgänge, die eine Bedrohung der Verfügbarkeit darstellen, müssen dazu erkannt werden. Im Fall einer hohen Netzbelastung ist es aber schwierig zu entscheiden, ob diese durch einen Angriff oder durch erhöhte, aber durch normale Nutzung bedingte Aktivität verursacht wird. Neben der Erkennung muß das System solche Angriffe aber zunächst schadlos überstehen, was bei der Blockade übermäßig vieler Systemressourcen nicht immer gewährleistet ist. Auch in der Java-Laufzeitumgebung gibt es bisher keine wirkungsvollen Schutzmechanismen; diese sollen aber für folgende Versionen erforscht werden [Gon99]. In dieser Arbeit wird keine Untersuchung dieser Richtung durchgeführt.

2.1.2 Sicherheitspolitiken und Sicherheitsmodelle

Theoretisch könnte man ein System genau dann als sicher bezeichnen, wenn keines der in 2.1.1 beschriebenen Ziele Vertraulichkeit, Integrität und Verfügbarkeit verletzt werden kann. Dies ist aber im allgemeinen für ein gegebenes System noch nicht einmal entscheidbar. Daher wird ein System als sicher bezeichnet, wenn es dazu geeignet ist, die in einer *Sicherheitspolitik* definierten Anforderungen in der Praxis umzusetzen [Ker91].

Eine Sicherheitspolitik spezifiziert also die gewünschte Sicherheit, die von einem System gewährleistet werden soll. Damit ist die sicherheitstechnische Güte eines Systems nur relativ zur festgelegten Sicherheitspolitik meßbar, deren Zweckmäßigkeit nicht allgemein, sondern nur unter Berücksichtigung der vom System zu erfüllenden Aufgaben überprüft werden kann.

Ein *Sicherheitsmodell* legt Eigenschaften fest, welche Sicherheitspolitiken besitzen müssen, die dem Modell entsprechen. Es gibt Sicherheitsmodelle, die in bestimmten Situationen und für bestimmte Aufgaben besser geeignet sind als andere. Hier werden einige Sicherheitsmodelle aus der Literatur [Pfl97, Ker91] in kurzgefaßter Form dargestellt. Dabei werden aktive Systembestandteile wie Benutzer, Prozesse und Rechner als *Subjekte* und passive wie Dateien oder Datensätze als *Objekte* bezeichnet.

Der nötige Grad an Sicherheit ist immer abhängig von den drohenden Gefahren, gegen die man sich absichern möchte, und vom – teilweise schwer definierbaren – Wert der zu schützenden Objekte. Die Erreichung der Sicherheit muß dabei mit vertretbarem Aufwand möglich

sein, um eine Akzeptanz durch die Benutzer zu erreichen. Die Sicherheitspolitik ist damit ein Kompromiß zwischen Erfüllbarkeit der Aufgaben und Gewährleistung von Schutz.

2.1.2.1 Protected Perimeter

Dieser Kompromiß wird auch beim Modell des *Protected Perimeter* deutlich, bei dem davon ausgegangen wird, daß alle Benutzer eines lokalen Netzes unschädlich und alle außerhalb unheilbringend sind. Daher wird mit einem Protected Perimeter versucht, das lokale Netz vom externen zu trennen. Eine vollständige physikalische Trennung ist zwar denkbar, verhindert aber auch eine effiziente Ausführung von Aufgaben, die beispielsweise über elektronische Post, das World Wide Web oder Netzwerkverbindungen mit anderen Firmen erledigt werden könnten.

Daher wird die strikte Trennung gelockert und ein Filter zwischen internem und externem Netz eingesetzt. Eine solche *Firewall* [Cal98] entscheidet für jedes zu übertragende Datenpaket anhand der beteiligten Rechner, Protokolle, Anwendungen oder anderer Kriterien, ob es die Netzgrenze passieren darf oder nicht. Solche Einstellungen der Firewall werden in einer Sicherheitspolitik festgelegt. Neben der Filterung kann auch eine unbemerkte Umleitung oder Verschlüsselung der Datenpakete eingestellt werden. Mit *Encrypting Firewalls* können also virtuelle private Netzwerke (VPN) aufgebaut werden, indem der Datentransfer zwischen den Firewalls zweier Unternehmensstandorte verschlüsselt über öffentliche Netze übertragen wird.

Das lokale Netz darf an keiner anderen Stelle als der Firewall mit dem öffentlichen verbunden werden; die gesamte externe Kommunikation muß über die Firewall laufen, wodurch sie schnell zu einem Engpaß oder Ausfallgrund eines ganzen Netzwerks werden kann. Zudem sind Firewalls oft hinderlich für Mobile Code (mehr dazu in Abschnitt 2.1.5), der nicht die für seine Zwecke nötigen Netzwerkverbindungen aufbauen kann. Auch andere Programme müssen speziell angepaßt werden, um eine Firewall als Proxy nutzen zu können.

Ein Protected Perimeter allein ist im übrigen meist nicht ausreichend, da auch den eigenen Mitarbeitern nicht generell getraut werden kann. Angriffe von Insidern sind sehr gefährlich, da die eigenen Mitarbeiter mehr Vorwissen als fremde Angreifer haben.

2.1.2.2 Mandatory Access Control

Bei *Mandatory Access Control (MAC)* wird darum von einer zentralen Autorität festgelegt, welche Subjekte auf welche Objekte zugreifen dürfen. Ein Beispiel dafür ist das *Bell-LaPadula-Modell*, das aus dem militärischen Bereich stammt und auch als *Multilevel Security* bezeichnet wird. Subjekten und Objekten sind *Sicherheitsstufen* zugeordnet (unclassified, classified, secret, top secret o. ä.). Je zwei Stufen stehen entweder in einer Dominiert-Beziehung oder sind inkompatibel, so daß eine Halbordnung der Stufen entsteht. Ein Subjekt kann nur ein Objekt beschreiben, von dessen Level sein eigener dominiert wird, und nur dann lesen, wenn sein Level den des Objekts dominiert. Einfacher wird dies auch als write-up und read-down bezeichnet, so daß ein Informationsfluß nur entlang der Halbordnungsbeziehungen

nach oben erlaubt ist. Subjekte mit höherer Sicherheitseinstufung können so selbst abgeleitete Daten hoch eingestufter Objekte nicht in neue Objekte weitergeben, die von niedrigeren Subjekten gelesen werden können. Eine bidirektionale Kommunikation zwischen zwei Partnern kann allerdings nur stattfinden, wenn beiden dieselbe Sicherheitsstufe zugeordnet ist.

2.1.2.3 Discretionary Access Control

Mit *Discretionary Access Control (DAC)* wird jedem Objekt ein Besitzer zugeordnet, der nach seinem Ermessen selbst festlegen kann, wer außer ihm lesend oder schreibend darauf zugreifen darf. Die Auswirkungen einer festgelegten Sicherheitspolitik sind jedoch nicht direkt nachvollziehbar; so kann ein berechtigter Leser beispielsweise eine Kopie eines Objekts erzeugen und dieses neue Objekt für weitere, zuvor nicht berechtigte Subjekte freigeben. Die Zugriffskontrolle bei vielen Dateisystemen ist auf diese Weise geregelt.

2.1.2.4 Dynamische Modelle

Eine festgelegte Sicherheitspolitik bleibt im praktischen Einsatz nicht statisch bestehen, sondern ändert sich dynamisch. So können sich die Zugriffsberechtigungen der Benutzer oder die Sicherheitseinstufung von Dateien mit der Zeit ändern. Daher gibt es besondere Sicherheitsmodelle, die solche Aspekte der Dynamik besser unterstützen. Das *Chinese-Wall-Modell* ist auf die Situation in Beratungsgesellschaften abgestimmt, von denen sensitive Daten konkurrierender Firmen verarbeitet werden. Ein Berater kann zunächst auf die Daten aller Klienten zugreifen, aber nach dem Zugriff auf die Daten einer Firma nicht mehr auf die der anderen.

Daneben wurden weitere Modelle entwickelt [Pfl97]; darunter auch solche, die im Gegensatz zu den hier vorgestellten andere Hauptziele als die Geheimhaltung der Objekte haben, insbesondere auch deren Integrität.

Das in Java verwendete Sicherheitsmodell wird in Abschnitt 2.2.2 vorgestellt. In dieser Arbeit wird das Java-Sicherheitsmodell so erweitert, daß nicht nur die Zugriffskontrolle, sondern auch weitere Sicherheitsmechanismen in der Sicherheitspolitik eingestellt und nicht von Anwendungsentwicklern vorgegeben werden. Beispielsweise kann für sensitive Dateien oder Datenübertragungen neben den Zugriffsberechtigungen auch eine Integritätsprüfung oder ein Schutz der Vertraulichkeit durch kryptographische Algorithmen konfiguriert werden.

2.1.3 Kryptographie

Die Bereiche Kryptographie und Sicherheit sind nicht dasselbe und können zunächst völlig unabhängig voneinander betrachtet werden. Aber sie sind eng miteinander verbunden, denn eine Verschlüsselung kann die Geheimhaltung und Integrität von Daten sichern. Kryptographie ist also ein wichtiges Hilfsmittel zur Gewährleistung von Sicherheit.

Alle Verschlüsselungssysteme sind auch ohne Schlüsseldiebstahl prinzipiell unsicher, da mit genug Rechenleistung und genug Vorwissen jede Verschlüsselung auch ohne den Schlüssel geknackt werden kann. Die einzige Ausnahme ist das eher theoretische One-time Pad [Sch96], bei dem der verwendete Schlüssel so groß ist wie die Daten selbst und nach der Benutzung vernichtet wird. Zudem muß der Schlüssel über einen sicheren Kanal zum Empfänger übertragen werden.

Die folgenden grundlegenden kryptographischen Konzepte werden in den weiteren Teilen dieses Abschnitts vorgestellt:

- *Einweg-Hash-Funktionen,*
- *symmetrische Verschlüsselung* mit geheimen Schlüsseln,
- *asymmetrische Verschlüsselung* mit Paaren öffentlicher und privater Schlüsseln sowie
- *hybride Verfahren* als Kombination symmetrischer und asymmetrischer Verschlüsselung.

2.1.3.1 Einweg-Hash-Funktionen

Einweg-Hash-Funktionen [Sch96] können im Bereich Sicherheit zur Kontrolle der Datenintegrität bei Speicherung oder Übertragung von Daten eingesetzt werden. Kurz gesagt ist das Ergebnis einer Einweg-Hash-Funktion für eine Eingabe leicht zu berechnen, aber die Umkehrung schwer. Zudem ist es schwierig, zwei Eingabewerte zu finden, für die eine Einweg-Hash-Funktion denselben Hash-Wert erzeugt.

Der Hash-Wert ist üblicherweise sehr viel kleiner als die Eingabe, so daß er zusätzlich zu den Nutzdaten übertragen oder gespeichert werden kann. Nach dem Empfang der Daten wird die Hash-Funktion darauf erneut angewandt und das Ergebnis mit dem ebenfalls empfangenen Hash-Wert verglichen. Sind die Werte verschieden, so sind die Daten mit Sicherheit ebenfalls verändert worden. Bei einem gleichen Hash-Wert ist wegen des kleineren Wertebereichs gegenüber dem der Nutzdaten eine Manipulation zwar nicht ausgeschlossen, aber sehr unwahrscheinlich.

Um eine auf veränderte Daten abgestimmte Modifikation des Hash-Werts zu verhindern, kann zusätzlich zu den Nutzdaten noch ein geheimer Schlüssel als weitere Eingabe der Hash-Funktion verarbeitet werden. So ist es ohne den Schlüssel nicht möglich, den Hash-Wert zu berechnen und damit eine Modifikation zu verdecken. Der Schlüssel muß auf Sender- und Empfängerseite bekannt sein, darf aber nicht an Unbefugte gelangen.

Konkrete Einweg-Hash-Funktionen sind beispielsweise MD5 (MD für Message Digest) oder SHA (Secure Hash Algorithm) [Sta95].

2.1.3.2 Symmetrische Verschlüsselung

Durch eine symmetrische Verschlüsselung werden Nutzdaten unter Einbeziehung eines *geheimen Schlüssels* in einen Code umgewandelt, der mit demselben Schlüssel wieder zurück-

übersetzt werden kann. Daher müssen Sender und Empfänger im Besitz des Schlüssels sein, der jedoch vor Unbefugten geschützt werden muß.

Die Schlüsselverwaltung und -verteilung gestaltet sich daher schwierig, da bei vielen kommunizierenden Parteien für jede potentielle Verbindung ein Schlüssel vorhanden sein muß, so daß insgesamt exponentiell viele Schlüssel verwaltet und geschützt werden müssen. Es wurden auch Lösungen vorgestellt, die diese Größenordnung verringern, beispielsweise in [GW90]; allerdings werden dabei Kompromisse eingegangen, indem Vertrauensbereiche vergrößert werden und nicht mehr für je zwei Kommunikationspartner ein eigener Schlüssel zur Verfügung steht.

Systeme mit symmetrischer Verschlüsselung sind z. B. IDEA (International Data Encryption Algorithm) und DES (Data Encryption Standard) [Cal98, Sta95].

2.1.3.3 Asymmetrische Verschlüsselung

Asymmetrische Verschlüsselung funktioniert prinzipiell wie die symmetrische; allerdings werden statt eines einzigen nun zwei Schlüssel benötigt. Zur Verschlüsselung wird der *öffentliche Schlüssel* des Empfängers benutzt, der später seinen *privaten Schlüssel* zur Entschlüsselung einsetzen muß. So wird erreicht, daß niemand außer dem vorgesehenen Empfänger die Daten entschlüsseln kann – abgesehen von Schlüsselverlust oder erfolgreicher Umgehung des Entschlüsselungsverfahrens.

Aus dem privaten Schlüssel kann sehr einfach der öffentliche Schlüssel erzeugt werden, umgekehrt ist es schwer. Daher kann der öffentliche Schlüssel frei zugänglich sein; anders als bei der symmetrischen Verschlüsselung ist die Schlüsselverteilung nicht auf sichere Kanäle angewiesen.

Ein Nachteil ist neben der geringeren Geschwindigkeit im Vergleich mit symmetrischen Verfahren die Anfälligkeit gegen *Chosen-Plaintext-Attacken* [Sch96]. Dabei benutzt der Angreifer den öffentlichen Schlüssel des eigentlichen Empfängers eines verschlüsselten Codes und verschlüsselt damit nacheinander alle möglichen Werte aus dem Wertebereich der Nutzdaten. Stimmt das Ergebnis dieser Verschlüsselung mit dem verschlüsselten Code überein, so steht der benutzte Eingabewert als ursprünglicher Inhalt der Nachricht fest. Dieser Angriff ist jedoch nur bei einem kleinen Wertebereich der Nutzdaten erfolgversprechend, da sonst sehr viele Möglichkeiten getestet werden müssen; bei einer asymmetrischen Verschlüsselung kleiner Datenmengen können umfangreichere Zufallsdaten hinzugefügt und bei der Entschlüsselung wieder entfernt werden, um Chosen-Plaintext-Attacken zu erschweren.

2.1.3.3.1 Digitale Signaturen und Zertifikate

Bei einigen asymmetrischen Verschlüsselungssystemen sind Ver- und Entschlüsselung umkehrbar [Sta95]; die Entschlüsselungsfunktion mit dem privaten Schlüssel kann bei solchen Verfahren auf die Nutzdaten angewendet werden, die später mit der eigentlichen Verschlüsselungsfunktion und dem öffentlichen Schlüssel wiederhergestellt werden können. Um Platz

und Rechenzeit zu sparen, kann die Entschlüsselungsfunktion auch auf das kürzere Ergebnis einer Einweg-Hash-Funktion angewendet werden. Wird die so erzeugte *Signatur*, die auch als *Message Authentication Code (MAC)* bezeichnet wird, zusammen mit den Nutzdaten verschickt, kann deren Integrität durch Verschlüsselung der Signatur und anschließenden Vergleich mit einem erneut berechneten Hash-Wert der Nutzdaten geprüft werden.

Zudem kann die Signatur nur bei Besitz des privaten Schlüssels erzeugt werden, so daß der Erzeuger der Nachricht festgestellt werden kann. Von diesem ist jedoch nur der öffentliche Schlüssel bekannt, der zum Fälschen einer Nachricht auch ausgetauscht werden könnte. Um feststellen zu können, ob ein öffentlicher Schlüssel tatsächlich dem angegebenen Subjekt gehört, kann ein *Zertifikat* für den öffentlichen Schlüssel benutzt werden. Dies ist ein Datensatz, der den öffentlichen Schlüssel selbst und identifizierende Merkmale des Subjekts, etwa Namen und Adresse, enthält.

Dieser Datensatz wird von einer vertrauenswürdigen Zertifizierungsstelle mit deren privatem Schlüssel signiert, um die Zugehörigkeit des Schlüssels zum angegebenen Subjekt zu bestätigen. Da der öffentliche Schlüssel der Zertifizierungsstelle allgemein bekannt ist, kann die Echtheit des Zertifikats geprüft werden. Vertraut man dem Ausstellungsverfahren dieser Stelle, so kann man davon ausgehen, daß der öffentliche Schlüssel und die zusätzlichen Angaben im Zertifikat tatsächlich zusammengehören.

Mit verketteten Zertifikaten können auch die Schlüssel der Zertifizierungsstellen wiederum zertifiziert werden. Das Ende einer solchen Kette bildet ein nicht zertifiziertes, sondern selbstsigniertes Zertifikat einer allgemein anerkannten Stamm-Zertifizierungsstelle, das in vielen Software-Produkten bereits mitgeliefert wird, um die Zertifikate anderer Stellen und letztendlich die der Schlüsselbesitzer zu überprüfen. Ein Beispiel ist VeriSign, Inc., deren Zertifikate beispielsweise bei Microsoft Internet Explorer und Netscape Communicator mitgeliefert werden.

Um Vertraulichkeit und Integrität gleichzeitig zu gewährleisten, kann ein asymmetrischer Verschlüsselungsalgorithmus in beiden Richtungen nacheinander auf die Nutzdaten angewendet werden [Sch96].

Beispiele für asymmetrische Verschlüsselungssysteme sind der RSA-Algorithmus (von Rivest, Shamir und Adleman) für Verschlüsselung und digitale Signaturen sowie DSA (Digital Signature Algorithm), der als Standard der US-Regierung nur für digitale Signaturen und nicht für Verschlüsselung eingesetzt wird [Sch96, Sta95].

2.1.3.4 Hybride Verfahren

Um die Vorteile der öffentlichen Schlüssel zu nutzen, ohne die geringe Geschwindigkeit der asymmetrischen Verschlüsselungsverfahren in Kauf nehmen zu müssen, wird bei hybriden Verfahren [Sch96] ein geheimer Schlüssel für eine einzige Kommunikationssitzung erzeugt, der mit asymmetrischer Verschlüsselung gesichert übertragen wird.

Zunächst schickt dazu die erste Partei der zweiten ihren öffentlichen Schlüssel. Die zweite Partei erzeugt einen zufälligen, geheimen Schlüssel für die Sitzung, verschlüsselt diesen mit dem öffentlichen Schlüssel der ersten Partei und sendet ihn zurück. Die erste Partei entschlüsselt den empfangenen Sitzungsschlüssel mit dem eigenen privaten Schlüssel. Anschließend findet eine Kommunikation mit symmetrischer Verschlüsselung unter Verwendung des Sitzungsschlüssels statt, nach deren Abschluß er wieder zerstört wird.

Durch die kurzzeitige Verwendung ist die Gefahr einer Entdeckung und böswilligen Ausnutzung des geheimen Schlüssels geringer. Das Verfahren wird durch die Geschwindigkeitsvorteile gegenüber asymmetrischen Verfahren insbesondere bei Online-Kommunikation eingesetzt. Gleichzeitig kann der Vorteil einer einfacheren Verteilung öffentlicher Schlüssel genutzt werden.

Bei SSL (Secure Sockets Layer) [PRG99, Sun00c] wird dieses Verfahren angewandt. Zudem bietet SSL die Option, Server und Client über Zertifikate zu authentifizieren, bevor die eigentliche Kommunikation stattfindet.

2.1.4 Authentifizierung

Authentifizierung ist ein Vorgang zur Feststellung der Identität eines Benutzers oder eines Rechners, der im Namen eines Benutzers arbeitet. Dieses Wissen wird benötigt, um Sicherheitspolitiken mit für Subjekte festgelegter Zugriffskontrolle durchzusetzen. Besonders bei der Nutzung öffentlicher Netze ist es wichtig zu wissen, wer sich mit einem System verbinden will. Ebenso ist es für den Benutzer wichtig, daß er mit dem richtigen Rechner verbunden wird. Die Authentifizierung eines Rechners kann durch ein Zertifikat erfolgen, in dem die Netzwerkadresse des Rechners mit einem öffentlichen Schlüssel verknüpft wird; der dazugehörige private Schlüssel muß auf dem Rechner wie üblich so gesichert werden, daß er nicht entwendet werden kann.

Der grundlegende Ansatz zur Authentifizierung eines Benutzers ist die Frage nach einem Benutzernamen und einem Paßwort und der Vergleich dieser Daten mit gespeicherten Listen. Solche einfachen Lösungen sind jedoch gegen Abhören oder Erraten der Paßwörter anfällig. Da Paßwörter bevorzugt werden, die ein Mensch sich auch merken kann, werden sie meist aus einer relativ kleinen Menge ausgewählt, beispielsweise aus den Wörtern eines Wörterbuchs, und können daher durch eine automatisierte Suche herausgefunden werden.

Weiterentwicklungen beschäftigen sich daher beispielsweise mit einer Einbeziehung der für einen Benutzer charakteristischen Anschlaggeschwindigkeit von Tasten [MRW99] oder erweitern die Abfrage von Benutzername und Paßwort mit einer zusätzlichen Anforderung, auf die der Benutzer in einer bestimmten Weise antworten muß. Bei einem solchen *Challenge-Response-System* könnten etwa einige Ziffern dargestellt werden, auf welche der Benutzer nach Anwendung einer ihm bekannten Verknüpfungsfunktion, beispielsweise der Addition, durch Eingabe des Ergebniswertes antworten muß. Taschenrechner-ähnliche Geräte würden auch die Anwendung komplizierterer Funktionen ermöglichen [Pfl97].

Zudem wurden Variationen des grundlegenden Ansatzes vorgestellt, beispielsweise *One-Time Passwords*, von denen jedes Paßwort nur einmal benutzt werden kann. Nachteil dieser Erweiterung ist die beschränkte Anzahl von Paßwörtern, die ein Benutzer mitführen kann.

Statt leicht zu erratender Paßwörter können auch Smartcards und andere Hardware-basierte Mechanismen oder biometrische Verfahren [Cal98, Wec90] eingesetzt werden. Diese Möglichkeiten sollen bei den in dieser Arbeit eingesetzten Mechanismen berücksichtigt werden, so daß die genaue Methode der Authentifizierung flexibel gewählt werden kann.

2.1.5 Mobile Code

Die in Java integrierten Sicherheitskonzepte zielen vor allem darauf ab, ohne Gefährdung des eigenen Systems auch unbekannte →Anwendungen auszuführen, die aus verschiedenen Quellen im Internet stammen können. Nicht nur Java →Applets und ActiveX Controls sind solcher *Mobile Code*, sondern grundsätzlich „alles, was einen entfernten Rechner zu einem veränderten Verhalten veranlaßt“ [Gon99]. Dazu gehören also ebenso die Remote Shell in Unix mit entfernt ausführbaren Skripten, der Remote Procedure Call (RPC), aber auch Daten wie die des Domain Name Service (DNS) oder Microsoft-Word-Dateien mit Makros und ähnliches.

Mit Mobile Code wird keine neue Art von Bedrohung erzeugt, sondern lediglich bestehende Lücken in verbreiteten Sicherheitsmechanismen ausgenutzt. Bei sehr speziellem Mobile Code wie einem DNS Request sind die Sicherheitsimplikationen leicht überschaubar. Wenn dagegen vollständig allgemeiner Code wie ActiveX Controls benutzt werden, kann die gesamte Schnittstelle der Win32 APIs genutzt werden – auch ihre Sicherheitslücken.

Es gibt zwei Reaktionen auf die Verbreitung von Mobile Code [Gon99]: Zum einen wird versucht, ähnlich wie beim Protected Perimeter aus 2.1.2.1 sämtlichen Mobile Code an den Systemgrenzen abzublocken, womit jedoch die erwünschte Funktionalität ebenfalls verhindert wird. Zudem müßten dazu alle Eintrittswege und alle Formate, die Mobile Code enthalten könnten, gefiltert werden, was sicherlich nicht vollständig möglich ist. Die andere Reaktion ist der Versuch, die bestehenden Systeme sicherer zu machen, um Mobile Code besser kontrollieren und damit dessen Vorteile weiter nutzen zu können. So hat Java auch die Zielsetzung, eine sichere Plattform für die Ausführung nicht vertrauenswürdiger und mobiler Applets und →Applikationen zu sein; die in Java zunächst vor allem zu diesem Zweck integrierten Sicherheitsmechanismen werden im folgenden Teil dieses Kapitels vorgestellt.

2.2 Sicherheit in der Java-Umgebung

Java ist sowohl für Client- als auch für Server-Anwendungen mittlerweile weit verbreitet [MF99, PRG99, Gon99]. Wegen der unterschiedlichen Sicherheitsanforderungen, die sich aus verschiedenen Einsatzszenarios ergeben, wurde Java mit der Version 1.2 so ausgelegt, daß die üblicherweise benötigten Sicherheitsfunktionen bereits vorhanden sind und zusätzlich eine Erweiterbarkeit für spezifische Anforderungen gegeben ist.

Die grundlegenden Sicherheitseigenschaften des Betriebssystems, in dem die Virtual Machine ausgeführt wird, bleiben von ihr unangetastet. So sind bei vorhandener Zugriffskontrolle wie beispielsweise unter Solaris auch für die Virtual Machine nur die Ressourcen zugänglich, auf die der Benutzer zugreifen kann, von dem sie gestartet wurde. Andererseits sind bei unzureichend auf Sicherheit ausgelegten Betriebssystemen wie Microsoft Windows 98 zumindest für Java-Anwendungen alle Sicherheitseinrichtungen des Java-Systems aktiv, so daß die Sicherheit bei Ausführung von Java-Code auf diesen Systemen erhöht wird, da das eigentliche Betriebssystem hinter der →Java-Klassenbibliothek verborgen und somit nicht direkt zugreifbar ist.

Im Gegensatz zu vielen anderen Technologien wurde Java von vornherein mit dem Entwurfsziel Sicherheit entwickelt und bietet dabei auch Erweiterungsmöglichkeiten, wodurch Probleme durch einen nachträglichen Einbau der Sicherheitsmerkmale vermieden werden, wie beispielsweise die Erhaltung der Abwärtskompatibilität. Das ursprüngliche Sicherheitskonzept war noch sehr simpel, bot aber eine Grundlage zur Weiterentwicklung in den folgenden Java-Versionen.

Die Java-Plattform basiert auf der Sprachdefinition [GJS00] und auf der Spezifikation der Java Virtual Machine [LY99]. Eine Java-Laufzeitumgebung enthält grundsätzlich eine Virtual Machine sowie die Java-Klassenbibliothek [Sun00f]; zudem können weitere Bibliotheken eingesetzt werden und über den Klassenpfad oder mit dem Extension-Mechanismus [Sun99c] für Anwendungen zur Verfügung gestellt werden. In der Java-Umgebung gibt es also mehrere Aspekte der Sicherheit:

- Die *Java Virtual Machine* enthält Sicherheitsmechanismen als Grundlage zur sicheren Ausführung von Anwendungen.
- Die in Java verwendete *Zugriffskontrolle* erlaubt eine konfigurierbare Sicherheitspolitik mit erweiterbaren Zugriffsberechtigungen.
- Umfangreiche Sicherheitsdienste werden in der *Java-Klassenbibliothek* bereits mitgeliefert und können von sicherheitsrelevanten Anwendungen eingesetzt werden. Weitere Sicherheitsdienste stehen in *zusätzlichen Bibliotheken* zur Verfügung, die ebenfalls von Anwendungen benutzt werden können, aber nicht im Lieferumfang der Java-Laufzeitumgebung enthalten sind, sondern nachträglich installiert werden müssen.

In den folgenden Abschnitten werden diese Bereiche vorgestellt; die weitere Arbeit beschäftigt sich anschließend mit einer flexiblen Integration dieser Sicherheitsmechanismen in die Teile der Klassenbibliothek mit Zugriff auf Systemressourcen, so daß diese Mechanismen nicht während der Entwicklung von Anwendungen eingebunden werden müssen, sondern bei der Installation konfiguriert werden können.

2.2.1 Sicherheitsmechanismen der Java Virtual Machine

In Java können wie auch in anderen objektorientierten Sprachen Zugriffsrechte für Klassen, Methoden und Attribute vom Programmierer vergeben werden. So kann deren Nutzung auf definierte Schnittstellen eingeschränkt werden. Die Definition von Packages bietet neben einer größeren Übersichtlichkeit auch die Möglichkeit, den Zugriff auf das Package der Klasse zu beschränken. Die Einhaltung der Zugriffsrechte wird nicht nur vom Compiler geprüft, sondern auch während der Ausführung in der Virtual Machine, um Gefahren durch veränderte Compiler zu verhindern.

Zu den grundlegenden Sicherheitseigenschaften der Java-Umgebung gehört auch Typsicherheit, die beim Laden und während der Ausführung überprüft wird, was Gefahren durch fehlerhafte Speicherzugriffe und Methodenaufrufe aufgrund falscher Typumwandlungen verhindert. Zudem gibt es in Java keine unsicheren Konstrukte wie Zugriffe auf Arrays ohne Indexüberprüfung. Auf den Speicher kann nicht über beliebige Adressen, sondern nur über Objektreferenzen zugegriffen werden, deren tatsächliche Speicheradressen von der Virtual Machine verwaltet werden.

2.2.1.1 Bytecode Verification

Der von einer Virtual Machine interpretierte Bytecode wird vor der Ausführung überprüft. Zur Motivation der Gründe werden hier zunächst die Zusammenhänge kurz erläutert.

2.2.1.1.1 Virtual Machine und Bytecode

Eine Virtual Machine kennt ähnlich wie ein realer Prozessor eine definierte Menge von Instruktionen [LY99], die bei der Programmausführung abgearbeitet werden können und als Bytecode bezeichnet werden. Dieser Bytecode befindet sich mit zusätzlichen Symboltabellen in Klassendateien, die von einem Compiler aus Java-Programmen erzeugt und von der Virtual Machine geladen und anschließend ausgeführt werden.

Die Ausführung kann dabei durch Interpretation jeder einzelnen Instruktion oder durch vorherige Compilierung für die vorliegende Hardware-Plattform erfolgen, wie bei der Hotspot Virtual Machine [DG98]. Eine Virtual Machine kann zur weiteren Geschwindigkeitssteigerung

auch in einem Prozessor untergebracht werden, so daß der Bytecode direkt durch Hardware ausgeführt wird[2].

Die Virtual Machine ist prinzipiell von Java unabhängig, da sie Instruktionen und Symboltabellen verarbeitet. Der Bytecode wird von einem Compiler als Klassendatei erzeugt und ist damit unabhängig von der Java-Sprachdefinition; es können also auch Compiler zur Übersetzung anderer Sprachen in Bytecode entwickelt werden, der dann ebenso plattformunabhängig von der Virtual Machine ausgeführt werden kann[3].

2.2.1.1.2 Prüfung des Bytecode

Wird ein Compiler eingesetzt, der nicht alle Sicherheitsregeln prüft oder böswillig modifziert wurde, können im Bytecode auch Instruktionen auftreten, die bei ihrer Ausführung Zugriffsrechte oder Typsicherheit verletzen würden. Daher wird deren Einhaltung bei allen Instruktionen zur Laufzeit von der Virtual Machine geprüft. Weitere Bestandteile der *Bytecode Verification* sind Prüfungen beim Aufruf von Methoden, die sicherstellen, daß Sichtbarkeitsregeln wie beispielsweise bei Private-Methoden nicht verletzt werden, daß die richtige Anzahl von Argumenten mit den erwarteten Typen benutzt werden, und daß keine Stack-Überläufe auftreten. Ebenso wird sichergestellt, daß Zahlen nicht in Objektreferenzen umgewandelt werden, um so unerlaubt auf andere Speicherbereiche zuzugreifen.

Die Prüfungen zur Laufzeit führen zu einer verlangsamten Ausführung von Anwendungen; daher werden diese teilweise direkt nach dem Laden vom *Bytecode Verifier* statisch anhand des vorliegenden Bytecodes einmalig ausgeführt [LB98, Gol98]. Da jedoch eine statische Überprüfung aller Eigenschaften nicht oder nicht einfach möglich ist, werden viele der Prüfungen weiterhin zur Laufzeit vorgenommen. Zur Beschleunigung werden einige Instruktionen nach erfolgreicher Prüfung jedoch durch interne, nicht in der Spezifikation enthaltene Instruktionen ersetzt, so daß eine erneute Ausführung dieser Instruktion ohne zusätzliche Prüfungen durchgeführt werden kann.

Es wurden auch weitere Verfahren zur statischen Prüfung von Eigenschaften des Bytecodes vorgeschlagen, beispielsweise in [Hag98]; dieses Forschungsthema ist also noch nicht abgeschlossen. Da die statische Bytecode Verification von dieser Arbeit unberührt bleibt, wird hier jedoch nicht ausführlicher auf dieses Thema eingegangen. Allerdings werden in Abschnitt 4.3.2 einige Veränderungen an den zur Laufzeit durchgeführten Prüfungen für Wrapper vorgestellt.

---

[2] Beispiele sind *Zucotto Xpresso* (http://www.zucotto.com), *Patriot Scientific PSC1000A* (http://www.ptsc.com/), *Dallas Semiconductor TINI* (http://www.ibutton.com/TINI) oder die *Sun picoJava-Architektur* (http://www.sun.com/embedded/databook/pdf/datasheets/805-4634-02.pdf).

[3] Solche Compiler existieren beispielsweise für *Smalltalk* (http://www.smalltalkjvm.com), *Ada* (http://www.gnat.com/texts/products/pjava_set.htm) und *Eiffel* (http://www.loria.fr/projets/SmallEiffel).

2.2.1.2 Dynamic Class Loading

Dynamic Class Loading ermöglicht die Installation von Software-Komponenten zur Laufzeit oder die flexible Auswahl tatsächlich benutzter Klassen erst bei der Ausführung einer Anwendung. In dieser Arbeit wird der Mechanismus genutzt, um Wrapper statt der von ihnen gewrappten Klassen zu laden. Die folgenden Vorteile des Dynamic Class Loading in Java werden in [LB98] aufgezählt:

- *Spätes Laden:* Klassen werden erst dann geladen, wenn sie benötigt werden, wodurch Speicherverbrauch und Startzeit einer Anwendung verringert werden. Dies macht sich insbesondere bei Anwendungen bemerkbar, deren Klassen über ein Netzwerk geladen werden.
- *Anwendungsdefinierte Ladestrategie:* Class Loader sind herkömmliche Java-Objekte, wodurch Programmierer eine weitreichende Kontrolle über die Ladevorgänge haben und so beispielsweise Klassen über ein Netzwerk oder aus einem Repository geladen oder auch neue Versionen von Klassen zur Laufzeit nachgeladen werden können. Wie sich in Kapitel 4 noch zeigen wird, sind die vorgesehenen Eingriffsmöglichkeiten für die Ziele dieser Arbeit jedoch nicht ausreichend, so daß tiefgreifendere Änderungen vorgenommen wurden.
- *Typsicheres Binden:* Die Typsicherheit wird durch den dynamischen und anpassbaren Mechanismus beim Laden von Klassen nicht aufgegeben. Dazu sind zusätzliche Prüfungen notwendig, die jedoch nicht bei jeder Benutzung einer Klasse, sondern nur einmalig beim Laden der Klasse durchgeführt werden, wodurch das Laufzeitverhalten nicht übermäßig verschlechtert wird.
- *Getrennte Namensräume:* Da Klassen neben ihrem Namen auch anhand des Class Loaders identifiziert werden, der sie geladen hat, können mit mehreren Loadern getrennte Namensräume geschaffen werden, um in einer Java Virtual Machine mehrere Anwendungen ohne gegenseitige Beeinflussung oder unterschiedliche Klassen gleichen Namens benutzen zu können.

Klassen bestehen neben den auszuführenden Instruktionen auch aus einer Symboltabelle, die symbolische Referenzen auf Attribute, Methoden und Namen anderer Klassen enthält. Beim Auflösen einer symbolischen Referenz zu einer anderen Klasse wird zunächst geprüft, ob diese bereits in der Java Virtual Machine vorhanden ist. Falls nicht, wird ein Class Loader mit dem Laden der angeforderten Klasse beauftragt und ihm dazu der Name der Klasse übergeben.

Die Aufgabe eines solchen als Java-Klasse implementierten Class Loaders besteht darin, den angeforderten Bytecode zu laden und im System zu verankern. Üblicherweise werden die Klassen aus standardisierten Klassendateien im Dateisystem oder über Netzwerk geladen. Anwendungsdefinierte Class Loader können die Klassen aber auch in anderen Formaten oder aus weiteren Quellen laden oder eine Compilierung zur Laufzeit durchführen.

Ein Class Loader könnte den geladenen Klassen auch zusätzliche Instruktionen hinzufügen und so deren Funktionalität erweitern. Der Ansatz wird in dieser Arbeit jedoch nicht verfolgt, da ein Wrapper dazu auf Bytecode-Ebene definiert werden müßte und nicht als Java-Klasse implementiert werden könnte.

2.2.1.2.1 Hierarchie von Class Loadern und Delegation

Da Class Loader selbst Klassen sind, werden diese ebenfalls mittels Dynamic Class Loading geladen. Um den ersten solchen Class Loader laden zu können, enthält die Virtual Machine einen *Bootstrap Class Loader*, der alle zur ihrer Initialisierung notwendigen Klassen lädt, der aber nicht als Java-Klasse, sondern beispielsweise in C implementiert ist.

Derjenige Class Loader, von dem eine Klasse geladen wurde, wird als deren *Defining Class Loader* bezeichnet. Enthält eine Klasse eine symbolische Referenz auf eine andere, die noch nicht geladen wurde, so wird erst zum Zeitpunkt der ersten Benutzung diese Referenz aufgelöst, indem der Defining Class Loader der Ausgangsklasse mit dem Namen der referenzierten Klasse aufgerufen wird, um die benötigte Klasse zu laden [Gon98b]. Um einen anwendungsdefinierten Class Loader festzulegen, kann dieser aus der Anwendung explizit aufgerufen werden, um eine Klasse zu laden; alle von dieser Klasse referenzierten weiteren Anwendungsklassen werden anschließend von diesem Class Loader angefordert.

Die erste Klasse einer Applikation wird ohne Möglichkeit zur Beeinflussung von einem neu erzeugten `URLClassLoader` geladen, die eines Applets von einem `AppletClassLoader`; beides sind →Subklassen von `ClassLoader` und in der Klassenbibliothek enthalten.

Ein Class Loader kann die angeforderte Klasse auf seine spezifische Art und Weise laden, beispielsweise aus dem Dateisystem, über Netzwerk, aus einer Datenbank oder aus einem Repository. Zudem ist aber auch eine *Delegationsbeziehung* unter allen instanziierten Class Loadern definiert, so daß jeder Loader einen Delegation Parent kennt, den er mit dem Laden von Klassen beauftragen kann, für die er nicht zuständig ist. Der oberste Class Loader der Delegationshierarchie wird als *Primordial Class Loader* bezeichnet und entspricht in der Sun-Implementierung der Java-Laufzeitumgebung dem Bootstrap Class Loader [Gon99]. Durch die Delegation kann der mit dem Laden einer Klasse beauftragte Class Loader von deren definierendem Class Loader verschieden sein.

Der Bootstrap Class Loader ist für das Laden der Java-Klassenbibliothek zuständig und sucht deren Bestandteile im Klassenpfad für Systemklassen, dem `bootclasspath`. Klassen, die so geladen wurden, bekommen aus historischen Gründen Zugriffsrechte auf alle Ressourcen, die für andere Klassen mit einer Sicherheitspolitik geschützt werden können (siehe 2.2.2.1). Dem zum Laden einer Anwendung erzeugten `URLClassLoader` steht der Klassenpfad aus der Umgebungsvariable `CLASSPATH` bzw. aus Kommandozeilenparametern zur Verfügung.

Die hier vorgestellten Mechanismen, die in der Java Virtual Machine verankert oder eng mit ihr gekoppelt sind, bieten einerseits Erweiterungsmöglichkeiten und gewährleisten dennoch Typsicherheit, die als Grundlage für weitere Sicherheitskonzepte wichtig ist, da sich ohne

diese unkontrollierbare Angriffsmöglichkeiten durch Ausnutzung fehlerhafter Programme bieten würden [Gol98]. Die dennoch aufgetretenen Sicherheitslücken in Java [Gon99, MF99, KNN98, Pfl97] sind nicht auf konzeptionelle Schwächen, sondern auf Fehler bei der Implementierung oder Besonderheiten der Ausführungsumgebung zurückzuführen und konnten für die jeweils nächste Java-Version behoben werden [Gon99, KNN98].

2.2.2 Konfiguration und Durchsetzung von Zugriffskontrolle in Java

Die Sicherheitsziele von Java waren zu Beginn vor allem darauf ausgerichtet, Gefährdungen des lokalen Systems durch Mobile Code zu vermeiden, der in Form von Applets auch aus nicht vertrauenswürdigen Quellen des Internet geladen werden kann. Im JDK 1.0 wurde daher eine *Sandbox* verwirklicht, in der sämtlicher über Netzwerk geladener Code ohne Zugriffsrechte auf Systemressourcen wie das Dateisystem oder Netzwerkverbindungen zu dritten Rechnern ausgeführt wurde. Im Gegensatz zu diesen sehr eingeschränkten Möglichkeiten für Code aus Netzwerkquellen wurde Programmen aus dem Dateisystem vollständig vertraut und unbeschränkter Zugriff auf alle Systemressourcen gewährt.

Daß die Ausführung in einer Sandbox nicht in jedem Fall wünschenswert ist, zeigt sich am Beispiel eines unternehmensinternen Applets, das über Netzwerk jedem Mitarbeiter zur Verfügung steht und dessen Vertrauen genießt, aber dennoch in der möglichen Funktionalität stark eingeschränkt würde. Daher wurden im JDK 1.1 *Signaturen für Applets* eingeführt, womit die Quelle eines Applets überprüfbar wird; solchen von definierbaren Unterzeichnern signierten Applets wurden wie lokal installierten Anwendungen unbeschränkte Zugriffsrechte gewährt [KNN98].

Der freie Zugriff auf alle Systemressourcen ist aber weder für signierte Applets noch für lokale Applikationen optimal. Auch einer Anwendung oder →Komponente, die auf die lokale Festplatte kopiert wurde, wird nicht in jedem Fall vollständiges Vertrauen geschenkt. Eine Einschränkung der Zugriffsrechte auf die tatsächlich für die Funktionalität benötigten Ressourcen verringert die möglichen Auswirkungen eventuell bösartigen Codes und ist mit JDK 1.2 möglich.

Dessen Zugriffskontrollmechanismen unterscheiden sich in den folgenden Eigenschaften von ihren Vorgängern [Gon99]:

- *Flexibel:* Die Grenzen der Sandbox sind nicht fest, sondern können flexibel definiert werden, wobei Applets und Applikationen gleich behandelt werden.
- *Politikgetrieben:* Die konkrete Sicherheitspolitik wird nicht mehr von der Java-Laufzeitumgebung vorgegeben und muß – bis auf Spezialfälle – nicht durch Programmierung implementiert werden, sondern kann in Konfigurationsdateien vom Benutzer oder Systemadministrator festgelegt werden.
- *Erlaubnisbasiert:* In der Sicherheitspolitik werden Zugriffserlaubnisse auf Ressourcen an Programmteile vergeben.

- *Feinkörnig:* Die erlaubten Zugriffe können für jede Ressource genau spezifiziert werden, beim Dateisystem beispielsweise die Berechtigungen zum Lesen oder Schreiben einzelner Verzeichnisse und Dateien.
- *Erweiterbar:* Zur Einführung neuer Ressourcentypen muß keine aufwendige Programmierung durchgeführt werden; es ist lediglich die Definition neuer Erlaubnistypen erforderlich, die von den bestehenden Zugriffskontrollalgorithmen verarbeitet werden können. Auch anwendungsspezifische Zugriffskontrollen für geschützte Programmteile sind damit möglich.

Die Bestandteile der Java-Zugriffskontrolle werden in den folgenden Abschnitten vorgestellt: Welche Zugriffe erlaubt sind, wird in der *Security Policy* mit *Access Permissions* festgelegt; die Durchsetzung dieser Sicherheitspolitik leistet der *Security Manager* zusammen mit dem *Access Controller*.

2.2.2.1 Security Policy

Das Verhalten der Java-Laufzeitumgebung richtet sich nach der Security Policy, in der festgelegt ist, welcher Code auf welche Ressourcen zugreifen darf. Dazu gibt es in der Java-Klassenbibliothek die Klasse `java.security.Policy`. Ihre wichtigsten Bestandteile sind →statische Methoden zum Setzen und Lesen eines systemweit gültigen Policy-Objekts sowie →abstrakte Methoden zum Herausfinden der Zugriffserlaubnisse eines bestimmten Programmcodes; denn die Security Policy besteht aus einer Menge von Access Permissions.

Eine konkrete Implementierung erbt von der Klasse `Policy` und bestimmt, wie die Access Permissions konfiguriert werden können und aus welcher Quelle sie gelesen werden, beispielsweise aus einer Datei, einer Datenbanktabelle oder durch Aufruf eines Verteilungsdienstes, der netzwerkweit abgestimmte Policies verteilt. Welche konkrete Klasse tatsächlich benutzt wird, kann in der Konfigurationsdatei `java.security` bestimmt werden.

Die Standardimplementierung `PolicyFile` liest die Konfiguration aus einem oder mehreren Policy Files, deren Dateinamen ebenfalls in `java.security` festgelegt werden; mit der Standardeinstellung wird eine systemweite Datei in einem Unterverzeichnis der Java-Laufzeitumgebung und eine benutzerspezifische in dessen Home-Verzeichnis erwartet. Zudem kann eine alternative oder zusätzliche anwendungsspezifische Datei als Kommandozeilenparameter beim Programmstart angegeben werden [Sun98e]. Ein Policy File besteht aus Einträgen wie diesem:

```
grant signedBy "ls10adm,irbadm", codeBase "http://*.cs.uni-dortmund.de/" {
    permission java.io.FilePermission "c:/tmp/*", "read,write,delete";
    permission java.net.SocketPermission "*.uni-dortmund.de",
        "connect,resolve";
};
```

Abbildung 2-1: Beispiel eines Eintrags in einem Policy File der Standardimplementierung

Mit diesem Eintrag wird Programmcode, der von Rechnern aus dem Fachbereich Informatik der Universität Dortmund stammt und von `ls10adm` und `irbadm` signiert wurde, mit den angegebenen Rechten zum Lesen, Schreiben und Löschen von Dateien im Verzeichnis `c:/tmp` sowie zum Auffinden von und Verbinden mit beliebigen Rechnern aus der Domain `uni-dortmund.de` ausgestattet.

Zur Unterstützung bei der Definition einer Security Policy wird mit dem JDK 1.2 das Werkzeug `policytool` mit einer simplen graphischen Oberfläche ausgeliefert, womit die Policy Files der Standardimplementierung bearbeitet werden können [Sun98f]. Abbildung 2-2 zeigt das Hauptfenster nach dem Öffnen eines Policy File, worin neben anderen der Eintrag aus Abbildung 2-1 zu sehen ist. Das Bearbeitungsfenster für diesen Eintrag in Abbildung 2-3 listet die vergebenen Access Permissions auf, die wiederum einzeln wie in Abbildung 2-4 editiert werden können. Vorteil dieses Werkzeugs ist lediglich, daß die Standard-Permissions und deren vordefinierte Parameter aus Listen ausgewählt werden können.

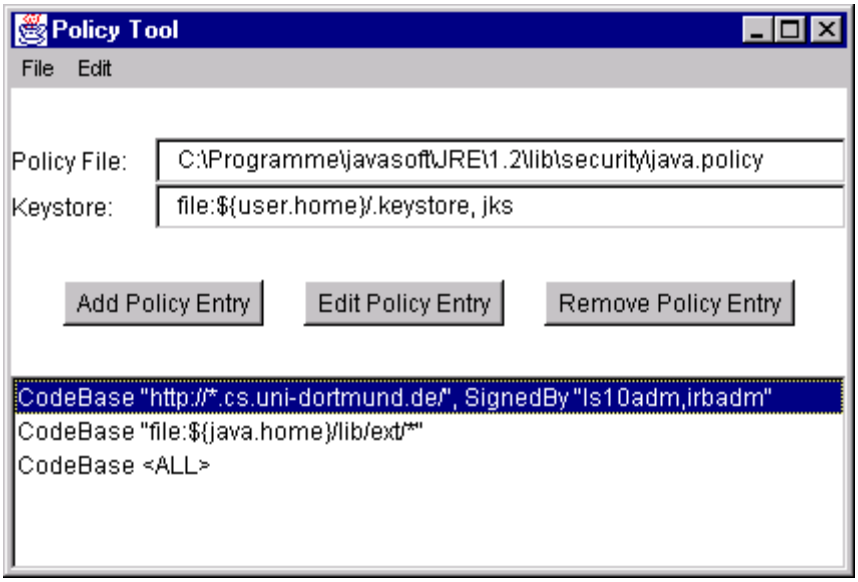

Abbildung 2-2: Hauptfenster des Policy Tools

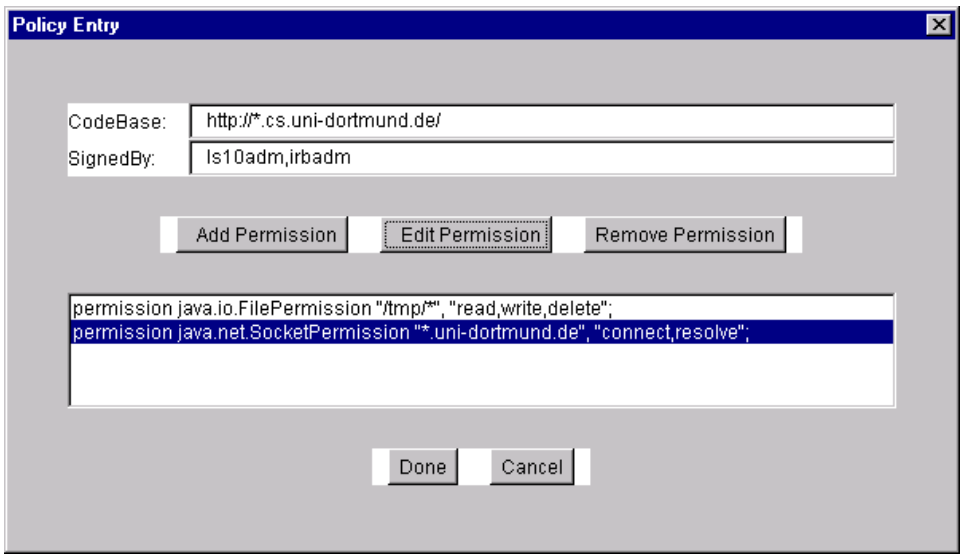

Abbildung 2-3: Bearbeiten eines Policy-Eintrags im Policy Tool

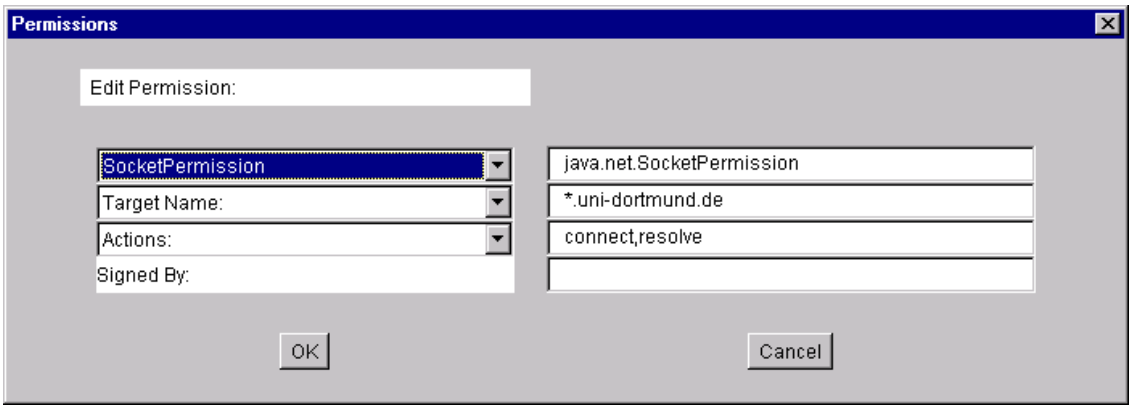

Abbildung 2-4: Bearbeiten einer Permission im Policy Tool

2.2.2.1.1 Access Permissions

Zur Repräsentation der in einer Policy enthaltenen Access Permissions als Java-Objekte steht eine Hierarchie verschiedener Permission-Klassen zur Verfügung, deren Wurzel die →abstrakte Klasse `java.security.Permission` ist [Sun98d]. Die in Abbildung 2-1 verwendeten `java.net.SocketPermission` und `java.io.FilePermission` sind Beispiele für konkrete Permissions aus dieser Hierarchie.

Beim Lesen der Policy werden anhand der implementierungsspezifischen Definition die Permission-Objekte instanziiert. Die Definition enthält dazu für jede Permission deren Klassennamen sowie zwei Parameter, die dem Konstruktor der Permission-Klasse übergeben werden: der erste dient zur genaueren Angabe der Ressource und der zweite spezifiziert Aktionen, die damit durchgeführt werden dürfen. Als Beispiel sei hier eine `java.io.FilePermission` mit den Parametern `c:/tmp/*` und `read,write,delete` genannt.

Jede konkrete Permission-Klasse muß also einen Konstruktor mit zwei Parametern enthalten, da das Policy-Objekt die Permissions nur anhand dieser Informationen erzeugen kann. In der Klasse `Permission` werden die beiden wichtigsten abstrakten Methoden als Schnittstelle definiert, die von jeder konkreten Permission implementiert werden müssen:

- `public abstract boolean equals(Object obj)` stellt die Gleichheit zweier Permissions fest. Da die konkreten Parametrisierungen einer Permission-Klasse individuell interpretiert werden können, kann die Übereinstimmung nicht durch die abstrakte Klasse `Permission` entschieden werden. Diese Methode wird dafür genutzt, Duplikate aus Mengen von Permissions zu entfernen.
- `public abstract boolean implies(Permission permission)` dient zur Feststellung, ob eine Zugriffserlaubnis zwangsläufig eine andere zur Folge hat. Beispielsweise folgt aus einer Leseberechtigung für `c:/tmp/*` auch die für `c:/tmp/file1.txt`. Auch dieser Vergleich kann nur von der konkreten Implementierung durchgeführt werden.

Die Methode `implies` ist die Grundlage zur Überprüfung der Zugriffsberechtigung bei einem Zugriffsversuch: Programmcode, der sicherheitsrelevante Aktionen ausführt, erzeugt zunächst ein Permission-Objekt mit genauer Angabe der Ressource und der auszuführenden Aktion; dieses Objekt wird dann mittels `implies` mit allen Permissions verglichen, die der aufrufende Code besitzt. Ergibt einer der Aufrufe `true`, so wird die Aktion durchgeführt, andernfalls mit einer Exception abgebrochen.

Als Beispiel wird bei Benutzung des `java.io.FileInputStream` eine `java.io.FilePermission` mit den Parametern `c:/tmp/file1.txt` und `read` erzeugt und geprüft, bevor tatsächlich auf diese Datei zugegriffen wird. Besitzt der aufrufende Code beispielsweise eine `java.io.FilePermission` mit `c:/tmp/*` und `read,write,delete`, so wird von deren `implies`-Methode festgestellt, daß die fragliche Berechtigung darin enthalten ist.

Durch diese Ausnutzung der Objektorientierung mit einer aktiven Prüfung von Zugriffsrechten durch deren repräsentierende Objekte wird eine Trennung von Konfiguration der Sicherheitspolitik und deren Durchsetzung erreicht; die Durchsetzung wird durch Security Manager und Access Controller geleistet, was in Abschnitt 2.2.2.2 beschrieben wird.

Vorteil ist eine unabhängige und freie Definition von Permissions; diese können selbst bestimmen, welche Konfigurationsmöglichkeiten sie bieten, wie sie beispielsweise auf Wildcards reagieren, und wie sie sich zu anderen Permissions desselben Typs verhalten. Zudem ist neben der Vergabe von Zugriffsrechten anhand einer statischen Definition auch eine dynamische Berücksichtigung von Zuständen zur Laufzeit denkbar.

Zudem können ohne Änderungen an bestehenden Klassen neue Typen von Zugriffsberechtigungen eingeführt werden, indem eine neue Permission-Klasse implementiert und die Vergabe der Berechtigung vom sicherheitsrelevanten Dienst vor der Ausführung geprüft wird. So können auch anwendungsdefinierte Zugriffsberechtigungen festgelegt und in den üblichen Policy Files vergeben werden [Gon99].

2.2.2.1.2 Code Source

Wie am Beispiel in Abbildung 2-1 gezeigt, wird der Code, dem Berechtigungen zugeteilt werden, an zwei Eigenschaften identifiziert, nämlich an seiner Herkunftsquelle und an einer Menge von Unterzeichnern, mit deren privaten Schlüsseln der Code signiert sein muß. Die Klasse `java.security.CodeSource` verwaltet diese Eigenschaften.

Die Quelle wird als Uniform Resource Locator (URL) angegeben und kann sowohl eine Internet-Adresse als auch eine Adresse des lokalen Dateisystems sein. Die Quellenangabe zur Spezifikation einer Code Source kann allgemeiner sein als die genaue Adresse des tatsächlichen Codes. So enthält die Angabe `http://*.cs.uni-dortmund.de/` aus dem Beispiel auch die Java-Klassen aus `http://ls10-www.cs.uni-dortmund.de/classes/bookshelf.jar`.

Zusätzlich zur passenden Quelle muß diese Datei aber auch mit den privaten Schlüsseln der geforderten Unterzeichner signiert worden sein. So müssen zumindest `ls10adm` und `irbadm` die genannte JAR-Datei signieren, um die Rechte aus Abbildung 2-1 den darin enthaltenen Klassen zu übertragen. Die öffentlichen Schlüssel zu diesen Unterzeichnernamen werden in einem *Keystore* erwartet, der in 2.2.2.1.3 beschrieben wird.

Zur Prüfung, ob eine Code Source eine andere einschließt, enthält die Klasse `CodeSource` die Methode `public boolean implies(CodeSource codesource)`. Analog zum Ablauf bei Permissions wird zum Feststellen der Zugehörigkeit einer Klasse zu vorhandenen Code Sources zunächst ein temporäres Code-Source-Objekt mit der genauen Quelle dieser Klasse und den gefundenen Unterzeichnern erzeugt und per `implies` mit den vorhandenen Code-Source-Objekten verglichen.

Eine Code Source kann auch ohne Unterzeichner festgelegt werden; dann wird bei der Zugehörigkeitsprüfung kein Unterzeichner gefordert. Ebenso kann die Quellenangabe weggelassen werden, wodurch jede beliebige Quelle zugelassen wird. Eine Code Source mit leerer Unterzeichnermenge und leerer Quellenangabe bezieht sich somit auf die Menge aller erreichbaren Java-Klassen.

2.2.2.1.3 Keystore

In einer Security Policy werden Unterzeichner durch einen Aliasnamen wie `ls10adm` definiert; die Schlüssel zu diesen Namen sind in einem Keystore gespeichert. Im Keystore können zwei verschiedene Arten von Einträgen gespeichert sein; dies sind zum einen private und öffentliche Schlüssel mit einer optionalen Kette von Zertifikaten (siehe 2.1.3.3.1) und zum anderen Zertifikate von Zertifizierungsstellen, deren Korrektheit vorausgesetzt wird und die für Vertrauensentscheidungen eingesetzt werden können.

Ein Keystore bietet Methoden zum Hinzufügen und Löschen von Zertifikaten und Schlüsseln unter einem Aliasnamen sowie zum Auslesen eines Eintrags anhand des Namens und zum Herausfinden des Alias anhand eines gegebenen Schlüssels. Diese Methoden werden sowohl beim Prüfen der Signaturen von JAR-Dateien benutzt als auch von den Werkzeugen `keytool`

und `jarsigner`, die zur Verwaltung des Keystore-Inhalts bzw. zum Signieren der JAR-Dateien dienen; diese Werkzeuge müssen über die Kommandozeile bedient werden [Sun98f].

Die Methoden können über die Klasse `java.security.KeyStore` aufgerufen werden, deren konkrete Implementierung von einem konfigurierbaren *Cryptographic Service Provider* geliefert wird, der in der Konfigurationsdatei `java.security` festgelegt werden kann. Dieser Provider ist Teil der *Java Cryptography Architecture*, die in Abschnitt 2.2.3.1 vorgestellt wird. Je nach Provider können verschiedene Arten von Keystores zur Verfügung stehen, die dann von der Java-Laufzeitumgebung ebenso wie von `keytool`, `jarsigner` und `policytool` genutzt werden.

Der zum Prüfen der Permissions verwendete Keystore wird mit einem Eintrag im Policy File mit URL und Typ angegeben. Der in Abbildung 2-2 sichtbaren Angabe entspricht beispielsweise dieser Eintrag: `keystore "file:${user.home}/.keystore", "jks"`. Der angegebene Typ muß von einem der eingestellten Cryptographic Service Provider zur Verfügung gestellt werden und ist selbst für das Laden des angegebenen Keystore verantwortlich.

Die Sun-Implementierung des Providers enthält den JavaKeyStore (JKS), der die Schlüssel in einer Datei ablegt und X.509-Zertifikate verarbeitet [Sun98b]. Das JDK 1.2 wird mit einer JKS-Datei ausgeliefert, die bereits verschiedene Stammzertifikate der Zertifizierungsstellen VeriSign, Inc. und Thawte Consulting cc enthält.

2.2.2.2 Security Manager

Der Security Manager ist die zentrale Stelle zur Durchsetzung der Sicherheitspolitik; er entscheidet darüber, welche Zugriffe auf Ressourcen durchgeführt werden dürfen. Dazu enthält er Methoden zur Überprüfung aller geschützten Zugriffe, beispielsweise `public void checkRead(String file)` für das Lesen einer Datei mit einem bestimmten Namen.

Die Klassen der Java-Klassenbibliothek, die solche Ressourcenzugriffe als Dienst anbieten, wie beispielsweise `java.io.FileInputStream`, rufen die Check-Methoden auf, bevor sie ihren Dienst ausführen. Die Check-Methode prüft, ob der Zugriff erlaubt ist und kehrt bei positiver Entscheidung zurück, so daß die Programmausführung fortgesetzt und der Zugriff durchgeführt wird; bei negativer Entscheidung wird eine Exception geworfen, was die Ausführung der Dienstmethode abbricht und letztendlich zu einem Programmabbruch führt, falls die Exception nicht an anderer Stelle gefangen wird.

Aus Gründen der Kompatibilität mit den älteren Java-Versionen ist bei der Ausführung einer Anwendung in der Java Virtual Machine zunächst kein Security Manager vorhanden; dieser muß entweder von der Anwendung installiert oder in der Kommandozeile angegeben werden. Damit ist es möglich, andere als den von Sun vorgegebenen Security Manager zu benutzen, um Entscheidungen über die Vergabe von Zugriffsrechten auf andere Weise zu treffen [Sun98a].

Im JDK 1.0 implementierte der mitgelieferte Security Manager das Sandbox-Prinzip, und mit JDK 1.1 kam die generelle Freigabe von Ressourcen für Signed Applets hinzu. Eine abweichende Sicherheitspolitik konnte in dieser Version noch nicht durch eine flexible Security Policy, sondern nur durch Programmierung eines anderen Security Managers erreicht werden; die Sicherheitspolitik und ihre Durchsetzung waren also darin gekoppelt [KNN98].

Nachteil dieser Kopplung ist die nur mit Aufwand und negativen Folgen mögliche Erweiterung der verfügbaren Ressourcen, für die neue Check-Methoden eingeführt werden mußten, was den Security Manager mit der Zeit unnötig komplex werden läßt. Zum einen war die erneute Programmierung eines komplizierten und daher fehleranfälligen Algorithmus zur Entscheidung über die Vergabe oder Verweigerung des Zugriffs nötig, und zum anderen mußten alle benutzerdefinierten Security Manager ebenfalls um diese Methode erweitert werden. Die Vergabe anwendungsspezifischer Zugriffsrechte war zudem nur mit einem darauf abgestimmten Security Manager möglich, was bei einem Einsatz mehrerer Anwendungsteile, die jeweils einen eigenen Security Manager voraussetzen, erneut Aufwand verursacht.

Im JDK 1.2 gibt es zusätzlich zu den einzelnen Check-Methoden für Ressourcenzugriffe die allgemeine Methode `public void checkPermission(Permission perm)`, die mit beliebigen, auch anwendungsdefinierten Permission-Objekten aufgerufen werden kann. Ist die übergebene Permission in den Rechten des aufrufenden Codes enthalten, so kehrt die Methode zurück, ansonsten wirft sie eine Exception. Somit wird die Sicherheitspolitik nicht mehr von der Implementierung des Security Managers bestimmt, sondern ist als Security Policy vom Benutzer oder Systemadministrator konfigurierbar und damit von der Durchsetzung entkoppelt.

Der mitgelieferte `java.lang.SecurityManager` delegiert die Prüfung der Access Permissions in dieser Methode an den Access Controller, der in Abschnitt 2.2.2.2.2 vorgestellt wird. Alle anderen Check-Methoden des Security Managers erzeugen ein passendes Permission-Objekt und lassen dies von der Methode `checkPermission` und damit letztendlich ebenfalls vom Access Controller prüfen.

Der damit eigentlich überflüssige Security Manager wurde dennoch beibehalten, um gänzlich andere Sicherheitsmodelle integrieren zu können, die nicht auf den Zugriffskontrollentscheidungen des Access Controllers basieren sollen. Laut [Gon99] können dynamische Modelle oder Multilevel Security aus 2.1.2 mit einem speziellen Security Manager unterstützt werden.

2.2.2.2.1 Protection Domain

Um die an verschiedene Klassen vergebenen Rechte prüfen zu können, muß eine Zuordnung zwischen einer Klasse und ihren Rechten vorgenommen werden. Diese Zuordnung geschieht nicht direkt, sondern indirekt über *Protection Domains:* Eine Protection Domain erhält eine Menge von Rechten; jede Klasse ist genau einer Domain zugeordnet und besitzt damit deren Rechte. Zu welcher Protection Domain eine Klasse gehört, wird beim Laden vom Class Loader anhand der Code Source festgestellt; existiert zu einer Code Source noch keine Protection Domain, so wird diese erzeugt.

Auch in `java.security.ProtectionDomain` existiert eine Methode `public boolean implies(Permission permission)`, mit der festgestellt wird, ob die Rechte der Protection Domain die angegebene Access Permission enthalten. Vorteil der Einführung von Protection Domains gegenüber der direkten Klasse-Rechte-Beziehung ist die bessere Erweiterbarkeit ohne Veränderung der Klasse `java.lang.Class` [Gon99].

2.2.2.2.2 Access Controller

Die Klasse `java.security.AccessController` stellt einen allgemeinen Zugriffskontrollalgorithmus zur Verfügung und bildet die Grundlage für die Standardimplementierung des Security Managers. Sie enthält lediglich statische Methoden; die wichtigste davon ist `public static void checkPermission(Permission perm)` und implementiert den genannten Algorithmus.

Die Grundidee ist, einen Zugriff genau dann zu erlauben, wenn alle Protection Domains sämtlicher Klassen, die zur Anforderung des Zugriffs geführt haben, die erforderlichen Rechte besitzen. Die beteiligten Klassen und damit deren Protection Domains werden herausgefunden, indem alle verschachtelten Methodenaufrufe anhand des →Stacks zurückverfolgt werden, was als *Stack Inspection* bezeichnet wird [WF98]. Auf dem Stack befinden sich zum Zeitpunkt der Zugriffsprüfung Einträge aller Klassen, die im aktuellen →Thread nacheinander Methoden aufgerufen haben und nun auf deren Rückgabewert warten. Jede dieser Klassen muß die nötigen Rechte zum Zugriff auf die Ressource haben, da alle gemeinsam zur Ausführung des Zugriffs beigetragen haben und dessen Rückgabewert erhalten können.

Somit können Klassen nicht zusätzliche Rechte erlangen, indem sie andere Klassen mit mehr Rechten aufrufen und damit beauftragen, auf eine Ressource zuzugreifen. Ein Sonderfall ergibt sich beim Starten eines neuen Thread, denn dieser erhält einen eigenen, leeren Stack. Damit die Methodenaufrufe, die zu diesem neuen Thread geführt haben, ebenfalls berücksichtigt werden, wird bei dessen Start eine Liste aller beteiligten Protection Domains aus dem Stack des aufrufenden Thread erzeugt und dem neuen Thread hinzugefügt. Diese Domains werden bei der Zugriffsprüfung dann zusätzlich zu denen des aktuellen Threads beachtet. Die Liste dieser Protection Domains bildet den *Access Control Context*, der als `java.security.AccessControlContext` in Java abgebildet ist [Gon98a].

2.2.2.2.3 Privileged Code

Es ist nicht in jedem Fall sinnvoll, die Rechte an alle Klassen auf dem Stack vergeben zu müssen. Als Beispiel sei eine Klasse genannt, die den Benutzer nach einem Paßwort fragt und dazu eine Datei mit Paßwörtern benötigt. Soll diese Paßwortabfrage von einer Anwendung aufgerufen werden, müßte auch die Anwendung Zugriffsrechte auf die Datei haben, was nicht erwünscht ist.

Eine Änderung dieses Verhaltens bei Zugriffsprüfungen ist mit *Privileged Code* möglich. Wird das Öffnen der Paßwortdatei als Privileged Code ausgeführt, so werden die zuvor ausgeführten Methodenaufrufe und deren Protection Domains bei der Prüfung des Stack nicht be-

achtet. Es reicht also, die Zugriffsrechte an die einzelne Klasse zu vergeben; die Anwendung benötigt diese nicht.

Im `AccessController` gibt es dazu die Methode `public static native Object doPrivileged(PrivilegedAction action)` sowie einige damit verwandte Methoden [Sun98c]. Diese Methoden markieren den aktuellen Eintrag des Stack als privileged und starten die Ausführung der in der `PrivilegedAction` angegebenen Operationen.

Die Prüfung von Berechtigungen der Protection Domains des aktuellen Access Control Context wird beim Erreichen eines solchen als privileged gekennzeichneten Eintrags abgebrochen, so daß vorherige Methodenaufrufe und die Rechte der zugehörigen Klassen nicht geprüft werden.

Die Verwendung von `doPrivileged` kann bei fehlerhaftem Einsatz ein erhebliches Sicherheitsrisiko darstellen, da auf Anforderung beliebigen Codes geschützte Ressourcenzugriffe mit den Rechten einer einzigen Klasse durchgeführt werden. In [Sun00b] und [Sun98c] werden Richtlinien und Ratschläge zum Erstellen sicherer Programme und speziell auch zur Anwendung von Privileged Code gegeben.

2.2.3 Sicherheitsdienste der Java-Klassenbibliothek und zusätzlicher Bibliotheken

Neben dem in 2.2.2 vorgestellten grundlegenden Sicherheitskonzept mit flexiblen und erweiterbaren Zugriffskontrollmechanismen stehen in der Java-Klassenbibliothek und in zusätzlichen, optionalen Bibliotheken weitere Sicherheitsdienste zur Verfügung, die in diesem Abschnitt vorgestellt werden.

Im Gegensatz zur Zugriffskontrolle kann der Anwender den Einsatz dieser Dienste jedoch nicht in einer Sicherheitspolitik festlegen; der Anbieter muß dies bei der Entwicklung von Anwendungen oder Komponenten bereits berücksichtigen und implementieren. Diese Arbeit bietet dem Anwender nun eine Konfiguration und Durchsetzung von Sicherheitsspezifikationen, die eine Verwendung dieser zusätzlichen Dienste einschließt.

2.2.3.1 Java Cryptography Architecture (JCA)

Die *Java Cryptography Architecture (JCA)* bietet kryptographische Funktionalität, die anderen Teilen des JDK und ebenso Anwendungen zur Verfügung steht. Enthalten sind beispielsweise der in 2.2.2.1.3 beschriebene Keystore und weitere Klassen zur Verarbeitung von X.509-Zertifikaten sowie Algorithmen zum Verschlüsseln und Signieren oder One-Way-Hash-Funktionen. Eng verwandt ist die *Java Cryptography Extension (JCE)*, die als separate Erweiterung zur Java-Klassenbibliothek installiert werden muß und in Abschnitt 2.2.3.2 vorgestellt wird.

JCA ist als Provider-Architektur ausgelegt, die verschiedene und mehrere Kryptographie-Implementierungen unterstützt, so daß weitere Algorithmen oder andere Implementierungen

durch einen zusätzlichen Cryptographic Service Provider (CSP) einfach hinzugefügt werden können [Sun99b]. Somit ist JCA unabhängig von konkreten Algorithmen und deren Implementierung.

JCA definiert *Application Programming Interfaces (APIs)* für verschiedene kryptographische Dienste, die bei der Programmierung einer Anwendung eingesetzt werden können. Auf der anderen Seite werden *Service Provider Interfaces (SPIs)* definiert, die für einen konkreten Algorithmus von einem Provider implementiert werden müssen. Ein Cryptographic Service Provider kann durch Eintragung in der Konfigurationsdatei `java.security` oder zur Laufzeit durch einen Methodenaufruf installiert werden und gibt der JCA Auskunft über die von ihm angebotenen Dienste und deren Namen.

Um einen kryptographischen Dienst zu benutzen, muß dessen API aufgerufen werden; JCA sucht eine passende Implementierung bei den installierten Providern. Zur genaueren Festlegung kann auch ein bestimmter Algorithmus über seinen Namen oder ein bestimmter Provider angefordert werden.

Die folgenden Dienste werden von JCA im JDK 1.2 definiert [Sun99a]:

- `MessageDigest` zum Berechnen der Hash-Werte von Daten.
- `Signature` zum Signieren von Daten und Prüfen von Signaturen.
- `KeyPairGenerator` zum Erzeugen von Paaren privater und öffentlicher Schlüssel als Subklassen von `Key` für konkrete Algorithmen.
- `KeyFactory` zum Konvertieren von Schlüsseln zwischen verschiedenen Darstellungen.
- `CertificateFactory` zum Erzeugen von Zertifikaten für öffentliche Schlüssel und von Certificate Revocation Lists (CRLs) [Sun98b].
- `KeyStore` zum Speichern von Schlüsseln und Zertifikaten.
- `AlgorithmParameters` zum Verwalten von Parametern, die für konkrete Algorithmen notwendig sind.
- `AlgorithmParameterGenerator` zum Erzeugen der Parameter für einen Algorithmus.
- `SecureRandom` zum Erzeugen von Zufallszahlen, die für kyptographische Algorithmen benötigt werden.

In JCA wird ein Dienst zur Erzeugung von Objekten mit gänzlich neuem Inhalt als Generator und die Erzeugung von Objekten aus existierenden Daten als Factory bezeichnet.

Eine Anwendung kann sich ein API-Objekt jeder dieser Dienste erzeugen lassen und diese benutzen. Dieses API-Objekt sucht in den angemeldeten Providern nach einem passenden Dienst und erzeugt ein SPI-Objekt, das von einem der Provider angeboten wird und das Service Provider Interface implementiert. Anforderungen der Anwendung werden vom API-Objekt an sein privates SPI-Objekt weitergereicht.

Mit dem JDK wird von Sun der Provider `sun.security.provider.Sun` ausgeliefert, der Klassen für den Digital Signature Algorithm (DSA), den Hash-Algorithmus MD5, den proprietären Algorithmus SHA1PRNG zur Erzeugung von Zufallszahlen, eine `CertificateFactory` für X.509-Zertifikate, den proprietären `KeyStore` JavaKeyStore (JKS) sowie die nötigen unterstützenden Dienste enthält, etwa die Schlüsselerzeugung für DSA [Sun99a]. Andere Cryptographic Service Provider können zusätzliche Algorithmen oder andere Implementierungen von Algorithmen bieten, beispielsweise auch plattformspezifische oder solche mit Hardware-Unterstützung.

2.2.3.2 Java Cryptography Extension (JCE)

Zusätzlich zur Java Cryptography Architecture (JCA) bietet die optional erhältliche *Java Cryptography Extension (JCE)* weitere kryptographische Algorithmen, die nicht von dem in Abschnitt 2.2.2 vorgestellten Java-Sicherheitskonzept vorausgesetzt werden, aber bei der Entwicklung von Anwendungen eingesetzt werden können.

Ein weiterer Grund für die Trennung von JCA und JCE waren US-Exportbeschränkungen für kryptographische Systeme, die mit großen Schlüssellängen arbeiten. Mit JCE 1.2.1 wurden daher *Jurisdiction Policy Files* eingeführt, mit denen die Länge der verarbeiteten Schlüssel eingeschränkt werden kann; durch die gelockerten Exportbestimmungen können jedoch in den meisten Ländern Jurisdiction Policy Files ohne Schlüssellängenbegrenzung eingesetzt werden.

JCE entspricht der aus JCA bekannten Architektur und definiert die folgenden Dienste, die von verschiedenen Providern angeboten werden können [Sun00d]:

- `Cipher` dient zum Ver- und Entschlüsseln von Byte-Arrays und benötigt zur Ausführung Instanzen von `Key`, `AlgorithmParameters` und `SecureRandom` als Initialisierung des konkreten Algorithmus.
- `CipherInputStream` und `CipherOutputStream` verknüpfen einen `Cipher` mit einem `Input-` bzw. `OutputStream` und können beispielsweise in Verbindung mit `FileInput-` und `FileOutputStream` oder anderen Streams eingesetzt werden.
- `KeyGenerator` erzeugt geheime Schlüssel für symmetrische Verschlüsselungsverfahren.
- `SecretKeyFactory` konvertiert symmetrische Schlüssel zwischen verschiedenen Darstellungen.
- `KeyAgreement` dient zum Erzeugen einer gemeinsamen, geheimen Information – beispielsweise eines geheimen Schlüssels – aus den öffentlichen Schlüsseln mehrerer beteiligter Parteien.
- `Mac` steht für Message Authentication Code (MAC) und dient zum Schutz der Integrität von Daten; konkrete Implementierungen können Einweg-Hash-Funktionen oder digitale Signaturen mit asymmetrischen Verschlüsselungsverfahren sein.

Wie bei JCA können auch für JCE zusätzliche Provider installiert werden, die andere Algorithmen oder Implementierungen bieten. Der mitgelieferte Provider `com.sun.crypto.provider.SunJCE` bietet die Verschlüsselungsalgorithmen DES, Triple DES und Blowfish in den Ausführungsmodi Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB) und Propagating Cipher Block Chaining (PCBC) sowie die MACs HMAC-MD5 und HMAC-SHA1, ebenso wie die `Keystore`-Implementierung JCEKS mit stärkerem Schutz für private Schlüssel als im JKS [Sun00d]. Zusätzlich enthält JCE die Klasse `SealedObject`, die in Abschnitt 2.2.3.4 vorgestellt wird.

2.2.3.3 Java Secure Socket Extension (JSSE)

Um bei Netzwerkkommunikation über Sockets das von Netscape entwickelte Protokoll Secure Sockets Layer (SSL) oder Transport Layer Security (TLS) einsetzen zu können, kann die *Java Secure Socket Extension (JSSE)* installiert werden.

Statt eines normalen `Socket` kann dann ein `SSLSocket` eingesetzt werden, von dem alle Details dieser Protokolle implementiert werden und das von einer `SSLSocketFactory` vor der Verbindung erzeugt und initialisiert wird; eine dabei erzeugte `SSLSession` speichert Parameter, die beim ersten Verbindungsaufbau ausgetauscht wurden, und kann bei neuen `SSLSockets` für weitere Verbindungen mit demselben Kommunikationspartner wieder eingesetzt werden.

Auch JSSE 1.0.2 folgt der Provider-Architektur, so daß weitere Protokolle und Algorithmen zusätzlich installiert werden können. Der Provider `com.sun.net.ssl.internal.ssl.Provider` enthält Implementierungen für SSL 3.0 und TLS 1.0 inklusive `Cipher`-Kombinationen mit gleichzeitiger Authentifizierung, Key Agreement, Verschlüsselung und Integritätssicherung sowie eine Schlüsselverwaltung für X.509-Schlüssel, die aus einem Keystore geladen werden können [Sun00c].

2.2.3.4 Signed Object, Sealed Object, Guarded Object

Neben der Nutzung verschlüsselter Kanäle wie bei JSSE kann es auch sinnvoll sein, die Verschlüsselung oder Signierung auf einzelne Objekte anzuwenden, um sie unabhängig von ihrem Aufenthaltsort gegen unberechtigtes Lesen bzw. Benutzen oder gegen Manipulationen zu schützen.

Für solchen objektorientierten Schutz gibt es im JDK 1.2 und JCE 1.2.1 drei Klassen [GS98]:

- `SignedObject`. Ein anderes Objekt kann in einem Signed Object aus dem JDK verpackt werden und darin weitergereicht werden; das Signed Object erzeugt beim Einpacken eine Signatur des übergebenen Objektes und speichert diese zusätzlich zu diesem. Später kann das Signed Object dann feststellen, ob die transportierten Daten des verpackten Objekts noch zur Signatur passen, oder ob sie verändert wurden.

Voraussetzung ist die Serialisierbarkeit des zu schützenden Objekts, es muß also das Interface `Serializable` implementieren und damit in einen binären Datenstrom umzuwandeln sein. Bei der Erstellung eines `SignedObject` muß diesem ein `Signature`-Service-Objekt (siehe 2.2.3.1) und ein privater Schlüssel übergeben werden, um die Signatur erzeugen zu können. Bei einer späteren Verifikation muß der passende öffentliche Schlüssel und ebenfalls wieder ein `Signature`-Service zur Verfügung stehen. Um das Objekt zurückzuerhalten, sind allerdings keine zusätzlichen Angaben erforderlich.

- `SealedObject`. Die Vertraulichkeit eines Objekts kann durch ein Sealed Object geschützt werden, welches das übergebene Objekt serialisiert und mit einem zusätzlich benötigten `Cipher` (siehe 2.2.3.2) verschlüsselt. Das Objekt kann nur dann wiederhergestellt werden, wenn ein entsprechender Cipher zum Entschlüsseln zur Verfügung steht, der mit dem nötigen Schlüssel ausgestattet wurde. Die Klasse `SealedObject` ist in der JCE enthalten, aber als konkrete Klasse implementiert und nicht durch andere Provider ersetzbar, da der eingesetzte Cipher ausgetauscht werden kann.

- `GuardedObject`. Zugriffskontrolle auf Objektebene kann mit einem Guarded Object aus dem JDK implementiert werden; darin kann ein anderes Objekt zusammen mit einem `Guard` verpackt werden und wird nur dann wieder freigegeben, wenn der angegebene Guard zustimmt. Auf diese Weise kann eine Zugriffskontrolle für die Benutzung eines Objektes ohne dessen Veränderung erreicht werden.

  Beispielsweise könnten Objekte, die von einem Server für Clients zur Verfügung gestellt werden, in einem Guarded Object ausgeliefert werden. Die Zugriffskontrolle kann dann auf dem Client ausgeführt werden, so daß der Access Control Context (siehe 2.2.2.2.2) des Clients und nicht der des Server-Threads zur Feststellung einer Zugriffsberechtigung berücksichtigt wird.

Die Klassen der verpackten Objekte können mit dieser Lösung also ohne eigene Sicherheitsmaßnahmen implementiert werden, da diese nachträglich hinzugefügt werden. Programmteile, die mit solchen Objekten umgehen, müssen allerdings explizit mit Benutzung von `SignedObject`, `SealedObject` oder `GuardedObject` arbeiten.

2.2.3.5 Java Authentication and Authorization Service (JAAS)

Die in Java vorhandenen Zugriffskontrollmechanismen treffen ihre Entscheidung aufgrund von Herkunftsquelle und Unterzeichnern der ausgeführten Klassen, also anhand deren Code Source. Dieses eher unübliche Vorgehen entspricht der Zielsetzung, eine sichere Ausführungsplattform auch für unbekannten Mobile Code zu sein, der aus nicht vertrauenswürdigen Quellen im Internet stammen kann.

Bei der Entwicklung von Mehrbenutzeranwendungen ist es jedoch oft erwünscht, die Zugriffsentscheidungen bei kritischen Programmteilen oder Ressourcenzugriffen auch danach treffen zu können, von wem die Anwendung ausgeführt wird. Der *Java Authentication and Authorization Service (JAAS)* ist als zusätzliche Bibliothek für das JDK 1.3 erhältlich und erweitert die bekannten Zugriffskontrollen um diese Möglichkeit [LGK99].

Dazu ist es zunächst erforderlich, den Benutzer zu authentifizieren, also seine Identität eindeutig festzustellen. Dies kann durch verschiedenste Methoden erreicht werden, wie in 2.1.4 angedeutet wurde; aus diesem Grund ist die Authentifizierung im JAAS 1.0 so gestaltet, daß unterschiedliche Authentifizierungsverfahren ohne Veränderung der Anwendung genutzt werden können. Ein Login, bei dem lediglich nach einem Benutzernamen und Paßwort gefragt wird, kann mit einem veränderten Konfigurationseintrag durch eine Überprüfung des Fingerabdrucks ersetzt werden.

Nachdem der Benutzer feststeht, kann dessen Autorisierung zur Ausführung geschützter Programmteile geprüft werden. Diese Zugriffskontrolle wird beim JAAS in Zusammenarbeit mit den bestehenden Mechanismen durchgeführt, so daß sowohl Code Source als auch Benutzer mit den üblichen Access Permissions ausgestattet werden können. Die Vergabe der Permissions für Benutzer erfolgt dabei getrennt von der gewohnten Security Policy; beides zusammen bildet aber gemeinsam die tatsächlich durchgesetzte Sicherheitspolitik.

In der JAAS-Implementierung dient die Klasse `Subject` zur Darstellung des Nutzers einer Anwendung oder eines Dienstes. Dabei kann es sich um einen menschlichen Benutzer oder um einen aufrufenden Dienst handeln. Ein Subject kann anhand eines oder mehrerer Bezeichner identifiziert werden; ein solcher Bezeichner wird `Principal` genannt. Zusätzlich zu den Principals enthält ein Subject eine Menge von Credentials, wobei es sich um dessen Schlüssel oder um Kerberos-Tickets handeln kann. Im JAAS kann einem Subject jedes beliebige Java-Objekt als Credential übergeben werden, und zwar entweder als öffentliches oder als privates Credential, die getrennt verwaltet werden.

Um Aktionen im Namen eines Subjects auszuführen, müssen diese wie bei Privileged Actions als anonyme oder benannte Subklasse von `java.security.PrivilegedAction` implementiert werden und zusammen mit dem Subject an die Methode `public static Object doAs(final Subject subject, final PrivilegedAction action)` übergeben werden. `doAs` erweitert vor der Ausführung von `action` zunächst den Access Control Context des aktuellen Threads mit den Rechten des Subjects; dazu nutzt es die mit JDK 1.3 hinzugekommene Möglichkeit zur Veränderung der Protection Domains bei einem Aufruf von `doPrivileged` aus 2.2.2.2.3. Bei der späteren Zugriffsprüfung werden somit auch die Access Permissions dieses assoziierten Subjects geprüft.

Die Permissions werden in der Klasse `javax.security.auth.Policy` des JAAS verwaltet und können mit der Standardimplementierung `com.sun.security.auth.PolicyFile` ähnlich wie im Policy File des JDK vergeben werden, hier allerdings auch an `Principals`:

```
grant Principal com.sun.security.auth.NTUserPrincipal "joe" {
  permission java.io.FilePermission "c:/home/joe/*", "read,write,delete";
};
```

Abbildung 2-5: Beispiel eines Eintrags in einem Policy File der Standardimplementierung des JAAS

Wahlweise können hier zusätzlich `codeBase` und `signedBy` gefordert oder mehrere Principals angegeben werden, die alle dem ausführenden Subject zugeordnet sein müssen. Zur Vergabe der Rechte an Principals können nicht die normalen Policy Files genutzt werden, da de-

ren Implementierung diese Möglichkeit nicht vorsieht; daher wurde eine zweite Implementierung einer Klasse `PolicyFile` geschaffen, die unabhängig von der bestehenden ist.

Die Authentifizierung eines Subjects wird von der Klasse `LoginContext` gesteuert; diese sucht nach einem konfigurierten `LoginModule`, das die Identität des Subjects feststellt und ihm die entsprechenden Principals und Credentials zuweist. Das konkrete Login Module `com.sun.security.auth.module.NTLoginModule` stellt den Benutzernamen des in Windows NT angemeldeten Benutzers fest. Neben weiteren mitgelieferten Modulen können auch eigene implementiert und durch Änderung der Konfiguration eingesetzt werden. Es ist auch möglich, mehrere Module zu konfigurieren, wobei wahlweise ein einzelnes Login ausreicht oder mehrere gefordert werden können [Sun00a].

2.2.4 Fazit und Bewertung

Die Java-Umgebung bietet aufgrund der folgenden Eigenschaften gute Voraussetzungen für die Ausführung sicherheitsrelevanter Anwendungen:

- Java bietet durch die Sprachdefinition und Bytecode Verification *Typsicherheit*, die beispielsweise in C nicht gegeben ist [Gol98].
- Zusammen mit diesen grundlegenden Mechanismen ermöglicht die Java-Zugriffskontrolle durch Beachtung von Zugriffsrechten aus einer konfigurierbaren Sicherheitspolitik einen *Schutz vor böswilligem Code*.
- Zudem stehen in der Java-Klassenbibliothek und zusätzlichen Bibliotheken umfangreiche, *in Anwendungen einsetzbare Sicherheitsdienste* zur Verfügung.

Diesen Vorteilen stehen jedoch auch eine Reihe von Nachteilen gegenüber:

- Auch Java kann nicht uneingeschränkt als sicheres System bezeichnet werden. Zahlreiche Fehler, die ein Umgehen der Sicherheitsmechanismen ermöglichen könnten, aber auch konzeptionelle Schwächen, werden in der Literatur erwähnt, beispielsweise in [Wal99, Gon99, MF99, KNN98, WBD97, Pfl97]. Einige der Sicherheitsprobleme konnten bereits behoben werden, andere sind noch ungelöst, und sehr wahrscheinlich werden in Zukunft weitere entdeckt.
- Noch ist keinerlei Schutz gegen *Denial-of-Service-Attacken* konzipiert oder implementiert; er ist aber in zukünftigen Java-Versionen zu erwarten [Gon98a].
- Die *Definition der Sicherheitspolitik* ist trotz rudimentärer Werkzeugunterstützung unübersichtlich, aufwendig und ist weder für Laien noch unternehmens- oder netzwerkweit möglich. Zur Definition einer übergreifenden Sicherheitspolitik statt individueller Konfiguration jedes Arbeitsplatzes sowie leicht verständlicher und übersichtlicher Darstellung von Einstellungen und deren Auswirkungen bieten sich weitere Untersuchungen an.
- Die Reichweite einer ohne zusätzlichen Programmieraufwand konfigurierbaren Sicherheitspolitik ist auf den Schutz von Ressourcen durch *Zugriffskontrolle* beschränkt. Weite-

re Schutzmaßnahmen wie Verschlüsselung von Dateien oder Netzwerkübertragungen sowie Zugriffskontrolle für Benutzer stehen zwar zur Verfügung; sie können allerdings nicht ausschließlich in einer Sicherheitspolitik definiert werden, sondern müssen vom Anwendungs- bzw. Komponentenentwickler explizit benutzt werden.

Gerade bei einer allgemein einsetzbaren Komponente ist es schwer oder zumindest sehr aufwendig, mehrere unterschiedliche Sicherheitsanforderungen bei ihrem Einsatz bereits im Voraus bei der Entwicklung zu berücksichtigen. Eine eventuell notwendige Verschlüsselung wäre durch Parametrisierung zwar zu berücksichtigen; die Voraussetzungen für deren Aktivierung oder Ausprägung sind jedoch beliebig komplex vorstellbar, was eine allgemeingültige Implementierung in einer Komponente zusätzlich erschweren oder unmöglich machen kann. Zudem ist der Einsatz zusätzlicher, eventuell anwendungs- oder unternehmensspezifischer Sicherheitsmerkmale von Anbietern allgemeiner Komponenten nicht vorhersehbar, was eine nachträgliche Ausstattung beliebiger Anwendungen oder Komponenten mit zusätzlicher Funktionalität erstrebenswert macht.

Die Ergebnisse dieser Arbeit ermöglichen die Konfiguration und Durchsetzung von erweiterten Sicherheitsspezifikationen, die nicht nur auf Zugriffskontrolle mit Ja/Nein-Entscheidungen basieren, sondern zusätzlich die vorgestellten Sicherheitsmechanismen wie Benutzerauthentifizierung und -autorisierung oder kryptographische Algorithmen einbeziehen können, ohne deren Verwendung in den ausgeführten Komponenten vorauszusetzen.

3 Anforderungen

Wie in Abschnitt 2.2.2 vorgestellt, kann in Java eine Sicherheitspolitik für Zugriffskontrolle konfiguriert werden, die zur Laufzeit von der →Java-Laufzeitumgebung durchgesetzt wird. Zusätzliche Sicherheitsmechanismen, beispielsweise die in 2.2.3 vorgestellten Bibliotheken, können jedoch nicht allein durch Definition in einer Sicherheitspolitik aktiviert werden, sondern müssen von Anwendungs- und Komponentenentwicklern eingesetzt werden.

Ein Anwender, der solche Mechanismen zum Schutz seiner Daten einsetzen möchte, ist auf Software angewiesen, bei deren Entwicklung seine Sicherheitsziele berücksichtigt wurden. Da kryptographische Algorithmen von den Anwendungen initialisiert werden müssen, benötigen diese auch Zugriff auf vertrauliche Informationen wie private oder geheime Schlüssel. Die erreichte Sicherheit ist also von eventuellen Lücken abhängig, die bei der Implementierung dieser Mechanismen in sämtlichen eingesetzten Anwendungen und Komponenten entstehen können.

Daher soll in dieser Arbeit eine Lösung entwickelt werden, mit der die Implementierung und insbesondere der Einsatz von Sicherheitsmechanismen vollständig von der Anwendungsentwicklung entkoppelt werden kann. Die Anwendungen sollen also unabhängig erstellt und nachträglich um Sicherheitsfunktionalität erweitert werden können. Eine solche Lösung ermöglicht eine klare Rollenaufteilung zwischen Anwendungsentwicklern, die sich auf die benötigte Fachlichkeit konzentrieren, und Sicherheitsexperten, die nachträglich Sicherheitsmechanismen für die Ausführung der entwickelten Software konfigurieren können, wie es für Zugriffskontrolle bereits möglich ist. Zudem wird die Kenntnis der Anwendungen über kryptographische Schlüssel überflüssig, und die möglichen Fehlerquellen bei der Verwendung solcher Informationen werden auf die ausschließlich für Sicherheit zuständigen Zusatzbibliotheken eingeschränkt.

Der Prototyp einer solchen Sicherheitsbibliothek soll im Rahmen dieser Arbeit entworfen und implementiert werden. Dabei liegt der Schwerpunkt auf einer einfachen und flexiblen Erweiterbarkeit der prototypisch entwickelten Funktionalität, um einerseits zusätzlich allgemein nutzbare Mechanismen in die bestehende Lösung integrieren zu können und andererseits auch anwendungs- oder umgebungsspezifische Anforderungen berücksichtigen zu können.

Derartige neue Anforderungen können sowohl einen Austausch von Sicherheitsmechanismen an bereits berücksichtigten Stellen zur Folge haben als auch eine Veränderung neuer Teile der Java-Klassenbibliothek. Daher ist zur nachträglichen Erweiterung ihrer Funktionalität eine

flexible Anpassung bestehender Bibliotheksklassen notwendig. Der Lösungsansatz des Wrapping zur Modifikation bestehender Klassen wurde in Abschnitt 1.2 bereits vorgestellt und soll als Grundlage der Sicherheitsbibliothek entwickelt und eingesetzt werden.

Die folgenden Abschnitte enthalten konkrete, an den Prototypen gestellte Anforderungen.

3.1 Funktionale Anforderungen

Die Sicherheitsbibliothek soll einen Schutz von Informationen bieten, die Anwendungen bei Ressourcenzugriffen verlassen. Dateisystemzugriffe sowie Netzwerkverbindungen über Sokkets und →Remote Method Invocation (RMI), die mit den dazu vorgesehenen Klassen der Java-Klassenbibliothek ausgeführt werden, sollen daher nicht nur verboten oder erlaubt, sondern auch durch zuschaltbare Mechanismen gesichert werden können.

Die dazu notwendige Grundfunktionalität kryptographischer Algorithmen für Verschlüsselung und →Message Authentication Codes steht in standardisierter Form bereits in den Java-Zusatzbibliotheken zur Verfügung, die in 2.2.3 vorgestellt wurden. Diese sollen mit der Sicherheitsbibliothek dieser Arbeit eingesetzt werden können, wobei die Vorteile der genannten Bibliotheken, beispielsweise Austauschbarkeit von Algorithmen, erhalten bleiben und die sonst von Anwendungen zu leistende Parameter- und Schlüsselverwaltung vollständig übernommen werden sollen.

Die Bibliothek soll auf Server-Seite ohne Eingriff durch Benutzer ausgeführt werden können, während auf Clients auch eine Benutzerauthentifizierung möglich sein muß. Wie die Algorithmen ist auch die Art der Authentifizierung durch Einsatz der Standardbibliotheken austauschbar. Statt leicht zu erratender Paßwörter können damit auch Smartcards und andere Hardware-basierte Mechanismen oder biometrische Verfahren [Wec90] eingesetzt werden, sind aber nicht Bestandteil dieser Bibliothek.

Die anzuwendenden Algorithmen und die dazu notwendigen Parameter sollen in einer Sicherheitspolitik konfiguriert werden können, die von der Bibliothek in Verbindung mit Wrapping durchgesetzt wird. Die Entscheidung über den Einsatz der Zusatzfunktionalität und dabei verwendete Parameter oder Schlüssel sollen anhand ähnlicher Bedingungen getroffen werden können wie bei der aus 2.2.2.1 und 2.2.3.5 bekannten Zugriffskontrolle, also anhand des gewünschten Ressourcenzugriffs, anhand der ausgeführten Programme oder Komponenten sowie anhand eines angemeldeten Benutzers.

3.2 Nichtfunktionale Anforderungen

Die grundsätzliche Anforderung besteht darin, die neue Funktionalität ohne Änderung an bestehenden Anwendungen in diese integrieren zu können. Die Anwendung und der Benutzer sollen dabei möglichst nicht tangiert werden.

Wo es möglich und sinnvoll ist, soll die neue Bibliothek Schnittstellen und Klassen bereits vorhandener Standardbibliotheken einsetzen, um eine Kompatibilität mit diesen Bereichen herzustellen und damit auch andere zukünftige Implementierungen nutzen zu können, die mit dieser gemeinsamen Grundlage entwickelt werden.

Neben der Implementierung soll sich auch der Entwurf an den vorgestellten Konzepten orientieren, wenn dies nicht im Widerspruch zu gewünschten Eigenschaften steht. Damit soll einerseits auf dem erreichten Stand aufgebaut werden und andererseits die Neuentwicklung in die bestehende Sicherheitslandschaft passend eingefügt werden.

Aufgrund der Zielsetzung dieser Arbeit müssen neue Ansätze entwickelt und untersucht werden, so daß an den experimentellen Prototypen aus Aufwandsgründen Anforderungen wie die folgenden nicht gestellt werden: großer Funktionsumfang, Messung der Performanz, Untersuchung auf Sicherheitslücken oder eine komfortable Benutzeroberfläche zur Definition der Sicherheitspolitik.

Die entwickelte Lösung soll aber im Rahmen der Arbeit durch einen Einsatz innerhalb einer bereits bestehenden Client-Server-Anwendung [BCC99] validiert werden, um die Funktionsfähigkeit der Konfiguration und Durchsetzung von Sicherheitsspezifikationen zu überprüfen.

4 Erweiterung der Java Virtual Machine um Wrapping

Die in Abschnitt 1.2.1 vorgestellte Grundidee erfordert Veränderungen am Dynamic Class Loading, um Klassen zur Laufzeit durch andere zu ersetzen und so ein Wrapping von Systemklassen und damit auch eine Veränderung des Verhaltens von Teilen der Java-Klassenbibliothek zu ermöglichen. Die Erweiterung der →Java Virtual Machine um einen Wrapping-Mechanismus wird in diesem Kapitel vorgestellt und bildet den Verankerungspunkt, um Sicherheitsmechanismen ohne Veränderung der Java-Klassenbibliothek oder der Anwendungsklassen auch nachträglich in bestehende Software-Systeme zu integrieren.

In Kapitel 5 wird anschließend eine Bibliothek von Klassen vorgestellt, die zusätzlich zu verfügbaren Sicherheitsmechanismen benötigte Funktionalität sowie Methoden für das erweiterte Modell zur Definition einer Sicherheitspolitik anbietet. Abschnitt 5.4.3 befaßt sich dabei mit den im Rahmen dieser Arbeit entwickelten Wrapper-Klassen, die mit dieser Bibliothek schließlich die Durchsetzung von konfigurierten Sicherheitsspezifikationen ermöglichen, ohne diese bei der Entwicklung von Anwendungen oder Komponenten berücksichtigen zu müssen.

4.1 Vorüberlegungen

Ziel dieser Arbeit ist die Integration von Sicherheitsmechanismen in bestehende Anwendungen, ohne deren Quelltexte zu modifizieren. Dies könnte mit zunächst geringem Aufwand auch ohne einen Wrapping-Mechanismus erreicht werden, indem die Klassen der Java-Klassenbibliothek [Sun00f] so verändert werden, daß zusätzlich zur vorhandenen Funktionalität auch die gewünschten Sicherheitsalgorithmen ausgeführt werden.

Dieses Vorgehen ist jedoch in zweierlei Hinsicht inflexibel. Zum einen werden die einsetzbaren Sicherheitsmechanismen dabei vom Entwickler dieser veränderten Klassenbibliothek festgelegt. Soll später von Dritten eine Modifikation der Algorithmen oder eine Erweiterung um zusätzliche Mechanismen erfolgen, so müssen dafür erneut Änderungen an den zuvor bereits geänderten Quelltexten der Klassenbibliothek vorgenommen werden. Dies führt zu erhöhtem Aufwand im Konfigurationsmanagement, denn vor dem Einsatz eines neuen Release der Java-Laufzeitumgebung müssen zunächst die dazugehörigen Quelltexte der Klassenbibliothek von Sun veröffentlicht werden und anschließend von allen beteiligten Parteien nacheinander erneut um die von ihnen ergänzte Funktionalität erweitert werden.

Zum anderen können bei diesem Vorgehen nur solche Klassen modifiziert werden, deren Quelltexte verfügbar sind und auch verändert werden dürfen. Andere Bibliotheken oder Komponenten, die neben der standardisierten Klassenbibliothek eingesetzt werden, können also unter Umständen nicht mit zusätzlichen Sicherheitsmechanismen ausgestattet werden.

Mit dem im Rahmen dieser Arbeit entwickelten Ansatz kann dagegen jede beliebige Klasse durch einen Wrapper ersetzt werden. Dieser wird als Java-Klasse implementiert und bildet die von außen zugängliche Schnittstelle der gewrappten Klasse nach, um zur Laufzeit an deren Stelle von allen anderen Klassen benutzt werden zu können. Die Methoden und Attribute, die die Schnittstelle einer Klasse bilden, sind mit Hilfe der →Reflection API auch dann zugänglich, wenn der Quelltext nicht vorliegt. Daher ist es möglich, Wrapper für beliebige Klassen der Klassenbibliothek, zusätzlicher Bibliotheken und Komponenten oder für Klassen der eigentlichen Anwendung zu entwickeln.

Des weiteren können die eingesetzten Wrapper für jeden Programmstart definiert werden, so daß flexibel andere Algorithmen oder zusätzliche Sicherheitsmechanismen eingesetzt werden können.

Im nächsten Abschnitt wird das Wrapping und die damit verbundenen Möglichkeiten ausführlicher beschrieben. Anschließend wird in 4.3 das Konzept für die Änderungen an der Java-Laufzeitumgebung vorgestellt, die das Wrapping ermöglichen. Der danach folgende Abschnitt 4.4 enthält die konkreten Implementierungsdetails der Modifikationen.

4.2 Wrapping

Die Grundidee des Wrapping-Ansatzes besteht darin, beliebige Klassen der Klassenbibliothek oder der eingesetzten Komponenten und Anwendungen zur Laufzeit durch andere Klassen, nämlich die Wrapper-Klassen, ersetzen zu können. Dies ermöglicht die Veränderung, Kontrolle oder Protokollierung des Verhaltens eines Software-Systems, ohne dessen Quelltexte, compilierte Klassendateien oder Installation verändern zu müssen.

4.2.1 Klassifizierung von Wrappern

Ein Wrapper wird vom Rest des Systems statt der durch ihn ersetzten Klasse aufgerufen und muß somit neben dem beabsichtigten Zusatznutzen auch deren ursprüngliche Aufgaben erfüllen. Um die vorhandene Funktionalität nicht innerhalb des Wrappers vollständig neu implementieren zu müssen (*autarker Wrapper*), bietet sich in den meisten Fällen eine Nutzung der eigentlich ersetzten Klasse durch den Wrapper an.

Grundvoraussetzung dafür ist zunächst lediglich, daß die ersetzte Klasse vom Wrapper zur Laufzeit auch angesprochen werden kann. Beziehungen des Wrappers zu dieser Klasse müssen zur Laufzeit unverändert bleiben, da die Wrapper-Klasse sich selbst referenzieren würde, wenn sie auch dann durch den Wrapper ersetzt würde.

Bei der konkreten Implementierung eines Wrappers kann die Funktionalität der gewrappten Klasse auf alle in Java auch für herkömmliche Klassen verfügbaren Arten wiederverwendet werden. Abhängig von der bei der Implementierung eines Wrappers gewählten Referenzierung der ursprünglichen Klasse ergeben sich unterschiedliche Eigenschaften, die im folgenden vorgestellt werden.

4.2.1.1 Aktive Wrapper

Eine Nutzung der gewrappten Klasse für ihre ursprüngliche Aufgabe ist beispielsweise durch Instanziierung eines Objekts aus dieser Klasse möglich, wie in Abbildung 4-1 als Klassendiagramm in der Unified Modeling Language (UML) [RJB98, FS97] gezeigt. Jedes Wrapper-Objekt hält damit im Hintergrund ein Originalobjekt bereit, an das es die Methodenaufrufe weiterreichen kann, nachdem oder bevor die für den Zweck des Wrappers nötigen Zusatzschritte durchgeführt werden, beispielsweise eine Verschlüsselung oder Entschlüsselung der Daten in den Aufrufparametern. Da der Wrapper dabei aktiv die übergebenen Daten ändert, wird ein solcher im folgenden als *aktiver Wrapper* bezeichnet. Dabei ist nicht ausschlaggebend, ob die Anweisungen zur Manipulation der Parameter in der Wrapper-Klasse oder einer zusätzlichen Klasse implementiert sind.

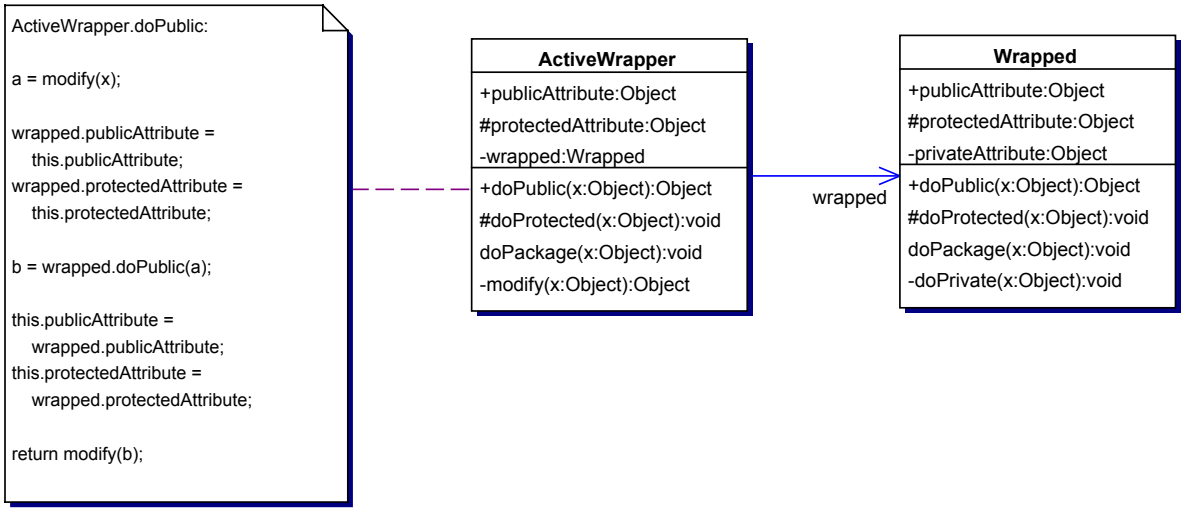

Abbildung 4-1: Aktiver Wrapper und Aufrufe von Methoden der gewrappten Klasse

Damit der Wrapper statt der gewrappten Klasse verwendet werden kann, muß er die gesamte von außen zugängliche Schnittstelle mit Methoden und Attributen nachbilden und die Methodenaufrufe auch bei nicht modifizierten Methoden an das Originalobjekt weiterleiten. Ebenso wie für Methodenaufrufe wird auch für die von außen zugänglichen Attribute die Wrapper-Instanz benutzt, so daß diese den aktuellen Zustand enthält. Daher müssen die Werte der Attribute vor dem Aufruf der Originalmethode in das Originalobjekt und danach zurück zum Wrapper kopiert werden.

Auf diese Weise können Änderungen an den von außen zugänglichen Attributen erfaßt werden, bevor und nachdem die Methode des Originalobjekts ausgeführt wird. Es entstehen allerdings Schwierigkeiten, wenn Attribute während der Ausführung der Methode verändert und noch vor ihrem Verlassen von anderen Objekten benötigt werden. Dies kann dann der Fall sein, wenn von der Methode ein Attribut des Originalobjekts geändert und danach ein Event ausgelöst wird, das von anderen Objekten empfangen wird und zum Lesen des Attributs aus dem Wrapper führt. Diese Fälle sind nur schwer und unter Berücksichtigung von Details der gewrappten Klasse zu berücksichtigen; bei einer sauberen Implementierung der zu wrappenden Klassen – insbesondere aus der Java-Klassenbibliothek – kann allerdings davon ausgegangen werden, daß alle Attribute privat sind und auf sie nur durch Set- und Get-Methoden zugegriffen werden kann, so daß im Wrapper kein duplizierter Zustand verwaltet wird.

4.2.1.2 Passive Wrapper

Statt die Aufrufparameter einer Methode zu verändern und an ein Objekt der gewrappten Klasse weiterzureichen, kann ein Wrapper die Parameter auch unverändert lassen und an eine spezielle Klasse weiterleiten, die ihrerseits die gesamte, um zusätzliche Maßnahmen erweiterte, Funktionalität der gewrappten Klasse bereitstellt, wie in Abbildung 4-2 dargestellt.

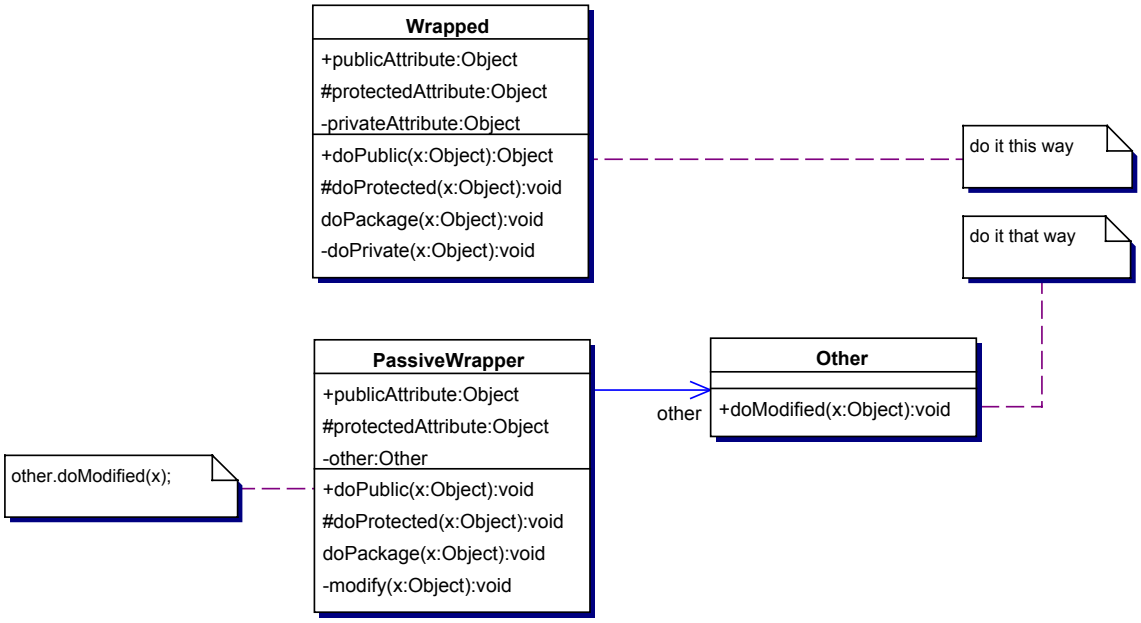

Abbildung 4-2: Passiver Wrapper und Aufrufe von Methoden einer separaten Klasse

Diese *passiven Wrapper* sind nur dann sinnvoll, wenn eine passende Version der zu ersetzenden Klasse bereits in einer Klassenbibliothek zur Verfügung steht. Ein passiver Wrapper ist bis auf die identische Schnittstelle von der gewrappten Klasse vollständig unabhängig, die zur Laufzeit dann unbenutzt bleibt. Somit entsteht keine Kopie der Werte öffentlicher Attribute, auch wenn diese im Wrapper erneut deklariert werden müssen.

4.2.1.3 Schlanke Wrapper

Sowohl aktive als auch passive Wrapper müssen sämtliche in der öffentlichen Schnittstelle der gewrappten Klasse vorhandenen Methoden anbieten und zu einem anderen Objekt umleiten. Um diesen bei der Implementierung und zur Laufzeit auftretenden Mehraufwand zu vermeiden, kann ein Wrapper auch von der gewrappten Klasse erben und muß somit nur die tatsächlich zu verändernden Methoden überschreiben. Durch die in Java implementierte →Polymorphie kann ein solcher *schlanker Wrapper* ebenfalls anstelle der ursprünglichen Klasse eingesetzt werden, ohne die öffentliche Schnittstelle vollständig neu zu implementieren. Die Attribute sind nicht in zwei verschiedenen Instanzen vorhanden und müssen daher auch nicht kopiert werden.

In den überschreibenden Methoden werden, falls nötig, wie bei einem aktiven Wrapper die Parameter verändert, in diesem Fall aber an die →Superklasse weitergegeben. Diese Situation ist in Abbildung 4-3 dargestellt.

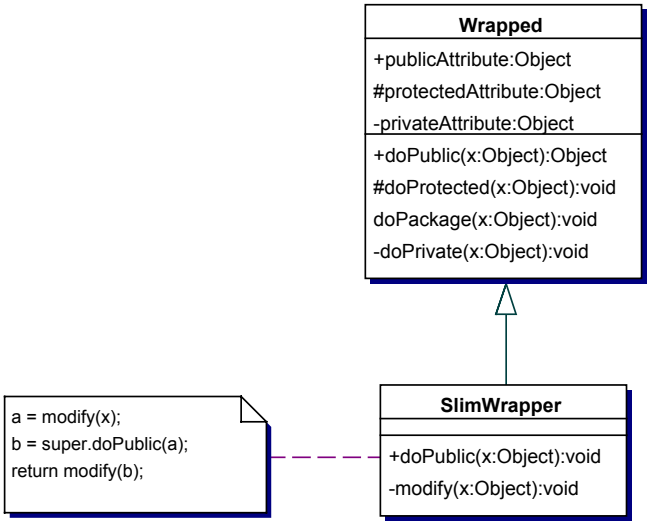

Abbildung 4-3: Schlanker Wrapper als Subklasse der gewrappten Klasse

4.2.1.4 Vergleich der Wrapper-Typen

Schlanke Wrapper führen zu geringem Aufwand und guter Überschaubarkeit bei der Implementierung, denn das veränderte Verhalten der Superklasse kann sehr einfach in der gewohnten objektorientierten Weise durch eine Vererbungsbeziehung und Polymorphie hergestellt werden. Gleichzeitig wird ein besseres Laufzeitverhalten als bei den anderen Typen erreicht; denn nur bei überschriebenen Methoden wird ein Zusatzaufwand verursacht. Alle anderen Methoden der gewrappten Klasse werden weiterhin ohne Umweg über den Wrapper direkt angesprochen.

Ein *aktiver Wrapper* bietet sich nur in Ausnahmefällen an, beispielsweise wenn dieser von einer anderen Klasse als der gewrappten Klasse erben soll; da Mehrfacherbung in Java nicht

möglich ist, muß dazu dieser Umweg in Kauf genommen werden. Allerdings ist dann auch die Aktualisierung eventuell vorhandener öffentlicher Attribute zu berücksichtigen, da deren Zustand in zwei Objektinstanzen gespeichert ist.

Autarke Wrapper sind nur dann sinnvoll, wenn das Verhalten einer Klasse vollständig verändert werden soll und nicht auf der ursprünglichen Klasse basieren kann. Ist das neue Verhalten bereits in einer anderen Klasse implementiert, kann statt dessen ein *passiver Wrapper* genutzt werden. In Abschnitt 4.2.3 wird jedoch ein Problem dieser Wrapper-Typen deutlich, durch das die Einsetzbarkeit auf bestimmte Fälle eingeschränkt wird.

4.2.2 Wrapping-Beziehung

In Java kann das Überschreiben einzelner oder aller Methoden einer Klasse durch Einschränkung der Zugriffsrechte verhindert oder auf Klassen desselben Package eingeschränkt werden. Insbesondere ein schlanker Wrapper würde damit auf die von der gewrappten Klasse vorgesehenen Manipulationsmöglichkeiten eingeschränkt. Autarke, aktive und passive Wrapper können dagegen eine Kopie der öffentlichen Klassenschnittstelle zur Verfügung stellen, wobei die aktiven Wrapper aber durch einschränkende Zugriffsrechte am Aufruf von Methoden der ursprünglichen Klasse gehindert werden können, wenn sie nicht darauf zugreifen können.

Darum muß ein Wrapper mit der von ihm gewrappten Klasse in eine spezielle Wrapping-Beziehung gebracht werden, die in Java bisher nicht vorgesehen ist. Dabei werden die Zugriffsmöglichkeiten der Wrapper auf die von ihnen gewrappten Klassen erweitert, so daß auch ein Aufrufen oder Überschreiben geschützter Methoden und Klassen möglich ist, wie in Abschnitt 4.3.2 beschrieben wird.

4.2.3 Mehrfaches Wrapping

Die Funktionalität einer gewrappten Klasse kann unter verschiedenen Gesichtspunkten verändert werden. Für die Klassen zum Lesen und Schreiben von Dateien bieten sich folgende Manipulationen an:

- *Verschlüsselung und Entschlüsselung* zur Sicherstellung der Vertraulichkeit von Daten,
- *Signierung* und spätere Prüfung der Integrität von Dateien,
- *Autorisierung* zur Verhinderung unberechtigter Zugriffe,
- *Logging,* um Dateisystemzugriffe abrechnen oder nachvollziehen zu können,
- *Umlenken in andere Verzeichnisse,* um nur Teile des Dateisystems für Zugriffe freizugeben und der Anwendung dennoch einen vollständigen Zugriff vorzuspiegeln, sowie
- *Debug-Hilfen,* beispielsweise durch Bildschirmausgaben bei bestimmten Zugriffen.

Bei der Entwicklung eines Wrappers für eine bestimmte Klasse können nicht sämtliche denkbaren Einsatzmöglichkeiten bedacht werden. Zudem wird ein Wrapper durch Kombination

mehrerer Zielsetzungen unübersichtlich und die Ausführung unnötig gebremst, wenn nicht alle Möglichkeiten dieses großen Wrappers genutzt werden sollen.

Es ist also vorteilhaft, mehrere Wrapper jeweils für eine spezielle Aufgabe zu entwickeln, die aber auch gemeinsam eingesetzt werden können. Dies wird durch eine Schachtelung von Wrappern erreicht, womit ein Wrapper selbst wieder gewrappt werden kann. Durch die Hintereinanderschaltung mehrerer spezieller Wrapper sind so verschiedene Modifikationen an einer Klasse möglich. In Abbildung 4-4 wird ein Beispiel der Schachtelung mehrerer Wrapper für die Systemklasse `java.io.FileInputStream` veranschaulicht. Die verschiedenen Wrapper verändern dabei genau die Methoden, für die es bei der jeweiligen Aufgabe nötig ist.

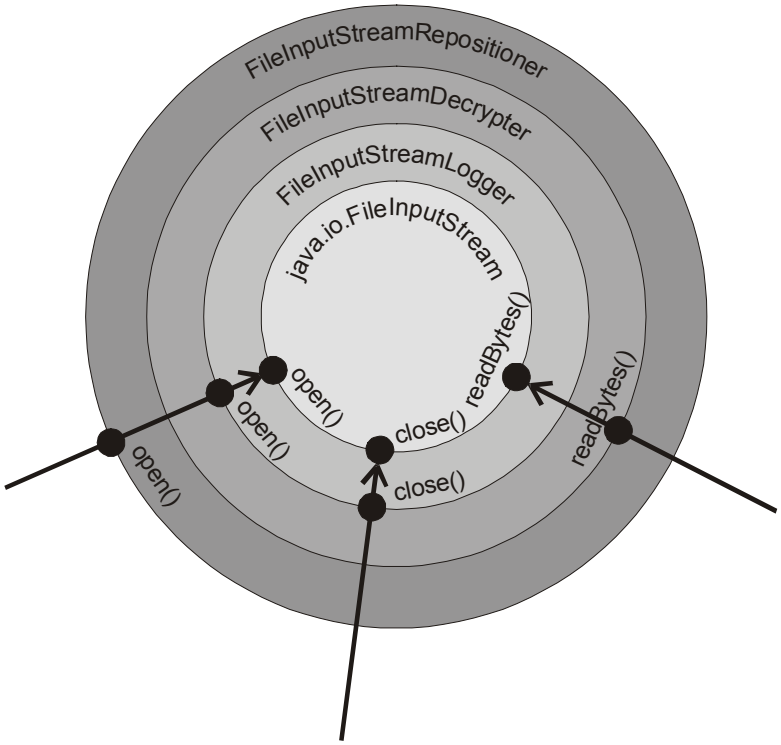

Abbildung 4-4: Geschachtelte Wrapper und überschriebene Methoden

Um die einzelnen Wrapper auch unabhängig voneinander einsetzen und entwickeln zu können, darf keine gegenseitige Kenntnis vorausgesetzt werden. Jeder Wrapper muß einzeln so entwickelt werden, daß er zur ursprünglich gewrappten Klasse paßt. Zur Laufzeit werden die unabhängig compilierten Wrapper dann je nach konfigurierter Reihenfolge miteinander verwoben. Autarke und passive Wrapper stellen hierbei ein Problem dar, da sie die Aufgaben nicht an die gewrappte Klasse weiterleiten und somit keine Verbindung zum nächstinneren Wrapper hergestellt werden kann. Diese Wrapper können daher nur als erster bzw. innerster Wrapper einer Klasse eingesetzt werden. Das bedeutet auch, daß nur ein einziger der eingesetzten Wrapper einer Klasse ein autarker oder passiver Wrapper sein kann.

4.3 Konzeption der Änderungen am Java-System

Um das vorgestellte Wrapping zu ermöglichen, müssen zwei grundlegende Abläufe des Java-Laufzeitverhaltens verändert werden. Erstens muß statt einer gewrappten Klasse von anderen Klassen ihr Wrapper aufgerufen werden, und zweitens erfordert ein Wrapper Zugriffsrechte auf die gewrappte Klasse, die über die ursprünglich von Java erlaubten Möglichkeiten hinausgehen. Beide Bereiche werden in diesem Abschnitt zunächst konzeptionell vorgestellt. Die Implementierungsdetails folgen in Abschnitt 4.4.

4.3.1 Modifikation der Klassenreferenzen

Das Wrapping wird zur Laufzeit durch Ersetzung der gewrappten Klassen mit ihren Wrappern vorgenommen. Diese Klassenersetzung wird über eine Modifikation der Referenzierung von Klassen gelöst. Referenzierte Klassen werden zur Laufzeit über ihren Namen angefordert und – falls noch nicht in der Java Virtual Machine registriert – durch einen Class Loader geladen wie in 2.2.1.2 beschrieben.

4.3.1.1 Namensersetzung

Um Klassen durch ihre Wrapper zu ersetzen, wird vor der Beauftragung des Class Loaders der Name einer gewrappten Klasse durch den Namen des Wrappers ersetzt. Durch diese *Namensersetzung* wird vom Dynamic Class Loading anschließend der Wrapper statt der gewrappten Klasse in das System geladen und dort unter deren Namen verankert. Somit beziehen sich alle folgenden Referenzierungen automatisch auf den Wrapper statt auf die von ihm gewrappte Klasse.

Aktive Wrapper benötigen zur Ausführung eine Instanz der gewrappten Klasse, schlanke Wrapper erben von der gewrappten Klasse. In beiden Fällen müssen die Wrapper ihre gewrappte Klasse referenzieren. Täten die Wrapper dies direkt, so würde auch in diesem Fall die Namensersetzung durchgeführt. Dies würde bei aktiven Wrappern zu einer endlosen Selbstreferenzierung und bei schlanken Wrappern zur einer Schleife in der Vererbungshierarchie führen.

Darum wird im Quelltext der Wrapper nicht der eigentliche Name angegeben, sondern ein speziell für diesen Wrapper veränderter Name. Dieser Name setzt sich unter anderem aus dem Namen der gewrappten Klasse und dem Namen des Wrappers zusammen. Der Wrapper `FileInputStreamLogger` aus Abbildung 4-4 beispielsweise referenziert somit statt `java.io.FileInputStream` eine Klasse mit dem konstruierten Namen `_java_io_FileInputStream_wrapwith_FileInputStreamLogger`. Der Name des Wrappers muß darin enthalten sein, um die Klassen bei mehrfachem Wrapping unterscheiden zu können.

Wrapper sind normale Java-Klassen und müssen daher wie üblich compiliert werden. Daher muß zum Zeitpunkt der Compilierung eine Klasse mit dem genannten konstruierten Namen zur Verfügung stehen. Darin müssen aber lediglich alle vom Wrapper aufgerufenen oder

überschriebenen Methoden mit ihrer Signatur ohne eine Implementierung enthalten sein. Dieser *Klassenrumpf* ermöglicht eine normale Compilierung des Wrappers und wird zur Laufzeit nicht benötigt; denn der konstruierte Name des Klassenrumpfs wird bei der Namensersetzung durch den Namen der gewrappten Klasse ersetzt. Durch dieses Vorgehen kann ein Wrapper zur Laufzeit schließlich die von ihm gewrappte Klasse aufrufen oder von ihr erben.

Die Situation während der Compilierung einiger schlanker Wrapper für `java.io.FileInputStream` ist in Abbildung 4-5 dargestellt. Statt von dieser Klasse erben die Wrapper von den speziell auf sie abgestimmten Klassenrümpfen, in denen lediglich die individuell benötigten Methoden enthalten sind.

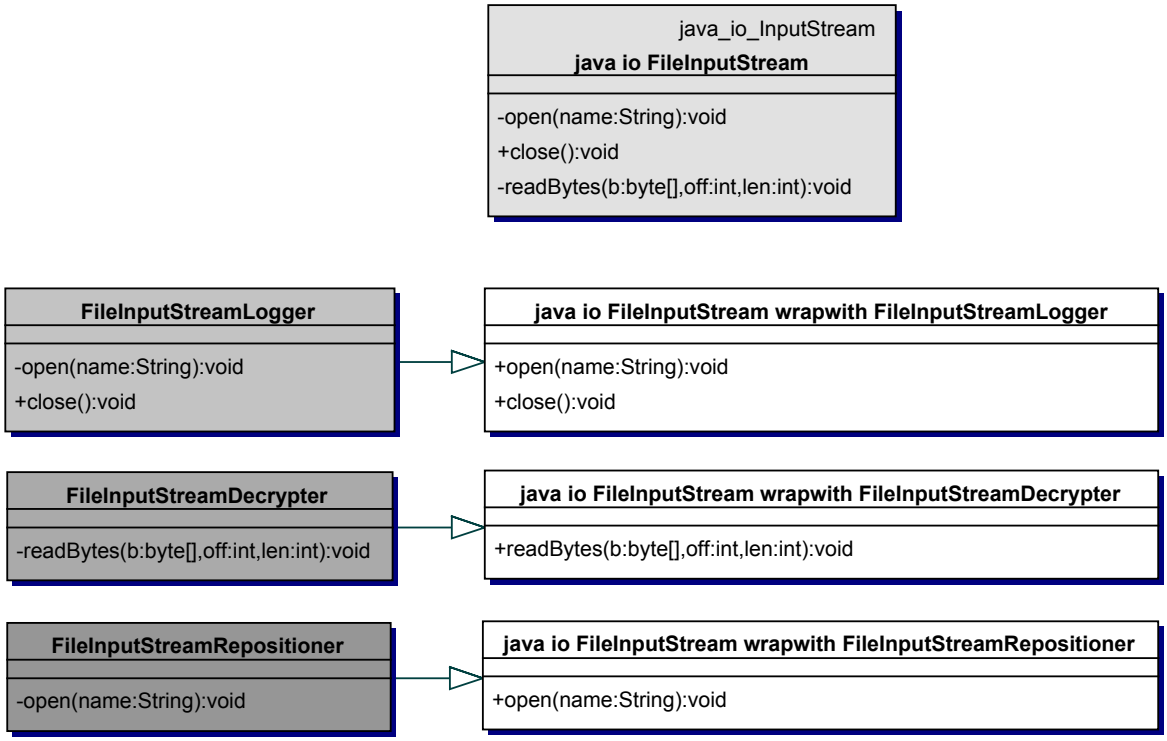

Abbildung 4-5: Schlanke Wrapper für java.io.FileInputStream, Klassenrümpfe zur Compilierung

Die nächsten beiden Abbildungen bieten einen Vergleich der Situationen ohne und mit Wrapper zur Laufzeit. Abbildung 4-6 enthält einige ausgewählte Klassen, die `java.io.FileInputStream` referenzieren sowie dessen Superklasse, wenn kein Wrapper aktiviert ist. Abbildung 4-7 zeigt dieselben Beziehungen mit zugeschaltetem schlanken Wrapper. Alle Referenzen beziehen sich nun auf den Wrapper `FileInputStreamLogger`; dieser erbt vom gewrappten `java.io.FileInputStream` und kann so mit Hilfe von `super` dessen Methoden aufrufen. Im Diagramm sind neben den Klassensymbolen die zur Laufzeit im Java-System verwendeten Klassennamen als Kommentare notiert. Wie oben beschrieben wird der Wrapper unter dem Namen der gewrappten Klasse geführt, und diese unter dem Namen des erwähnten Klassenrumpfes. Damit werden die Referenzen wie gewünscht zur Laufzeit verändert.

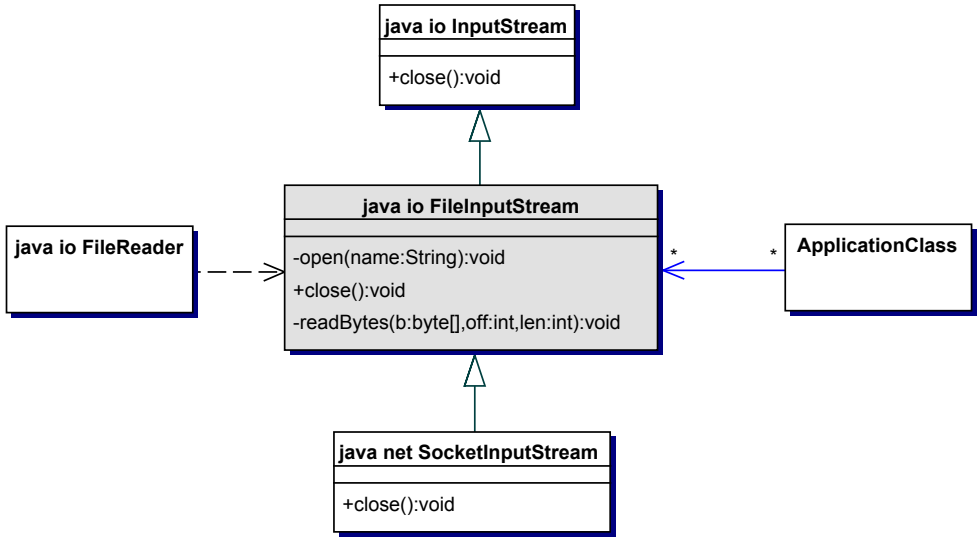

Abbildung 4-6: Einige Klassenbeziehungen von java.io.FileInputStream im Originalzustand

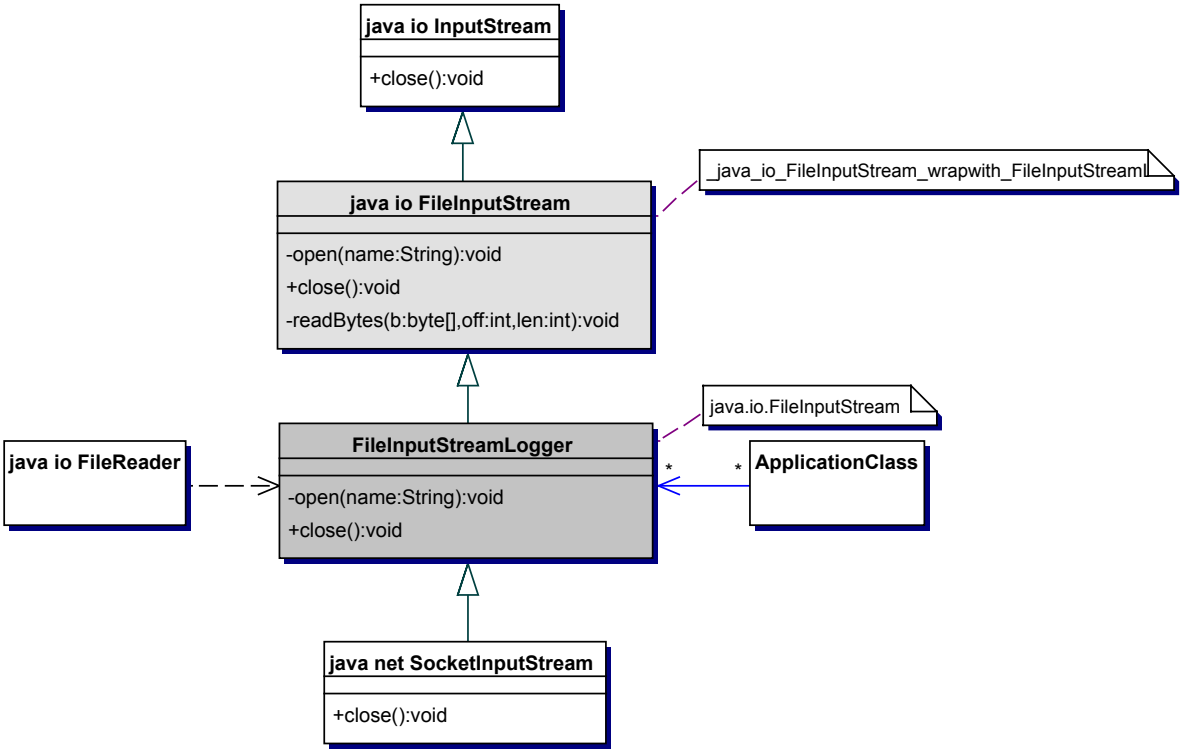

Abbildung 4-7: Klassenbeziehungen von java.io.FileInputStream mit einem Wrapper zur Laufzeit

4.3.1.2 Erweiterung auf mehrfaches Wrapping

Die bisherigen Überlegungen bezogen sich auf den Fall mit einem einzigen Wrapper. Bei mehreren ineinandergeschachtelten Wrappern wie in Abbildung 4-4 wird statt der gewrappten Klasse zur Laufzeit der äußerste Wrapper angesprochen, und jeder der Wrapper referenziert statt der gewrappten Klasse stets den nächstinneren Wrapper. Der innerste Wrapper schließlich benutzt dann die gewrappte Klasse. Durch diese Verkettung wird erreicht, daß jeder Wrapper die aufgerufenen Parameter in der von ihm vorgesehenen Weise modifiziert und anschließend an den nächstinneren Wrapper übergibt, den er für die von ihm gewrappte Klasse hält.

Also wird nicht nur die von den Wrappern eigentlich vorgesehene Klasse gewrappt, sondern auch die Wrapper selbst – bis auf den äußersten Wrapper. Um dies zu erreichen, ist die Namensersetzung auf das Mehrfach-Wrapping abgestimmt. Wie in Abbildung 4-8 graphisch veranschaulicht, werden zur Laufzeit für jede gewrappte Klasse neben ihrem eigenen Klassennamen (wrapped) ein Vektor mit sämtlichen aktiven Wrapper-Namen (wrapper[0..m]) sowie ein Vektor mit den Namen der für die einzelnen Wrapper vorgesehenen Klassenrümpfe (wrapperparent[0..m]) mitgeführt. Die Anzahl der verschachtelten Wrapper für diese Klasse ist m+1.

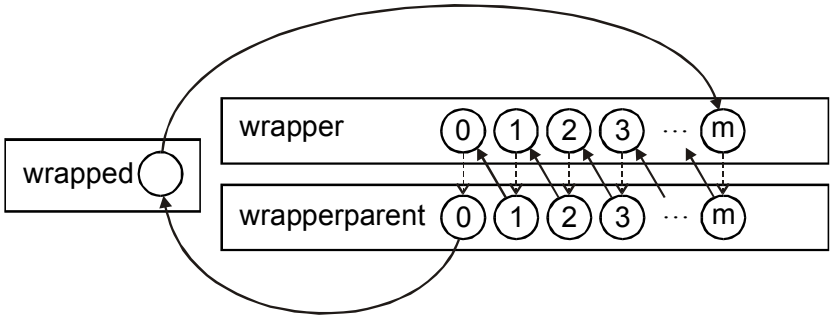

Abbildung 4-8: Schema der Ersetzung von Klassenreferenzen zur Laufzeit

Die Namensersetzung beim Laden der Klassen folgt den durchgezogenen Pfeilen:

- Der Name der gewrappten Klasse (wrapped) wird durch den Namen des äußersten Wrappers (wrapper[m]) ersetzt. Dadurch wird dieser im System statt der ursprünglichen Klasse angesprochen.
- Jeder Wrapper (wrapper[i] $\forall\ 0 \le i \le m$) referenziert eigentlich den von ihm angegebenen Klassenrumpf (wrapperparent[i]), was durch die gestrichelten Pfeile angedeutet ist. Stattdessen wird für $i > 0$ aber der nächstinnere Wrapper (wrapper[i-1]) geladen, was die Schachtelung mehrerer Wrapper ermöglicht.
- Der innerste Wrapper (wrapper[0]) enthält ebenfalls Referenzen auf den entsprechenden Klassenrumpf (wrapperparent[0]). Wie oben bereits beschrieben, wird statt dessen zur Laufzeit aber die gewrappte Klasse (wrapped) angesprochen, so daß der Wrapper deren Grundfunktionalität nutzen kann.

Bei nur einem einzigen Wrapper ($m = 0 \Leftrightarrow wrapper[m] = wrapper[0]$) entfällt die mittlere Regel.

Die Beziehungen der Wrapper aus Abbildung 4-4 zur Laufzeit sind in Abbildung 4-9 dargestellt. Der äußerste Wrapper ist darin unten und der innerste Wrapper oben angeordnet. In diesem Diagramm sind die veränderten Namen, unter denen die Klassen zur Laufzeit auftreten, wiederum neben deren Symbolen als Kommentare notiert. Die gewrappte Klasse erhält den Namen des im innersten Wrapper benutzten Klassenrumpfs. Alle inneren Wrapper werden wiederum gewrappt und treten daher unter dem Namen des Klassenrumpfs des nächstäußeren Wrappers auf. Der äußerste Wrapper schließlich wird unter dem Namen der gewrappten Klasse geführt.

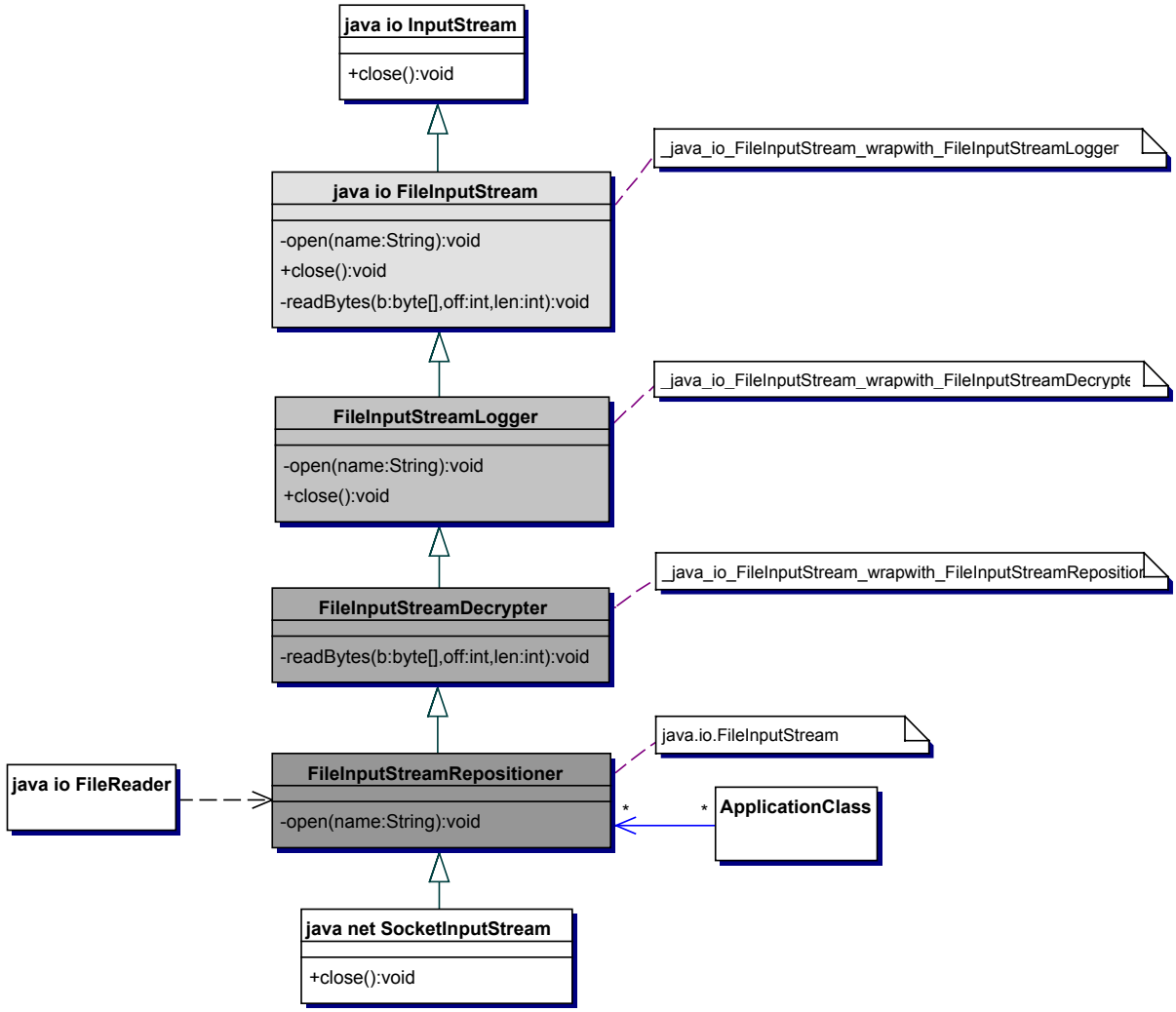

Abbildung 4-9: Klassenbeziehungen von java.io.FileInputStream mit mehreren Wrappern zur Laufzeit

Die in Abbildung 4-8 vorgestellten Datenstrukturen für die Namensersetzung enthalten dabei folgende Werte:

m = 2

wrapped = java.io.FileInputStream

wrapper[0] = FileInputStreamLogger
wrapperparent[0] = _java_io_FileInputStream_wrapwith_FileInputStreamLogger

wrapper[1] = FileInputStreamDecrypter
wrapperparent[1] = _java_io_FileInputStream_wrapwith_FileInputStreamDecrypter

wrapper[2] = FileInputStreamRepositioner
wrapperparent[2] = _java_io_FileInputStream_wrapwith_FileInputStreamRepositioner

Zusammen mit den Regeln zur Namensersetzung wird erreicht, daß die beteiligten Klassen unter den in Abbildung 4-9 zusätzlich eingetragenen Namen im Java-System geführt und angesprochen werden.

4.3.2 Erweiterte Zugriffsrechte für Wrapper

Passive und schlanke Wrapper benötigen Zugriff auf Methoden der gewrappten Klassen. Schlanke Wrapper müssen zudem die zu verändernden Methoden überschreiben können. Um auch dann in die Ausführung eingreifen zu können, wenn dies von den Anwendungs- oder Systementwicklern nicht bedacht oder nicht gewünscht wurde, muß ein Wrapper auch in solchen Fällen auf die von ihm gewrappte Klasse zugreifen und ihre Methoden überschreiben können, in denen dies durch Zugriffsrechte von Klassen oder Methoden eigentlich nicht erlaubt wäre, beispielsweise durch `private`-Methoden oder `final`-Klassen.

Die Compilierung der Wrapper-Klassen kann dabei noch ohne besondere Vorkehrung durchgeführt werden, obwohl die Sichtbarkeiten und Zugriffserlaubnisse auch vom Java-Compiler bereits überprüft werden. Denn die Wrapper referenzieren zum Compilierungszeitpunkt nicht die eigentlich gewrappten Klassen, sondern die Klassenrümpfe, die lediglich Signaturen der zu überschreibenden Methoden bereitstellen. Diese jeweils speziell für die Compilierung eines Wrappers vorzusehenden Klassenrümpfe definieren ihre Methoden so, daß der Zugriff bei der Compilierung eines Wrappers möglich ist, wie in Abbildung 4-5 gezeigt.

Die Sichtbarkeit und Zugriffsrechte werden von der Java Virtual Machine aber auch zur Laufzeit geprüft, um Angriffe durch modifizierte Compiler oder Fehler durch neu compilierte Anwendungsteile abzufangen [MF99]; die bei der Compilierung noch möglichen Zugriffe würden zur Laufzeit also unterbunden. Es gibt in Java bereits die Möglichkeit, durch Vergabe einer `java.lang.reflect.ReflectPermission` mit dem Target `suppressAccessChecks` diese Prüfungen zu umgehen [Sun98d]; allerdings gilt diese Erlaubnis nur für Methodenaufrufe, die über die Reflection API durchgeführt werden, und nicht für herkömmliche Aufrufe. Daher werden einige Teile der Bytecode Verification aus 2.2.1.1.2 für Wrapper außer Kraft gesetzt, um diese möglichst einfach implementieren zu können und insbesondere auch schlanke Wrapper als Erben geschützter Klassen zu erlauben.

4.3.2.1 Aufhebung von Zugriffsbeschränkungen

Die weiterentwickelte Java Virtual Machine umgeht für Wrapper die beim Laden oder bei der Ausführung geprüften Zugriffserlaubnisse und Einschränkungen der Sichtbarkeit, so daß ein Überschreiben von Methoden und der Zugriff auf Klassen und Methoden entgegen der Java-Konventionen möglich werden. Die folgenden Fälle werden auf diese Weise berücksichtigt:

- *Protected-Methoden* sind normalerweise nur für erbende Klassen und Klassen innerhalb desselben Package sichtbar. Die Package-Prüfung wird für Wrapper umgangen, damit diese auch in anderen Packages als dem der gewrappten Klasse definiert werden können.
- *Non-public-Methoden* werden ohne besondere Zugriffsangabe definiert und sind eigentlich nur im Package ihrer Klasse sichtbar. Die Sichtbarkeit wird auf Wrapper dieser Klasse ausgedehnt.
- *Non-public-Klassen* sind wie Non-public-Methoden nur innerhalb des Package und zusätzlich für ihre Wrapper sichtbar.
- *Final-Methoden* sind ausdrücklich als nicht zu überschreiben gekennzeichnet. Für Wrapper ist das Überschreiben aber nun möglich.
- *Final-Klassen* werden ebenso behandelt wie Final-Methoden.

Für die Lockerung der aufgezählten Einschränkungen wird zur Laufzeit an den nötigen Stellen überprüft, ob zwei Klassen zueinander in einer Wrapping-Beziehung stehen. Dabei kann es sich ebenso um eine gewrappte Klasse und den innersten oder einzigen Wrapper handeln wie um zwei aufeinanderfolgende Wrapper derselben Klasse. In diesen Fällen wird der Zugriff auf ansonsten nicht sichtbare bzw. zugreifbare Methoden oder Klassen doch gewährt. Der Geschwindigkeitsnachteil durch die Zusatzprüfungen ist tolerierbar, da diese für jede in einer Klasse vorhandene symbolische Referenz nur einmal bei ihrer ersten Benutzung durchgeführt wird und danach eine direkte Referenz zur Verfügung steht, die nicht mehr überprüft werden muß.

4.3.2.2 Dynamisierung statisch gebundener Referenzen

Static- und Private-Methoden sind anders als die oben aufgezählten Fälle zu behandeln, da bei diesen kein spätes Binden anhand des tatsächlich instanziierten Objekts durchgeführt wird, sondern die aufzurufende Methode und ihre Klasse bereits zur Compilierungszeit feststehen. In einem solchen Fall kann ein Überschreiben der Methoden nicht durch eine Lockerung der Laufzeitprüfungen erreicht werden; denn im →Bytecode der Klasse werden spezielle Instruktionen benutzt, die den Methodenaufruf statisch an die in einer bestimmten Klasse definierte Methode binden und nicht nach der Klasse des instanziierten Objekts richten.

In der Spezifikation des Bytecode für compilierte Java-Klassen gibt es vier verschiedene Instruktionen für Methodenaufrufe [LY99]:

- *invokevirtual.* Dabei wird zur Laufzeit ein spätes Binden ausgeführt und so die aufgerufene Methode aus der Klasse des instanziierten Objekts bestimmt. Dadurch ist ein Über-

schreiben von Methoden grundsätzlich möglich, kann aber durch Zugriffsbeschränkungen der Klassen oder Methoden verhindert werden. Die Prüfung dieser Einschränkungen sind für Wrapper wie oben beschrieben gelockert.

- *invokeinterface* für Aufrufe der Methoden eines Interface. Prinzipiell entspricht diese Instruktion dem beschriebenen invokevirtual, allerdings werden die durch eine Klasse implementierten Methoden eines Interface etwas anders referenziert als die einer ererbten Klasse [Ven99].
- *invokestatic* wird bei Aufrufen statischer Klassenmethoden eingesetzt. Bei diesen steht die Klasse, aus der die aufzurufende Methode stammt, schon zur Compilierungszeit fest. Ein Überschreiben ist weder vom Java-Compiler noch von der Java Virtual Machine vorgesehen. Daher werden auch keine Einschränkungen geprüft, die für Wrapper gelockert werden könnten. Somit ist für ein Überschreiben von statischen Klassenmethoden eine besondere Behandlung von invokestatic innerhalb der modifizierten Java Virtual Machine nötig.
- *invokespecial* wird für drei spezielle Aufgaben genutzt: Erstens für *Konstruktoraufrufe*, die einerseits nur statisch von ihrer Klasse abhängen; andererseits muß aus einem Konstruktor auch der Konstruktor der Superklasse aufgerufen werden können.

  Zweitens wird invokespecial bei Benutzung von `super` für explizite Aufrufe von Methoden der Superklasse in den Bytecode eingefügt. In sehr alten Java-Versionen (vor 1.0.2) wurde dabei statisch diejenige Klasse angesprochen, in der zur Compilierungszeit die geforderte Methode enthalten war. Ab der genannten Version wird jedoch auch berücksichtigt, daß sich die benötigte Methode durch neu compilierte Superklassen in anderen Klassen befinden kann [Ven99]. Daher wird für invokespecial bei Benutzung mit `super` nicht mehr ein rein statischer Aufruf ausgeführt, sondern es werden zunächst die Superklassen nach der geforderten Methode durchsucht. Somit werden auch die zur Laufzeit in die Klassenhierarchie eingefügten Wrapper korrekt angesprochen.

  Drittens und letztens wird invokespecial für Aufrufe von *Private-Methoden* genutzt. Da diese Methoden nur innerhalb einer Klasse bekannt sind, kann sich ein Aufruf auch nur auf eine Methode dieser Klasse beziehen, weshalb kein spätes Binden durchgeführt wird. Dies ist für das Überschreiben durch einen Wrapper hinderlich, denn Private-Methoden können damit ebenso wie statische Klassenmethoden nicht ohne weiteres überschrieben werden.

Das Überschreiben von statischen Klassenmethoden und Private-Methoden ist in der modifizierten Java Virtual Machine möglich. Dabei wird bei der Ausführung eines invokestatic oder invokespecial zunächst überprüft, ob es sich bei der Zielklasse um eine gewrappte Klasse handelt. Falls ja, werden nacheinander die für diese Klasse eingetragenen Wrapper nach einer Methode mit gleicher Signatur durchsucht. Dies geschieht analog zur Suche einer Methode in den Superklassen bei Benutzung von `super`.

Bei der nun dynamischen Ausführung der ursprünglich statischen invokestatic- und invokespecial-Aufrufe entsteht so ein Geschwindigkeitsnachteil. Nach der ersten Durchführung

werden diese Befehle im Arbeitsspeicher jedoch durch Sun-spezifische Instruktionen (`invokenonvirtual_quick` bzw. `invokevirtual_quick`) mit einer direkten Referenz auf die letztendlich genutzte Methode ersetzt, so daß diese Prüfung nicht erneut durchgeführt wird.

4.3.3 Konfiguration der eingesetzten Wrapper

Der entwickelte Ansatz erlaubt ein Wrapping oder Mehrfach-Wrapping beliebiger Klassen aus der Java-Klassenbibliothek oder von Komponenten- und Anwendungsklassen. Je nach Anwendungsfall können flexibel weitere Wrapper hinzugefügt werden. Die Definition der zu verwendenden Wrapper darf also nicht statisch, sondern muß konfigurierbar sein.

Um für jede Anwendung individuell die eingesetzten Wrapper festlegen zu können, kann die Konfiguration der Wrapper beim Start der Java Virtual Machine mit einem Parameter angegeben werden. Der Aufruf `java –wrap:wrapdef.txt MainClass` startet die Ausführung der `MainClass.class` unter Berücksichtigung der Wrapping-Konfiguration in der Konfigurationsdatei `wrapdef.txt`. Voraussetzung ist die Angabe der Option `wrapmore` in der globalen Konfigurationsdatei (siehe unten).

Um eine systemweite Konfiguration ohne Angabe beim Start der Anwendung festlegen zu können, wird von der erweiterten Java Virtual Machine zwangsweise zuerst eine globale Konfigurationsdatei namens `wrapping.properties` aus dem Unterverzeichnis `lib` der Java-Laufzeitumgebung gelesen (`%JAVA_HOME%/lib/wrapping.properties`). Die globale Konfigurationsdatei wird vor einer eventuell in der Kommandozeile angegebenen anwendungsspezifischen gelesen. Die global festgelegten Wrapping-Beziehungen können durch eine anwendungsspezifische Konfigurationsdatei daher nicht aufgehoben oder untergraben, sondern lediglich um zusätzliche Wrapper erweitert werden.

Werden Wrapper definiert, die beim Laden oder während der Benutzung einen Fehler verursachen, so kann die Java Virtual Machine unter Umständen nicht vollständig initialisiert werden, wenn dieser Fehler bereits beim Laden der Systemklassen auftritt. Wenn die Klassen zur Ausnahmebehandlung und Bildschirmausgabe zum Fehlerzeitpunkt ebenfalls noch nicht bereit sind, kann die resultierende Abbruchmeldung entsprechend unspezifisch ausfallen.

Um sicherzugehen, daß die weiterentwickelte Java Virtual Machine bei der Ausführung einer Anwendung tatsächlich eingesetzt wird, kann mit `–wrap:yes` ein Parameter zur expliziten Aktivierung des Wrapping angegeben werden, der einer herkömmlichen Virtual Machine nicht bekannt ist und dort zu einer Fehlermeldung beim Start führt. Zu Entwicklungs- und Testzwecken kann das gesamte Wrapping durch den Parameter `-wrap:no` abgeschaltet werden, wenn in der globalen Konfigurationsdatei `wrapoptional` angegeben ist (siehe unten). Dann wird auch die globale Konfigurationsdatei nicht gelesen.

Sowohl die globale als auch anwendungsspezifische Wrapping-Konfigurationen werden in Textdateien definiert, die wie folgt interpretiert werden:

- Leere Zeilen sowie Zeilen, die mit # beginnen, werden ignoriert.

- Beginnt eine Zeile mit `wrapclasspath`, so werden anschließend beliebig viele Leerzeichen und eine Pfadangabe in plattformspezifischem Format erwartet. Der angegebene Pfad wird dem Java-Bootclasspath hinzugefügt, so daß die dort abgelegten Wrapper-Klassen vom Java-System schon vor der Installation eines →Class Loaders gefunden werden können. Dies ist wichtig für das Wrapping von Systemklassen, die bereits beim Systemstart geladen werden, bevor der eigentliche →Klassenpfad ausgewertet und berücksichtigt wird. Zudem kann der für die Anwendung festgelegte Klassenpfad hierdurch unverändert bleiben.

  Mehrere Verzeichnisse mit Wrapper-Klassen können durch Semikola getrennt in einer `wrapclasspath`-Zeile oder in mehreren `wrapclasspath`-Zeilen angegeben werden. Die Suchreihenfolge entspricht der des Java-Systems; die Verzeichnisse werden also gemäß der Reihenfolge ihrer Definition durchsucht.

- Um zu verhindern, daß fremde oder ungeprüfte Wrapper in das System gebracht werden, kann mit `wrappersigner name` festgelegt werden, von wem diese signiert sein müssen. Der dazugehörige öffentliche Schlüssel wird aus dem von Java verwendeten Keystore entnommen, wie in Abschnitt 2.2.2.1.3 beschrieben.

  Um festzulegen, daß Wrapper von mehreren Parteien signiert werden müssen, können in einer Zeile mit `wrappersigner name1 name2 …` mehrere Namen angegeben werden. Um alternative Möglichkeiten zur Signierung der Wrapper zu erlauben, sind mehrere Zeilen mit `wrappersigner` anzugeben, von denen mindestens eine erfüllt sein muß.

- In der globalen Konfigurationsdatei muß mit einer Zeile `wrapmore` explizit die Angabe weiterer Wrapping-Konfigurationen in der Kommandozeile erlaubt werden. Ohne diese Angabe wird die Kommandozeilenoption `-wrap:dateiname` ignoriert, so daß kein weiterer Wrapper in das System gebracht werden kann, um eine böswillige Ausnutzung der erweiterten Zugriffsrechte für Wrapper durch Benutzer zu verhindern.

- Die Angabe `wrapoptional` erlaubt die Abschaltung des Wrapping mit dem Kommandozeilenparameter `-wrap:no`. Andernfalls können die festgelegten Wrapper nicht umgangen werden, es sei denn die Java Virtual Machine wird ausgetauscht oder die Wrapper-Klassen durch andere ersetzt, was aber durch Signieren der Wrapper verhindert werden kann.

- Zeilen, die mit `wrap` beginnen, enthalten die Definition einer Wrapping-Beziehung in folgendem Format: `wrap WrappedClass with Wrapper`. Die Klassenangaben müssen dabei inklusive der Packages in Punkt-Notation vorgenommen werden.

  Beispiel: `wrap java.io.FileInputStream with FileInputStreamLogger`

  Ein Mehrfach-Wrapping wird durch mehrere Zeilen `wrap A with` B_i $(0 \le i \le m)$ erreicht, wobei die erste gelesene Zeile den innersten Wrapper (wrapper[0] = B_0) und die letzte den äußersten Wrapper (wrapper[m] = B_m) für die gewrappte Klasse (wrapped = A) definiert. Die ebenfalls benötigten Namen der Klassenrümpfe werden wie folgt konstruiert:

```
wrapperparent[i] = packageWithDot(wrapper[i]) + "_" +
                   str_replace(".","_", wrapped[i]) +
                   "_wrapwith_" +
                   withOutPackage(wrapper[i])
```

packageWithDot(x) extrahiert dabei die Package-Angabe von x (inklusive des letzten Punkts), str_replace(replace,with,string) ersetzt die Zeichenfolge replace durch with in string, withOutPackage(x) gibt den Klassennamen x ohne die Package-Angabe zurück und + dient zur Konkatenation von Zeichenketten.

Diese Namen der Klassenrümpfe müssen von den Wrappern verwendet werden, wenn sie auf die gewrappte Klasse bzw. den nächstinneren Wrapper zugreifen.

- Zeilen, die mit `wrapmain` beginnen, definieren Wrapper für die Klasse mit der Main-Methode, die beim Programmstart geladen wird.

  Beispiel: `wrapmain with MainWrapper`

  Diese besondere Behandlung der Main-Klasse ist nötig, um Aktionen vor jedem Programmstart durchführen zu können, da im Voraus nicht die Klassennamen aller zu startenden Anwendungen bekannt sind.

Ein Beispiel für eine konkrete Wrapping-Konfigurationsdatei ist in Anhang A.4.4 zu finden.

4.4 Implementierung der Änderungen in der Java Virtual Machine

Die Implementierung der konzeptionell vorgestellten Änderungen erfolgt in den Quelltexten der Java Virtual Machine, die von Sun zu Forschungszwecken freigegeben sind und bei kommerziellem Einsatz lizenziert werden können. Diese Quelltexte liegen neben einigen Assembler-Teilen in C vor, weshalb auch die Wrapping-Mechanismen in C implementiert werden. Die Wrapper selbst werden jedoch wie in 4.2 beschrieben als herkömmliche Java-Klassen implementiert.

Im Rahmen dieser Arbeit wurden zunächst die Quelltexte des JDK 1.2.2 für Windows zugrundegelegt. Anschließend wurde die Java Virtual Machine von JDK 1.3.0 angepaßt und im weiteren Verlauf zur Validierung eingesetzt. Dabei wurde nicht die Hotspot Virtual Machine [DG98], sondern die herkömmliche Classic Virtual Machine benutzt.

Die Compilierung der Java Virtual Machine erzeugt bei der verwendeten Windows-Plattform die Datei `jvm.dll`. Um das Wrapping zu verwenden, ist lediglich diese Datei in die normale Java-Laufzeitumgebung einzusetzen. Java unterstützt die Installation mehrerer Virtual Machines, so daß beim Programmstart festgelegt werden kann, ob die ursprüngliche oder die veränderte Version benutzt werden soll.

Die nötigen Änderungen an der Java-Laufzeitumgebung betreffen also ausschließlich die Java Virtual Machine und nicht die Klassenbibliothek. Dies setzt bei einer Weiterentwicklung der

Java-Umgebung zwar die Implementierung der Änderungen in neuen Versionen der Quelltexte voraus; die Virtual Machine wird jedoch wesentlich seltener Änderungen unterzogen als die Klassenbibliothek. Eine Java Virtual Machine kann darüber hinaus auch für verschiedene Java-Versionen eingesetzt werden[4].

Die grundlegende Veränderung betrifft alle Stellen, an denen die Referenzen zu anderen Klassen aufgelöst werden. Eine zentrale Rolle spielt dabei das in 2.2.1.2 beschriebene Dynamic Class Loading, vor dessen Aufruf bei zu wrappenden Klassen eine Namensersetzung des angeforderten Klassennamens gemäß Abbildung 4-8 durchgeführt wird. Daraufhin lädt der Class Loader statt der eigentlich von der Anwendung geforderten Klasse ihren Wrapper, wodurch dieser im System anstelle der gewrappten Klasse verankert und nachfolgend angesprochen wird.

Um Änderungen an der Java Virtual Machine zu vermeiden, wurde im Rahmen dieser Arbeit auch versucht, die Anpassungsmöglichkeiten des Dynamic Class Loading zu nutzen und dazu mit einem anwendungsdefinierten und in Java implementierten Class Loader experimentiert, der dieselbe Aufgabe erfüllen sollte. Dieses Vorhaben scheiterte jedoch an einer Einschränkung beim Laden von →Native Code, so daß nur Klassen ohne Native Code gewrappt werden konnten. Die Java-Sicherheitsvorkehrungen, die das Laden einer anderen als der angeforderten Klasse verhindern sollen, konnten zuvor erfolgreich umgangen werden, was sich aber in zukünftigen Java-Versionen anders herausstellen könnte. Die erweiterten Zugriffsrechte der Wrapper auf die von ihnen gewrappten Klassen können ebenfalls nur innerhalb der Java Virtual Machine implementiert werden. Ohne diese Möglichkeiten wäre das Wrapping geschützter Klassen und Methoden nicht möglich.

Die in 4.3.3 erwähnte Möglichkeit der Prüfung von Wrapper-Signaturen wurde im Rahmen dieser Arbeit aus Aufwandsgründen zugunsten anderer Funktionalität nicht implementiert. Die Herkunft der Wrapper kann daher nicht sichergestellt werden. Bei einem Einsatz der erweiterten Virtual Machine sollte diese Prüfung hinzugefügt werden, um zusätzlichen Schutz gegen einen Austausch der ursprünglichen mit boshaften Wrapper-Klassen zu erreichen.

In Anhang A.1 wird jede Änderung an der Java Virtual Machine dokumentiert und die dabei verfolgte Absicht beschrieben, so daß damit eine Neuimplementierung in einer anderen Java-Version oder der Hotspot Virtual Machine mit begrenztem Aufwand möglich sein sollte; die Umstellung auf JDK 1.3.0 war mit geringem Aufwand möglich.

[4] „The Java HotSpot™ Server VM 2.0 is [...] fully compatible with the Java™ 2 Platform, Standard Edition version 1.3 (and version 1.2.2) on Windows.“
Quelle: http://www.javasoft.com/products/hotspot/2.0/download.html, 08.05.2000

4.5 Fazit

Die veränderte Java-Laufzeitumgebung bietet mit dem Wrapping-Mechanismus umfassende Eingriffsmöglichkeiten in bestehende Software-Systeme. Die wenigen nötigen Veränderungen der Java Virtual Machine erhöhen die Chancen auf eine einfache Portierung zu anderen Plattformen oder anderen Virtual Machines bzw. zu neuen Releases eines Anbieters.

Wrapper werden als herkömmliche Klassen vollständig in Java programmiert. Üblicherweise können sie mit dem Vererbungsmechanismus realisiert werden, so daß die Entwicklung der Wrapper in bekannter, objektorientierter Weise erfolgen kann. Damit ist zur Veränderung der Funktionalität lediglich ein Überschreiben der entsprechenden Methoden nötig und eine Wiederverwendung der ursprünglichen Klasse für die Grundfunktionalität möglich. Die erweiterten Zugriffsrechte erlauben den Wrappern auch die Anpassung geschützter Klassen und Methoden.

Im Gegensatz zu einigen anderen Ansätzen, die in Abschnitt 1.2.3 vorgestellt wurden, müssen dazu aber nicht sämtliche Methodenaufrufe geprüft werden, was eine Verschlechterung des gesamten Laufzeitverhaltens verursachen würde. Die Wrapper werden einmalig so im System verankert, daß sie statt der ursprünglichen Klassen angesprochen werden und erst beim Aufruf aktiv werden.

In dieser Eigenschaft ähnelt der Wrapping-Mechanismus dem Konzept der Polymorphie, durch das ebenfalls Objekte aus einer anderen als der ursprünglich vorgesehenen Klasse – und zwar aus deren Subklassen – von anderen Programmteilen aufgerufen werden können. Allerdings bezieht sich die Polymorphie auf einzelne Objekte, weshalb bei deren Erzeugung die konkrete Klasse angegeben werden muß. Um auch die Objekterzeugung zu flexibilisieren, kann ein Factory-Entwurfsmuster [GHJ95] eingesetzt werden, so daß die letztendlich benutzte Klasse nicht fest codiert werden muß, sondern durch Erzeugung einer anderen Factory zur Laufzeit festgelegt werden kann. Während diese Möglichkeit aber jeweils im Programmcode vorgesehen sein muß, kann mit Wrapping jede Klasse ohne Anpassungen im Quelltext durch einen oder mehrere Wrapper ersetzt werden.

Wrapper können sowohl für Klassen der Java-Klassenbibliothek eingesetzt werden als auch für Klassen aus zusätzlichen Bibliotheken oder zur Anpassung zugekaufter Komponenten oder einzelner Klassen einer bestehenden Anwendung. Die dazu nötige Kenntnis des Aufbaus einer Klasse oder einer Klassenhierarchie kann mit der Reflection API auch ohne deren Quelltexte erlangt werden. Dennoch dürfte es schwierig sein, Wrapper für fremde und unbekannte Anwendungsklassen zu entwickeln und ihre Funktionsfähigkeit in verschiedenen Situationen vollständig zu testen.

4.5.1 Einsatzmöglichkeiten für Wrapping

Bei der Entwicklung von Anwendungen kann die eigentliche Funktionalität getrennt von den Sicherheitsaspekten implementiert werden, die durch Wrapping von einem Administrator

beim späteren Einsatz der Anwendung individuell konfiguriert werden können; universell einsetzbare Wrapper mit dieser Zielsetzung werden in Abschnitt 5.4.3 vorgestellt. Um den Installationsaufwand für Analyse und Konfiguration einer auf Wrapping angewiesenen Anwendung oder Komponente zu verringern, sollten dem Administrator vom Lieferanten Konfigurationsdateien zur Verfügung gestellt werden, die auf maximale Sicherheit eingestellt sind und nur noch auf die jeweilige Einsatzsituation angepaßt werden müssen.

Ein Schutz vor Denial of Service Attacks aus Abschnitt 2.1.1.3, bei denen übermäßig viele ressourcenintensive Objekte erzeugt werden, könnte durch Wrapper erreicht werden, die beim Aufruf des Konstruktors oder anderer Methoden einen Zähler erhöhen und bei Überschreiten eines Grenzwerts Gegenmaßnahmen ergreifen.

Zur Implementierung von aufwendigen Sicherheitsmodellen wie Mandatory Access Control oder dynamischen Modellen wie Chinese Wall (siehe 2.1.2) kann es nötig sein, alle Objekte mit zusätzlichen Sicherheitsattributen wie einer Sicherheitsstufe auszustatten und diese Attribute der an einem Zugriff beteiligten Objekte in einem speziell angepaßten Security Manager (siehe 2.2.2.2) zu berücksichtigen. Der Wrapping-Mechanismus kann dabei als Grundlage dienen, um beispielsweise die Klasse `java.lang.Object` um zusätzliche Attribute und Methoden zu erweitern, die damit in sämtlichen Objekten aller Klassen vorhanden sind.

Objektorientierte Datenbankanbindungen, die eine Persistenz von Objekthierarchien ohne den Aufruf von Bibliotheksmethoden zum Laden verknüpfter Objekte bei der Navigation innerhalb dieser Hierarchien bieten möchten, können dazu alle notwendigen Befehle automatisch in die Klassendateien der persistenten Klassen einfügen. Diese Technik wird beispielsweise von der Objektdatenbank Versant eingesetzt, dessen sogenannter Enhancer[5] die compilierten Klassendateien bearbeitet, so daß im Quellcode keine zusätzlichen Befehle berücksichtigt werden müssen. Ein Nachteil entsteht dabei jedoch, wenn persistente Klassen von Klassen der Java-Klassenbibliothek oder anderer Bibliotheken erben. In diesem Fall müssen nämlich auch Teile dieser Bibliotheken mit dem Enhancer bearbeitet werden, so daß die persistenten Klassen nur mit den veränderten Bibliotheken ausgeführt werden können. Statt dessen könnte der Enhancer nun Wrapper mit dem zusätzlichen Programmcode für die persistenten Klassen und für die betroffenen Bibliotheksklassen erzeugen, so daß die ursprünglichen Klassen unverändert bleiben und zur Laufzeit durch ihre Wrapper ersetzt werden können.

Auch während der Software-Entwicklung selbst können Wrapper hilfreich sein. Zusätzlicher Code, der zum Testen benutzt wird, aber später nicht mit ausgeliefert werden soll, kann in einem Wrapper untergebracht werden, der lediglich in einer Testphase oder beim Debuggen aktiviert wird. So können vom Wrapper beispielsweise Protokolldateien oder Bildschirmausgaben erzeugt oder zusätzliche Prüfungen durchgeführt werden. Diese Debug-Codes müssen durch die Unterbringung in einem nur zur Entwicklungszeit aktivierten Wrapper vor der Auslieferung nicht aus den entwickelten Klassen entfernt werden.

[5] http://www.versant.com/developer/_docs/jvi/usage/doc_jvi24_usage_03.html

Ähnliche Wrapper können auch zum Testen fremder Komponenten verwendet werden, denn nicht nur deren offizielle Schnittstellen, sondern alle vorhandenen Methoden können mit der Reflection API herausgefunden werden. Auf diese Weise können auch Methodenaufrufe an Komponenten über deren Schnittstellen oder Aufrufe zwischen mehreren Klassen innerhalb einer Komponente analysiert werden. Mit protokollierenden Wrappern für Systemklassen kann zusätzlich verfolgt werden, welche Ressourcenzugriffe diese Komponenten ausführen.

Eine weitere Anwendungsmöglichkeit für Wrapper ist der Schutz des Inhalts von Objekten, beispielsweise als Signed Object, Sealed Object oder Guarded Object, die in 2.2.3.4 vorgestellt wurden. Diese Möglichkeiten stehen in der Java-Klassenbibliothek zur Verfügung, müssen jedoch bei der Erzeugung und Nutzung der betroffenen Objekte durch zusätzlichen Programmcode berücksichtigt werden. Ein Guard-Objekt, das den Zugriff auf ein geschütztes Objekt regelt, könnte zur Laufzeit als Wrapper statt des geschützten Objekts eingesetzt werden und bei Methodenaufrufen die Berechtigungen des Aufrufers prüfen. So müßte nicht explizit ein `java.security.GuardedObject` eingesetzt werden, was die Anwendungsentwicklung vereinfachen und den Einsatz des Guard-Mechanismus flexibler machen würde. Entsprechendes gilt für SignedObject und SealedObject.

Dabei ist auch denkbar, die Wrapper-Klassen mit einem automatischen Tool oder sogar dynamisch zur Laufzeit über die Reflection API [GJS00] zu erzeugen und on-the-fly zu compilieren. Dies bietet sich bei Test-Wrappern oder der Implementierung von Entwurfsmustern wie Guarded Object an, die gleichförmig auf beliebige Objekte angewendet werden können und nicht eine Änderung der Funktionalität einer speziellen Klasse zum Ziel haben. Mit einer solchen Weiterentwicklung des Wrapping könnten die in Abbildung 4-5 vorgestellte Bereitstellung von Klassenrümpfen zur Compilierung und die Entwicklung einzelner Wrapper für jede zu wrappende Klasse eingespart werden.

Der entwickelte Wrapping-Mechanismus steht unabhängig von den Zielen dieser Arbeit als eigenständige Lösung also auch für Aufgaben zur Verfügung, die in den folgenden Kapiteln nicht behandelt werden. Hier werden Wrapper vor allem für Klassen der Java-Klassenbibliothek mit Zugriff auf Systemressourcen eingesetzt, um darin Sicherheitsmechanismen einzupflanzen, die ohne Mehraufwand bei der Anwendungsentwicklung zum Einsatzzeitpunkt individuell konfiguriert und zur Laufzeit ausgeführt werden können. Der Wrapping-Mechanismus ist damit der Grundstein für die in den folgenden Kapiteln beschriebenen Entwicklungen.

5 Konfigurierbare und erweiterbare Sicherheitsbibliothek

Die in diesem Kapitel vorgestellte Klassenbibliothek dient dazu, Anwendungen oder Komponenten ohne Berücksichtigung von Sicherheitsmechanismen entwickeln und diese abhängig von der Ausführungsumgebung bei der Installation aktivieren und konfigurieren zu können. Zu diesem Zweck wurden Klassen entwickelt, die statt der entsprechenden Klassen aus der Java-Klassenbibliothek [Sun00f] eingesetzt werden können und die anhand einer festgelegten Konfiguration Sicherheitsmechanismen zuschalten. Diese Bibliothek heißt daher *Java Security Enforcement and Configuration (JSEC)*.

Steht bei der Ausführung die erweiterte Java Virtual Machine aus Kapitel 4 zur Verfügung, so kann die Klassenersetzung durch Wrapping von Java-Bibliotheksklassen erfolgen wie in 5.4.3 beschrieben wird; Anwendungen und Komponenten können somit unter Benutzung der üblichen Java-Klassen entwickelt werden und nachträglich mit Sicherheitsmerkmalen ausgestattet werden.

Es gibt jedoch Einsatzszenarien, die eine veränderte Java Virtual Machine ausschließen, beispielsweise bei der Entwicklung von Applets, die von Browsern unbekannter Nutzer ausgeführt werden. Um dennoch die Vorteile der einfacheren Anwendungsentwicklung und der individuellen Konfigurierbarkeit nutzen zu können, ist ebenso eine Ersetzung der Java-Bibliotheksklassen durch die JSEC-Klassen im Quelltext der entwickelten Komponenten möglich. Diese Klassen bieten den Entwicklern Schnittstellen, die den ersetzten Klassen entsprechen, so daß bei der Komponentenentwicklung lediglich andere Klassen verwendet und keine zusätzlichen Parameter bereitgestellt werden müssen.

5.1 Zusammenhang mit der Java-Sicherheitsarchitektur

Anwendungen oder Komponenten müssen zur Berücksichtigung von Sicherheitsmerkmalen üblicherweise die verschiedenen Bibliotheken einsetzen, die für Java zur Verfügung stehen. Dies sind insbesondere die Java Cryptography Architecture (JCA, siehe 2.2.3.1), die Java Cryptography Extension (JCE, siehe 2.2.3.2), die Java Secure Socket Extension (JSSE, siehe 2.2.3.3) und der Java Authentication and Authorization Service (JAAS, siehe 2.2.3.5).

Komponentenentwickler müssen dazu neben der Fachlichkeit auch die Konzepte dieser Bibliotheken verstehen und ihre umfangreichen Schnittstellen korrekt einsetzen. Die Bibliotheken bieten zwar Erweiterungsmöglichkeiten und können mit unterschiedlichen Algorithmen

arbeiten; insbesondere bei der Initialisierung entstehen jedoch Abhängigkeiten von konkreten Algorithmen und erheblicher Aufwand bei der Bereitstellung und Speicherung von Parametern und Schlüsseln.

Die neue Bibliothek JSEC verbirgt die Schnittstellen der genannten Bibliotheken und bietet den Komponentenentwicklern eine Schnittstelle, die keine zusätzlichen Parameter erfordert und somit nur einen minimalen Mehraufwand bei der Implementierung erzeugt; alle nötigen Parameter werden aus der festgelegten Sicherheitsspezifikation berechnet.

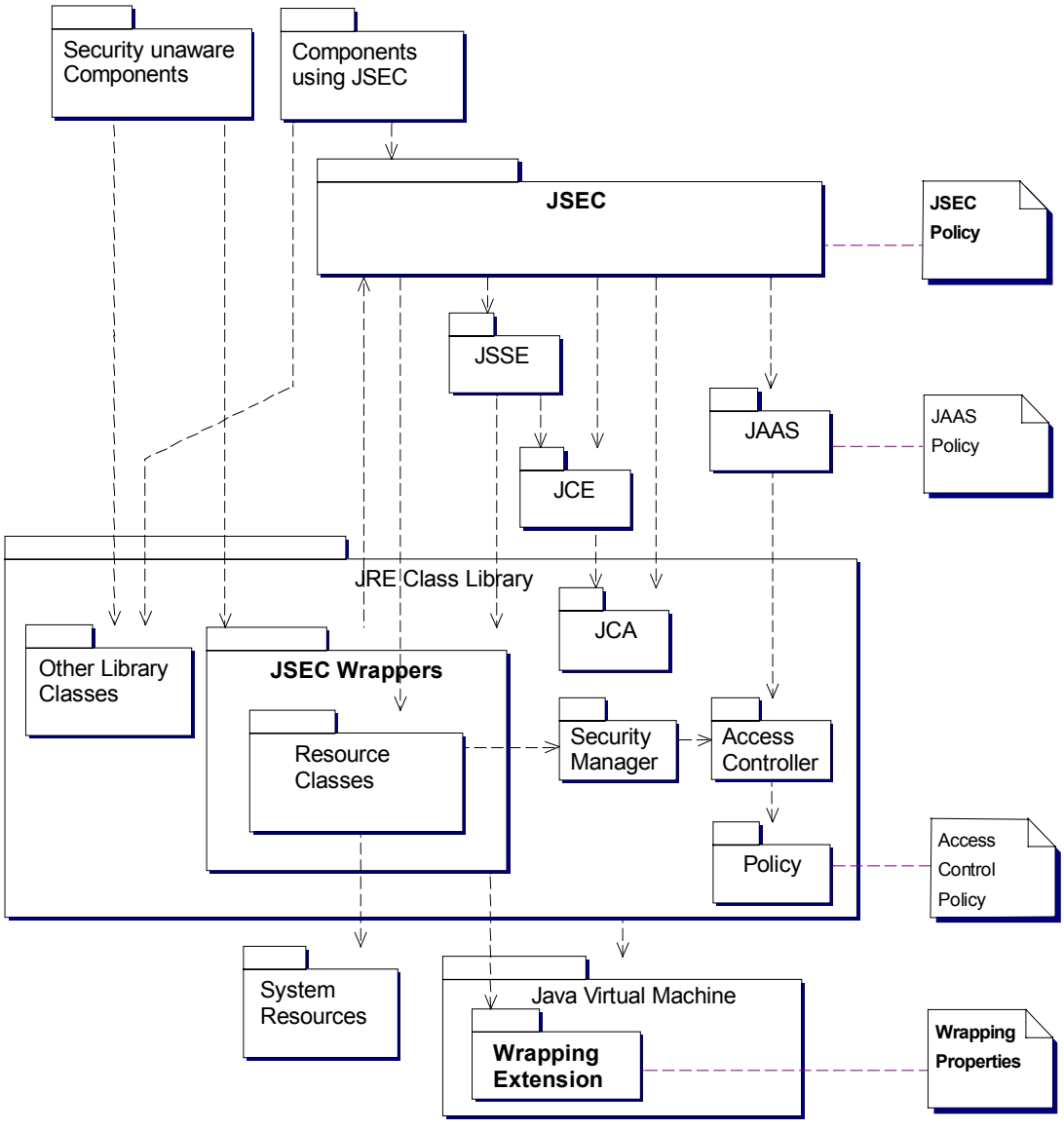

Abbildung 5-1: Einordnung von JSEC in die Java-Sicherheitsarchitektur

Die entstehende Architektur ist in Abbildung 5-1 dargestellt: In der Java-Klassenbibliothek ist die JCA enthalten, worauf die JCE aufbaut. JSSE verbindet die Sockets der Klassenbibliothek mit kryptographischen Algorithmen aus der JCE. Die Möglichkeiten des Access Controllers

werden von JAAS mit einer eigenen Policy-Schnittstelle erweitert. JSEC bedient die Schnittstellen dieser Bibliotheken anhand der Einstellungen aus der JSEC-Policy und bietet Anwendungskomponenten, die JSEC benutzen, eine einfache Schnittstelle.

Steht eine Java Virtual Machine mit Wrapping zur Verfügung, so können auch Komponenten mit Sicherheitsmerkmalen ausgestattet werden, die ohne Berücksichtigung von Sicherheit entwickelt wurden und lediglich die Java-Klassenbibliothek nutzen. JSEC-Wrapper umhüllen dann die Klassen aus der Klassenbibliothek, die auf Systemressourcen zugreifen und führen so auch bei deren Benutzung die JSEC-Klassen und damit die konfigurierten Sicherheitsmechanismen der einbezogenen Bibliotheken aus.

5.2 Übersicht zum Entwurf der JSEC-Bibliothek

JSEC (Java Security Enforcement and Configuration) bietet als Schnittstelle zu Anwendungskomponenten *Adapter*-Klassen, die statt entsprechender Klassen der Java-Klassenbibliothek für Ressourcenzugriffe benutzt werden können und dieselben Parameter erfordern. Die Adapter stellen also die Programmierschnittstelle von JSEC für Anwendungen und Komponenten dar oder werden von →Wrappern aufgerufen.

Anhand der konfigurierten Sicherheitsspezifikation werden die Ressourcenzugriffe von den Adaptern mit Sicherheitsmechanismen modifiziert. Diese Spezifikation wird im folgenden als JSEC Policy File bezeichnet, auch wenn sie nicht notwendigerweise in Form einer Datei vorliegen muß. Das JSEC Policy File enthält einerseits *Bedingungen*, unter denen Zugriffe modifiziert werden, und andererseits Spezifikationen der daraufhin anzuwendenden *Modifikationen*, also Sicherheitsmechanismen mit den dazu notwendigen Parametern.

- Eine *Bedingung* kann beispielsweise fordern, daß eine bestimmte Anwendung oder bestimmte Klassen mit einer angegebenen Signatur ausgeführt werden. Ähnlich wie beim erweiterten `grant`-Eintrag aus dem JAAS Policy File könnten die Mechanismen auch aufgrund des ausführenden Principals angewendet werden. Daneben sind aber auch weitere Bedingungen denkbar, beispielsweise eine Berücksichtigung der Tageszeit oder zusätzlicher Eigenschaften der Ausführungsumgebung.
- Beispiele für *Modifikationen* sind Ver- oder Entschlüsselung, Erzeugung oder Prüfung von Message Authentication Codes (MAC), Umleiten von Ressourcenzugriffen oder eine Aufzeichnung von Ereignissen in Logfiles. Neben unterschiedlichen Algorithmen sind hier auch grundsätzlich neue Arten von Modifikationen als Erweiterung denkbar.

Um solche veränderten Anforderungen ohne Neuimplementierung von JSEC berücksichtigen zu können, ist die Erweiterbarkeit ein wichtiges Ziel des Entwurfs. Im folgenden wird der JSEC-Entwurf vorgestellt und dabei insbesondere auch auf die berücksichtigten Erweiterungsmöglichkeiten hingewiesen.

Ein Adapter wird statt einer Klasse aus der Java-Klassenbibliothek zur Ausführung eines Ressourcenzugriffs aufgerufen und führt generell diese drei Schritte aus:

- *1. Schritt: ExecutionContext feststellen.* Der `ExecutionContext` enthält in einzelnen `ContextAspects` Informationen über den Zustand der Programmausführung, beispielsweise die ausgeführte Anwendung, die auf dem Stack vorhandenen Klassen und ihre Signaturen sowie die ausführenden Principals oder weitere Aspekte des Programmzustandes wie etwa die Tageszeit oder die Identität des Rechners.
- *2. Schritt: Engines erzeugen.* Eine Engine führt eine einzelne Modifikation eines Ressourcenzugriffs aus, beispielsweise eine Verschlüsselung mit einem bestimmten Algorithmus. Eine Beschreibung des vom Adapter ausgeführten Zugriffs und der festgestellte `ExecutionContext` werden mit den Bedingungen des JSEC Policy File verglichen und die dazu angegebenen Engines festgestellt.
- *3. Schritt: Engines anwenden.* Bei der Ausführung des Ressourcenzugriffs werden vom Adapter die Verarbeitungsmethoden der erzeugten Engines ausgeführt und so die konfigurierten Sicherheitsmechanismen angewendet.

JSEC ist zunächst eine implementierungsunabhängige Schnittstelle zur Ausführung dieser Schritte; diese Schnittstelle ist im Package `jsec` enthalten. Die konkrete Implementierung der einzelnen Bestandteile wird über Einträge in der Konfigurationsdatei `java.security` festgelegt.

Eine Standardimplementierung befindet sich im Package `edu.udo.jsec` und wird verwendet, wenn keine Angaben über eine andere gewünschte Implementierung gefunden werden. In diesem Package sind folgende Subpackages vorhanden:

- `policy` enthält eine Implementierung der JSEC Policy Files, die auf XML [BPS98] basiert und bereits einige Erweiterungsmöglichkeiten bietet; zudem kann durch eine Angabe in `java.security` eine andere Implementierung gewählt werden, so daß dieser Mechanismus ebenso wie bei der in Abschnitt 2.2.2.1 vorgestellten Access Control Policy austauschbar ist.
- `contexts` enthält konkrete Implementierungen verschiedener `ContextAspects`, die in einem JSEC Policy File als Bedingungen verwendet werden können und vom dazugehörigen `ContextGenerator` bei der Feststellung des `ExecutionContext` berücksichtigt werden. Alternative oder weitere Generatoren für andere Aspekte können in `java.security` eingestellt werden.
- `engines` enthält Klassen, die Sicherheitsmechanismen aus JCA, JCE, JSSE und JAAS anwenden und im JSEC Policy File konfiguriert werden können. Diese Engines sind unabhängig von den anderen Subpackages und können auch bei Nutzung anderer Adapter oder Policy-Implementierungen eingesetzt werden. Weitere Engines können separat implementiert und im JSEC Policy File benutzt werden.
- `adapters` enthält Ersatzklassen, die statt einiger Klassen der Java-Klassenbibliothek von Komponenten eingesetzt werden können. Diese Adapter führen die drei oben genannten

Schritte aus und benutzen dazu die implementierungsunabhängigen Klassen aus dem Package `jsec`, so daß sie auch mit anderen Implementierungen von `policy`, `context` und `engines` zusammenarbeiten können. Weitere Adapter können ebenfalls separat implementiert und von Anwendungskomponenten eingesetzt werden.

- `wrappers` enthält Wrapper für Klassen der Java-Klassenbibliothek. Sie rufen die Adapter aus dem Package `adapters` auf, so daß JSEC auch ohne Änderungen am Quellcode in bereits bestehende Anwendungen und Komponenten eingefügt werden kann. Dazu muß lediglich die erweiterte Virtual Machine aus Kapitel 4 zur Verfügung stehen und `wrapping.properties` um diese Wrapper erweitert werden.

In Abbildung 5-2 sind die Beziehungen dieser JSEC-Bestandteile untereinander und zu ihrem unmittelbaren Umfeld aus Abbildung 5-1 dargestellt. Der Entwurf des Gesamtsystems wird in den folgenden Abschnitten anhand der einzelnen Packages mit weiteren Details vorgestellt.

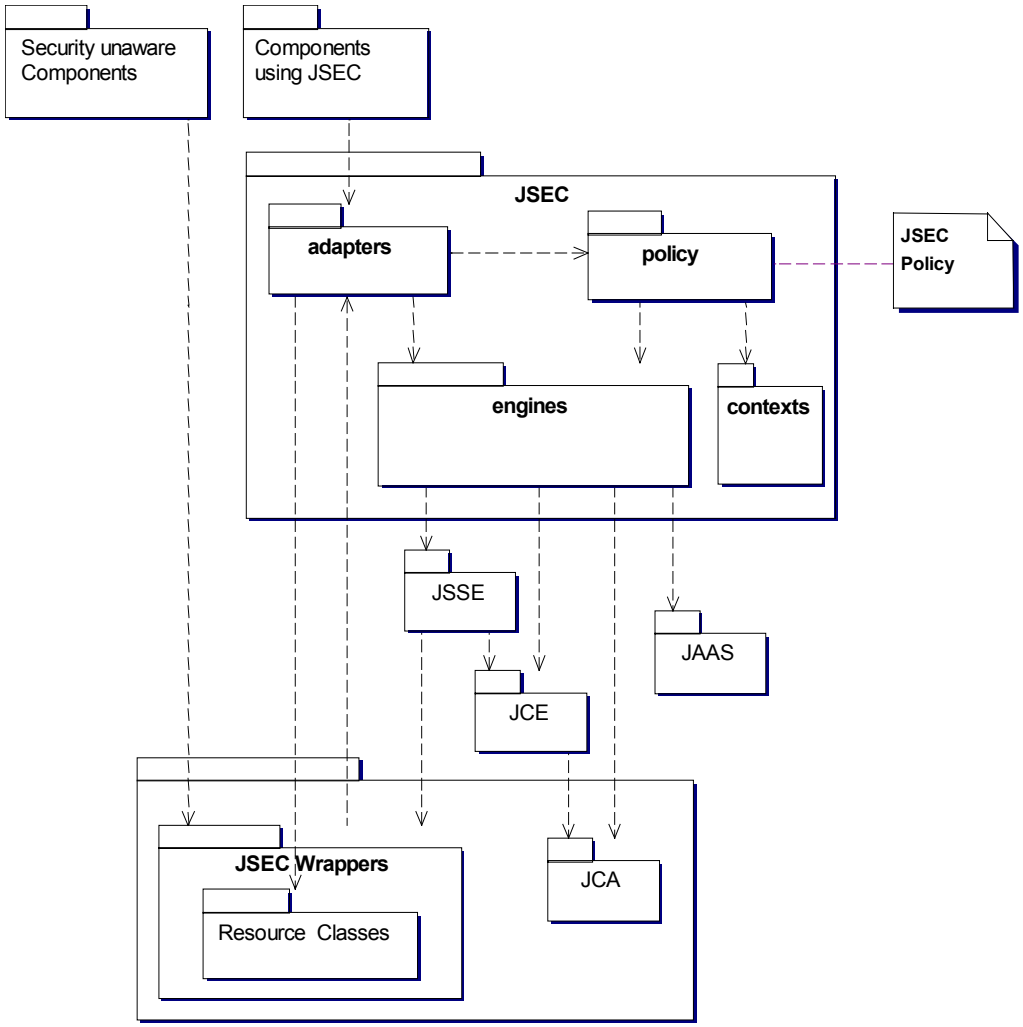

Abbildung 5-2: Interne und externe Beziehungen der JSEC-Bestandteile

5.3 Entwurf der allgemeinen Basis – Package jsec

Kern der JSEC-Bibliothek ist das Package `jsec`, das unabhängig von verschiedenen Implementierungen als gemeinsame Grundlage zur Verfügung steht und die drei in 5.2 vorgestellten Schritte unterstützt. Damit steht den Adapter-Klassen die JSEC-Funktionalität zur Verfügung, die in mehreren Implementierungen vorhanden sein kann. Das Package ist in Abbildung 5-3 als UML-Klassendiagramm [RJB98, FS97] dargestellt.

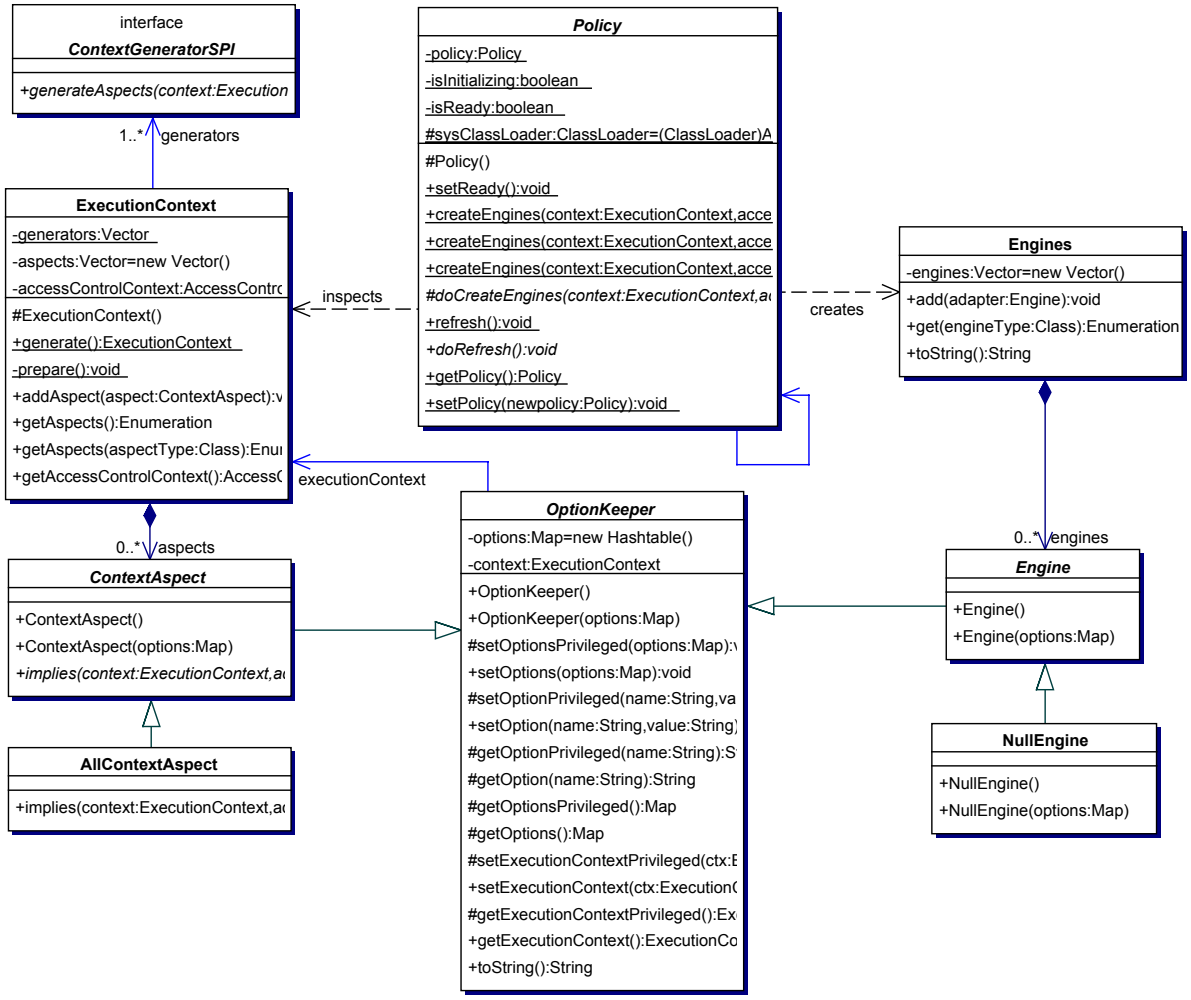

Abbildung 5-3: Das Package jsec mit der implementierungsunabhängigen JSEC-Schnittstelle

Neben der zentralen Klasse `Policy` sind in der Abbildung zwei weitere Bereiche rund um die Klassen `ExecutionContext` und `Engines` zu erkennen. Diese Bestandteile des Package `jsec` werden nun beschrieben.

5.3.1 ExecutionContext

Bei der Auswertung von Bedingungen im JSEC Policy File wird der Zustand der Ausführungsumgebung berücksichtigt. Dazu wird ein `ExecutionContext` erzeugt, der mehrere

verschiedene `ContextAspects` enthalten kann und damit den aktuellen Zustand der Programmausführung beschreibt.

Die verschiedenen ContextAspects werden von `ContextGenerator`en festgestellt, die ausgetauscht und erweitert werden können. So kann wie in den JAAS Policy Files berücksichtigt werden, welche Klassen ausgeführt werden und welcher Principal diese Klassen ausführt; dies wird in 5.4.4 vorgestellt. Die dazu notwendigen Parameter, beispielsweise Namen von zu prüfenden Klassen oder Principals, werden einem ContextAspect aus einem Policy File als Optionen übergeben, die von der Superklasse `OptionKeeper` verwaltet werden.

Zusätzlich bietet diese flexible Lösung aber auch die Möglichkeit, durch Konfigurationseinträge zukünftig weitere Aspekte zu berücksichtigen; so kann im Gegensatz zum Wechsel von Javas Policy Files (siehe 2.2.2.1) zu JAAS (siehe 2.2.3.5) eine Neuimplementierung der JSEC Policy Files verhindert werden.

Der von ContextGeneratoren mit ContextAspects gefüllte ExecutionContext wird an die Policy-Implementierung übergeben und ähnlich wie Permissions über die `implies`-Methode mit den ContextAspects aus dem JSEC Policy File verglichen. Welche Engines aus dem Policy File erzeugt werden, hängt also vom festgestellten ExecutionContext ab.

5.3.2 Policy

Ein Adapter ruft vor der Ausführung des Ressourcenzugriffs zunächst eine der folgenden Methoden der Klasse `Policy` auf, um die für diesen Zugriff konfigurierten `Engines` zu erhalten:

```
public static Engines createEngines(ExecutionContext context,
                                     Permission accessType)

public static Engines createEngines(ExecutionContext context,
                                     Permission accessType,
                                     Class engineType)

public static Engines createEngines(ExecutionContext context,
                                     Permission accessType,
                                     Class[] engineTypes)
```

Der erste Parameter bezeichnet den vom Adapter festgestellten `ExecutionContext`. Der Ressourcenzugriff wird als zweiter Parameter mit einer `Permission` spezifiziert, zu der in der Java-Klassenbibliothek für alle Ressourcenzugriffe bereits Subklassen existieren und weitere hinzugefügt werden können (siehe 2.2.2.1.1). Im JSEC Policy File sind als Bedingungen für die Anwendung von Engines ebenfalls Permissions spezifiziert, die über ihre `implies`-Methode mit dem aktuellen Zugriff verglichen werden und bei Übereinstimmung zur Erzeugung der Engines führen.

Als dritter Parameter kann eine Klasse oder eine Menge von Klassen angegeben werden, die als Engines zugelassen und vom Adapter aufgerufen werden können. Nur Engines dieser

Klassen oder Subklassen dieser Klassen werden erzeugt, auch wenn für diesen Ressourcenzugriff weitere Engines im JSEC Policy File spezifiziert wurden; denn ein Adapter kann nur Engine-Methoden aufrufen, deren Signaturen er kennt. Interfaces mit allgemein verwendbaren Signaturen für verschiedene Engines werden in 5.3.3.1 vorgestellt und können als Klassen an `createEngines` übergeben werden.

Die statischen `createEngines`-Methoden der abstrakten Klasse `Policy` überlassen die Erzeugung der Engines der konfigurierten `Policy`-Implementierung, die ihre eigenen JSEC Policy Files anhand der übergebenen Permission, des ExecutionContext und der geforderten Engine-Klassen untersucht, die spezifizierten Engines erzeugt und diese dem Adapter zur Verfügung stellt. Die Policy-Standardimplementierung wird in 5.4.1 vorgestellt und verarbeitet XML-Dokumente als Policy Files. Andere Implementierungen können jedoch auch andere Formate als XML und statt Dateien andere Quellen verwenden, beispielsweise eine Konfigurationsdatenbank.

5.3.3 Engines

Die erzeugten Engines werden in einem Container der Klasse `Engines` an den Adapter zurückgegeben. Die Methode `public Enumeration get(Class engineType)` bietet diesem eine weitere Möglichkeit zur Filterung bestimmter Engines in unterschiedlichen Zuständen; beispielsweise werden beim Öffnen einer Datei Engines berücksichtigt, die den Dateinamen verändern könnten, und beim Lesen der Daten solche, die den Datenstrom verändern.

Alle Engines sind Subklassen von `Engine` und damit auch von `OptionKeeper`, was ihnen die Verwaltung von benannten Optionen ermöglicht, die ebenfalls im JSEC Policy File konfiguriert werden können. Die Klasse `NullEngine` ist eine spezielle Engine, die keinerlei Modifikationen durchführt.

5.3.3.1 Engine Service Provider Interfaces – Package jsec.engines

Wie bereits in 5.3.2 erwähnt, muß ein Adapter die Signaturen der Methoden kennen, die in einer Engine zur Verfügung stehen. Daher sind in `jsec.engines` einige Service Provider Interfaces (SPI, siehe auch 2.2.3.1) definiert, die von konkreten Engines implementiert werden können. Adapter können passende Interfaces als verwendbare Engine-Klassen an `Policy` übergeben und erhalten so nur Engines mit Methoden, die vom Adapter anschließend aufgerufen werden können.

Die in Abbildung 5-4 aufgeführten Interfaces aus `jsec.engines` werden innerhalb der JSEC-Standardimplementierung von Adaptern eingesetzt und von Engines implementiert, was in 5.4.2 und 5.4.5 noch näher beschrieben wird. Diese Interfaces können jedoch lediglich eine Grundausstattung und keine vollständige Sammlung für alle denkbaren Engines darstellen. Sie sind dennoch innerhalb des Package `jsec` verankert, um eine gemeinsame Basis auch für neue Engines anderer Anbieter zu schaffen, so daß auch Adapter und Engines unterschiedlicher Hersteller zusammenarbeiten können, indem sie sich auf dieselben Interfaces beziehen.

Erstellt ein Anbieter A_1 weitere Interfaces, die zunächst nur von seinen eigenen Engines implementiert und von seinen eigenen Adaptern genutzt werden, so kann ein anderer Anbieter A_2 dies mit derselben Zielsetzung aber einer anderen Implementierung ebenfalls tun. Die Adapter von A_1 können somit trotz ähnlicher Funktionalität nicht mit den Engines von A_2 zusammenarbeiten und umgekehrt. In diesem Fall könnten Transformations-Engines entwickelt werden, die ein Interface von A_1 implementieren und die Methodenaufrufe an eine entsprechende Engine von A_2 weitergeben. So können Adapter von A_1 die gewohnten Interfaces anfordern und dennoch die Engines von A_2 nutzen; für den umgekehrten Fall müssen weitere Transformationen implementiert werden.

Abbildung 5-4: Das Package jsec.engines mit Service Provider Interfaces (SPI) für Engines

5.3.4 Auswahl konkreter Implementierungen

Das Package `jsec` bietet eine allgemeine Basis als Schnittstelle verschiedener Implementierungen der einzelnen Bestandteile von JSEC. So können Adapter, Policy, ContextGeneratoren, ContextAspects und Engines verschiedener Anbieter zusammenarbeiten, ohne sich zu kennen. Die konkreten Implementierungen der einzelnen Teile können wie folgt ausgewählt werden:

- Implementierte *Adapter* werden explizit in Anwendungskomponenten oder von Wrappern aufgerufen.
- Die *Policy*-Implementierung wird über den Klassennamen in `java.security` mit dem Parameter `jsec.policy.provider` festgelegt.
- Die Parameter für die konkreten *ContextGeneratoren* lauten `jsec.context.generator.1` (und 2, 3 usw.).
- Die *ContextAspects* werden von den dazugehörigen Generatoren bei der Feststellung des Programmzustandes instanziiert und können im JSEC Policy File in Bedingungen verwendet werden.

- Als Auswirkung einer Bedingung sind im JSEC Policy File die Klassennamen konkreter *Engines* definiert, die bei Zutreffen der Bedingung erzeugt werden.

Die bei einem Einsatz ohne Wrapping von Anwendungskomponenten aufgerufenen Adapter bilden die eigentliche Programmierschnittstelle von JSEC. Sie sind dennoch nicht Bestandteil der unabhängigen Schnittstelle im Package `jsec`, da weitere Adapter auch von anderen Herstellern hinzugefügt werden können, die zahlreiche andere Funktionen der Java-Klassenbibliothek oder weiterer Bibliotheken modifizieren könnten. Einige konkrete Adapter werden im Rahmen der Standardimplementierung in Abschnitt 5.4.2 vorgestellt.

5.4 Entwurf der Standardimplementierung – Package edu.udo.jsec

Das Package `edu.udo.jsec` enthält eine mögliche Implementierung der Konzepte aus 5.2. Abbildung 5-5 zeigt eine Übersicht der in `edu.udo.jsec` enthaltenen Subpackages, die nachfolgend beschrieben werden. Das Package `util` mit einigen mehrfach verwendeten Klassen und Methoden wird hier nicht genauer erläutert.

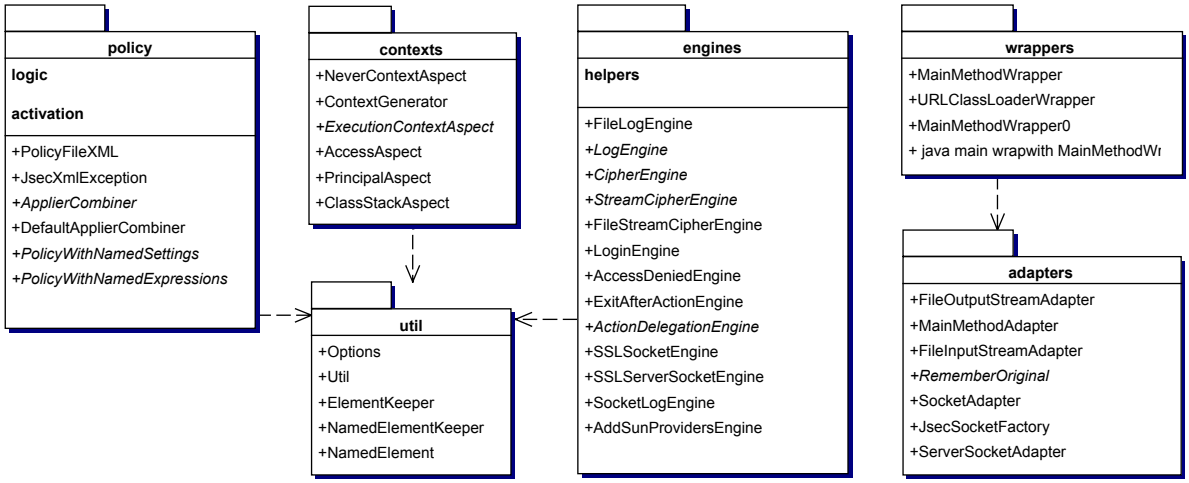

Abbildung 5-5: Bestandteile der Standardimplementierung in edu.udo.jsec

Die Bestandteile dieser Standardimplementierung können gemeinsam oder einzeln durch andere Lösungen ersetzt werden. Zudem können beispielsweise weitere Adapter und Engines von anderen Anbietern in eigenen Packages hinzugefügt werden und mit den bereits vorhandenen über die gemeinsame Schnittstelle `jsec` zusammenarbeiten. Zusätzliche Implementierungen könnten auch auf der Standardimplementierung aufbauen und so weitere Möglichkeiten hinzufügen.

5.4.1 JSEC-Policy in XML – Package edu.udo.jsec.policy

Ein JSEC Policy File enthält Regeln (*Rules*) zur Erzeugung von Engines. Jede Rule enthält eine Bedingung (*Condition*), die einen oder mehrere ContextAspects prüft. Wird die Bedin-

gung vom ExecutionContext erfüllt, so werden die in der *Implication* angegebenen Engines für den angegebenen Zugriffstyp erzeugt.

An JSEC Policy Files werden also grundsätzlich andere Anforderungen gestellt als an die bestehenden Access Control Policy Files oder JAAS Policy Files, weshalb deren Einsatz für JSEC nicht möglich ist.

Für die hier vorgestellte Implementierung der JSEC Policy Files wurde als Darstellungsform die *Extensible Markup Language (XML)* [BPS98] gewählt. So konnte auf die Implementierung eines Parsers verzichtet werden, da mit JAXP bereits eine XML-Bibliothek für Java existiert [Sun00e]. Hauptvorteil neben der einfachen Erweiterbarkeit ist die Verbreitung des XML-Standards, so daß beispielsweise bei der Entwicklung eines JSEC-Policy-Editors auch in anderen Programmiersprachen XML-Bibliotheken eingesetzt werden können, um eine einfache Transformation der Darstellung in entsprechende Strukturen im Hauptspeicher zu erhalten und umgekehrt. Zudem gibt es allgemeine Editoren, mit denen XML-Dokumente relativ komfortabel bearbeitet werden können und bei denen die nötigen Felder und mögliche Unterelemente in Eingabemasken angezeigt werden.

Die entwickelte *Document Type Definition (DTD)* für JSEC Policy Files befindet sich in Anhang A.2; in A.2.2 ist die Struktur auch graphisch dargestellt. Policy Files werden anhand dieser Definition von der XML-Bibliothek auf syntaktische Korrektheit geprüft und im Hauptspeicher als Baum von Objekten dargestellt, der bei der Policy-Auswertung durchlaufen werden kann.

5.4.1.1 Einführung am Beispiel

Grundelement der JSEC Policy Files in XML ist das Tag `<rule>`, das stets eine `<condition>` und eine `<implication>` enthält. Ein Policy File kann eine oder mehrere Rules enthalten. In Abbildung 5-6 ist ein Beispiel für ein XML Policy File mit nur einer Rule dargestellt:

- Die *Condition* enthält gemäß der DTD genau einen Operator, in diesem Fall das Tag `<access>`, das die Klasse einer Permission sowie deren Target-Namen und Actions als Parameter enthält, hier also eine `java.io.FilePermission "<<ALL FILES>>`[6]`", "read,write"`. Das `<access>`-Tag prüft, ob der ausgeführte Zugriff von der angegebenen Permission-Klasse abgedeckt wird; trifft die Condition zu, so wird die Implication ausgewertet.
- Eine *Implication* kann gemäß DTD einen oder mehrere Activators enthalten; im Beispiel ist lediglich ein `<apply>`-Tag angegeben, das eine `FileStreamCipherEngine` anwendet, also eine Engine zum Ver- und Entschlüsseln von Dateiströmen. In den `<option>`-Tags wird im Beispiel der DES-Algorithmus unter Benutzung des Schlüssels `mykey` aus dem Keystore vorgegeben.

[6] Die Zeichen < und > müssen in XML durch die Entities `&lt;` bzw. `&gt;` ersetzt werden, um Verwechslungen mit den Tag-Begrenzern auszuschließen.

```
<policy>
  <rule>
    <condition>
      <access class="java.io.FilePermission"
              target="&lt;&lt;ALL FILES&gt;&gt;" action="read,write"/>
    </condition>
    <implication>
      <apply class="edu.udo.jsec.engines.FileStreamCipherEngine">
        <option name="cipher.algorithm">DES</option>
        <option name="keystore.key.alias">mykey</option>
      </apply>
    </implication>
  </rule>
</policy>
```

Abbildung 5-6: Beispiel eines JSEC Policy File mit einer <rule>

Die Erzeugung der Engines läuft im Beispiel folgendermaßen ab: Eine Adapterklasse für `java.io.FileInputStream` fordert beim Lesen der Datei `c:\secret\file.txt` von der Policy alle Engines für diese Bedingungen:

- *Kontext:* Aktueller `ExecutionContext`, der von den konfigurierten ContextGeneratoren festgestellt wurde und beispielsweise auch PrincipalAspects mit den Principals des ausführenden Subjects enthält. In der Condition des Beispiels wird jedoch lediglich das `<access>`-Tag verwendet, so daß der ExecutionContext in diesem Fall unberücksichtigt bleibt.
- *Zugriff:* `java.io.FilePermission "c:\secret\file.txt", "read"`. Diese Spezifikation des Zugriffs wird von der Policy-Implementierung mit den in `<access>`-Tags definierten Permissions über die Methode `implies` verglichen (siehe 2.2.2.1.1); bei Übereinstimmung werden die in der Implication enthaltenen `<apply>`-Tags weiter untersucht.
- *Engine-Typen:* Vom Adapter für FileInputStream könnten `jsec.engines.FileNameEngineSPI` und `jsec.engines.FileInputStreamEngineSPI` angefordert werden. Das `<apply>`-Tag des Beispiels erfüllt diese Bedingung, da `FileStreamCipherEngine` das Interface `FileInputStreamEngineSPI` implementiert. Die Policy erzeugt schließlich diese Engine und übergibt ihr die angegeben Optionen.

Die in Abbildung 5-6 dargestellte Rule bewirkt also, daß alle Dateien, die ein Programm unter Benutzung eines Adapters oder eines Wrappers schreibt oder liest, mit dem DES-Algorithmus unter Verwendung eines Schlüssels namens `mykey` aus dem Keystore bearbeitet werden.

5.4.1.2 Weitere Merkmale

Wie an der DTD aus Anhang A.2 erkennbar, besteht ein Policy File grundsätzlich aus einem oder mehreren `<rule>`-Tags, die jeweils aus `<condition>` und `<implication>` bestehen.

Jede Condition enthält genau einen Operator; im Beispiel wurde der `<access>`-Operator verwendet. Wie oben bereits erwähnt, kann in der Condition jedoch auch eine Prüfung des aktuellen ExecutionContext durchgeführt werden. Dazu dient der Operator `<context>`; dieser prüft das Vorhandensein eines ContextAspect im ExecutionContext.

```
<context class="edu.udo.jsec.contexts.PrincipalAspect">
  <option name="class">com.sun.security.auth.NTUserPrincipal</option>
  <option name="name">Administrator</option>
</context>
```

Abbildung 5-7: Beispiel für <context>

In Abbildung 5-7 ist die Klasse des ContextAspect durch das Attribut `class` als `PrincipalAspect` angegeben. Diese Subklasse von ContextAspect prüft, ob ein bestimmter Principal über JAAS angemeldet ist; der Principal wird wie in JAAS üblich über eine Principal-Klasse und einen Namen definiert, die in `<option>`-Tags an den `PrincipalAspect` übergeben werden. Wurde also zuvor ein `NTLoginModule` verwendet, und ist in Windows NT der Administrator angemeldet, so wird das `<context>`-Tag des Beispiels erfüllt.

Es gibt zusätzlich Operatoren, die eine Definition logischer Ausdrücke ermöglichen, um eine Condition aus mehreren Bedingungen kombinieren zu können; dies sind `<and>`, `<or>`, `<not>` sowie `<true>` und `<false>`. Gemäß Ihrer Funktionalität enthalten Sie keinen, genau einen oder mehrere andere Operatoren.

```
<condition>
  <and>
    <context class="edu.udo.jsec.contexts.ClassStackAspect">
      <option name="class.1">com.acme.application.MainClass</option>
    </context>
    <context class="edu.udo.jsec.contexts.PrincipalAspect">
      <option name="class">com.acme.auth.Principal</option>
      <option name="name">user</option>
    </context>
    <not>
      <context class="edu.udo.jsec.contexts.PrincipalAspect">
        <option name="class">com.sun.security.auth.NTUserPrincipal</option>
        <option name="name">Administrator</option>
      </context>
    </not>
    <or>
      <access class="java.io.FilePermission"
              target="c:\secret\-" action="read,write"/>
      <access class="java.io.FilePermission"
              target="c:\applications\-" action="execute"/>
    </or>
  </and>
</condition>
```

Abbildung 5-8: Beispiel einer <condition> mit einem logischen Ausdruck

Eine Condition könnte damit wie in Abbildung 5-8 aufgebaut sein. Die dort dargestellte Bedingung ist wahr, wenn

- sich die Klasse `MainClass` auf dem Stack befindet *und*
- dem ausführenden Subject ein Principal der Klasse `com.acme.auth.Principal` mit dem Namen `user` zugeordnet ist *und*
- die Anwendung *nicht* vom Windows-NT-Administrator ausgeführt wird *und*
- auf eine Datei unter `c:\secret` schreibend oder lesend zugegriffen werden soll *oder* ein Programm unter `c:\applications` ausgeführt wird.

Um weitere, häufig benötigte Logikoperatoren, beispielsweise eine Implikation, ohne Änderung der DTD einführen zu können, steht der Operator `<op>` zur Verfügung. Als Attribut `class` wird der Name einer Klasse angegeben, die diesen Operator implementiert. Gemäß DTD können beliebig viele andere Operatoren innerhalb dieses Tags vorhanden sein; die implementierende Klasse ist dafür verantwortlich, die korrekte Anzahl von Kindern zu überprüfen. Abbildung 5-9 zeigt ein Beispiel des `<op>`-Tags, das innerhalb eines beliebigen Ausdrucks eingesetzt werden und selbst wieder andere Ausdrücke enthalten kann.

```
<op class="com.acme.logic.implication">
   <...>
   <...>
</op>
```

Abbildung 5-9: Beispiel für <op>

Zur Vermeidung mehrfacher Definition gleicher Conditions können diese als `<expression>` benannt werden. Eine Expression kann in jedem Ausdruck mit dem Operator `<evaluate>` wiederverwendet werden. Abbildung 5-10 zeigt ein Beispiel für die Verwendung dieser Möglichkeiten. Darin wird zunächst eine Expression `ntadmin` definiert, die lediglich einen Context-Operator zur Feststellung eines Windows-NT-Administrators enthält. Die zweite Expression `user_except_ntadmin` enthält einen logischen Ausdruck, der wahr ist, sobald der anwendungsdefninierte Principal `user` angemeldet ist, aber nicht der Windows-NT-Administrator; dabei wird die zuvor definierte Expression `ntadmin` mit einem `<evaluate>`-Tag wiederverwendet.

Wie Conditions können auch (Teil-)Implications benannt und wiederverwendet werden. Das Tag zur Definition heißt `<setting>`, und die Wiederverwendung innerhalb einer Implication oder einer anderen Setting wird mit `<activate>` vorgenommen. Dies ist in Abbildung 5-10 beispielhaft dargestellt: Die Setting `file_des_mykey` bestimmt, daß alle Dateizugriffe mit DES ver- und entschlüsselt werden und dazu der Schlüssel mit dem Namen `mykey` verwendet wird.

Die dargestellte Rule kann aufgrund dieser vordefinierten Makros in wenigen Zeilen definiert werden. Mit Expressions und Settings können also insbesondere umfangreiche Policies aus wiederverwendbaren Bestandteilen aufgebaut und damit deutlich übersichtlicher, wartungs-

freundlicher und kürzer definiert werden. Es können auch mehrere JSEC Policy Files benutzt werden, so daß gemeinsame Definitionen verschiedener Anwendungen in eine gemeinsame Datei ausgelagert werden können.

```
<policy>

   <expression id="ntadmin">
      <context class="edu.udo.jsec.contexts.PrincipalAspect">
         <option name="class">com.sun.security.auth.NTUserPrincipal</option>
         <option name="name">Administrator</option>
      </context>
   </expression>

   <expression id="user_except_ntadmin">
      <and>
         <context class="edu.udo.jsec.contexts.PrincipalAspect">
            <option name="class">com.acme.auth.Principal</option>
            <option name="name">user</option>
         </context>
         <not>
            <evaluate id="ntadmin"/>
         </not>
      </and>
   </expression>

   <setting id="file_des_mykey">
      <apply class="edu.udo.jsec.engines.FileStreamCipherEngine">
         <option name="cipher.algorithm">DES</option>
         <option name="keystore.key.alias">mykey</option>
      </apply>
   </setting>

   <rule>
      <condition>
         <and>
            <evaluate id="user_except_ntadmin"/>
            <access class="java.io.FilePermission"
                    target="c:\secret\-" action="read,write"/>
         </and>
      </condition>
      <implication>
         <activate id="file_des_mykey"/>
      </implication>
   </rule>

</policy>
```

Abbildung 5-10: Beispiel für <expression> und <evaluate> sowie <setting> und <activate>

5.4.1.3 Klassenentwurf

Das Klassengerüst zur Unterstützung der vorgestellten JSEC Policy Files in XML befindet sich im Package `edu.udo.jsec.policy` und basiert auf *Project X Technical Release 2*, dem zu Beginn der Entwicklung aktuellen Vorgänger der XML-Bibliothek *JAXP (Java API for XML Processing)* [Sun00e].

Zentraler Bestandteil dieses Package ist die Klasse `PolicyFileXML`, bei deren Instanziierung zunächst ein `XmlDocumentBuilder` und ein `ValidatingParser` aus der XML-Bibliothek erzeugt und damit die JSEC Policy Files gelesen und syntaktisch geprüft werden. Die folgenden XML-Dateien werden verarbeitet:

- Diejenige Datei, die mit dem Parameter `-Djsec.policy=filename.xml` beim Start der JVM angegeben wurde. Wird statt „=“ dabei „==“ benutzt, so wird keine weitere Datei gelesen.
- Alle Dateien, die mit `jsec.policy.url.1` (und 2, 3 usw.) in der Java-Konfigurationsdatei `java.security` angegeben wurden.

Dieses Vorgehen entspricht den herkömmlichen Java Access Control Policy Files. Als Dateiname kann jeweils auch eine http://-Adresse angegeben werden, so daß die Policy Files von einem Server bezogen werden können.

XML-Tag	Java-Klasse
<true>	edu.udo.jsec.policy.logic.True
<false>	edu.udo.jsec.policy.logic.False
<not>	edu.udo.jsec.policy.logic.Not
<and>	edu.udo.jsec.policy.logic.And
<or>	edu.udo.jsec.policy.logic.Or
<op>	edu.udo.jsec.policy.logic.Op
<context>	edu.udo.jsec.policy.logic.ContextEvaluator
<access>	edu.udo.jsec.policy.logic.AccessCheck
<expression>	edu.udo.jsec.policy.logic.NamedExpression
<evaluate>	edu.udo.jsec.policy.logic.NamedExpressionEvaluator
<apply>	edu.udo.jsec.policy.activation.Applier
<setting>	edu.udo.jsec.policy.activation.Setting
<activate>	edu.udo.jsec.policy.activation.SettingActivator
<rule>	edu.udo.jsec.policy.activation.Rule

Abbildung 5-11: Zuordnung der XML-Tags zu Java-Klassen

Der `XmlDocumentBuilder` wird vor dem Lesen so parametrisiert, daß die Objekthierarchie statt aus Standard-XML-Klassen aus speziellen Klassen erzeugt wird, die mit der in diesem Zusammenhang benötigen Funktionalität ausgestattet sind. Diese Zuordnung von XML-Tags zu Java-Klassen ist in Abbildung 5-11 dargestellt; die darin aufgeführten Klassen aus den Subpackages `logic` und `activation` werden in 5.4.1.3.1 und 5.4.1.3.2 beschrieben.

Nach dem Lesen der XML-Dokumente werden alle Vorkommen der drei möglichen Hauptelemente `<rule>`, `<expression>` und `<setting>` auf die Container `rules`, `expressions` und `settings` aus Abbildung 5-12 verteilt.

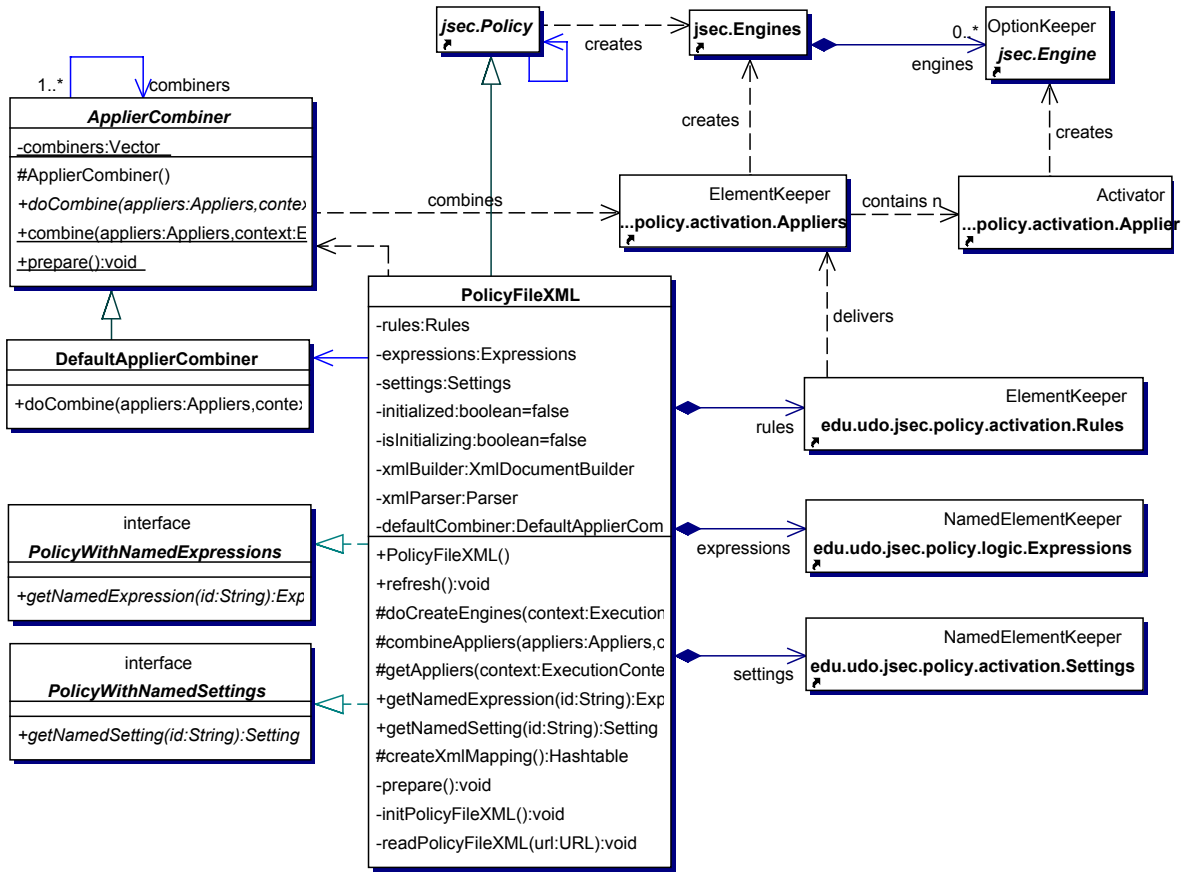

Abbildung 5-12: Klassendiagramm zum Package edu.udo.jsec.policy

`PolicyFileXML` stellt als konkrete Subklasse der abstrakten Klasse `jsec.Policy` insbesondere eine Implementierung dieser Methode zur Verfügung:

```
protected Engines doCreateEngines( ExecutionContext context,
                                   Permission accessType,
                                   Class[] engineTypes)
```

Die Methode stellt die Hauptschnittstelle der Policy-Implementierung dar. Zunächst ruft `doCreateEngines` den Container `rules` auf und erhält die zum ExecutionContext und zur Permission passenden `Applier`-Objekte; dieser Ablauf ist in 5.4.1.3.2 genauer beschrieben.

Jeder `Applier` enthält die Informationen zur Erzeugung einer Engine; die von `rules` gelieferten Applier entsprechen also den Engines, die aufgrund der Policy erzeugt werden sollen. Unter Umständen treffen in einem Policy File aber mehrere Conditions zu, die mehrere gleichartige Engines erzeugen; diese Engines werden dann nacheinander ausgeführt. In einigen Fällen kann dies akzeptiert werden, beispielsweise beim mehrfachen Verschlüsseln oder Erzeugen von Message Authentication Codes (MAC) – wenn auch beim Entschlüsseln bzw. Prüfen des MAC wieder alle entsprechenden Engines erzeugt werden.

In einigen Fällen könnte die mehrfache Anwendung gleichartiger Engines oder eine unglückliche Kombination aber zu Problemen führen. Daher werden vor der Erzeugung der Engines

ApplierCombiner ausgeführt, die ungültige Kombinationen korrigieren können. In java.security können mit den Parametern jsec.combiner.provider.1 (und 2, 3 usw.) ApplierCombiner definiert werden. Ohne einen solchen Eintrag wird der DefaultApplierCombiner ausgeführt, der lediglich Applier entfernt, dessen Engines nicht zu den geforderten Engine-Klassen passen.

Stehen die Applier schließlich fest, so erzeugen sie die Engines, die als Rückgabewert der Methode doCreateEngines an den aufrufenden Adapter ausgeliefert und von ihm anschließend aufgerufen werden.

Die Methode public void refresh() ist die zweite externe Schnittstelle von PolicyFileXML. Sie führt das Lesen der XML-Dokumente erneut aus, so daß Änderungen an den Dateien beim nächsten doCreateEngines berücksichtigt werden.

5.4.1.3.1 Logikelemente für Conditions – Package edu.udo.jsec.policy.logic

Die Klassen im Package edu.udo.jsec.policy.logic übernehmen die Auswertung der logischen Ausdrücke, die in einer Condition verwendet werden können und sind in Abbildung 5-13 dargestellt. Die abstrakte Superklasse Expression gibt die Methode public boolean evaluate (ExecutionContext context, Permission accessType) vor, die als Rückgabewert das Funktionsergebnis der jeweiligen Expression liefert. And, Or und Not rufen die evaluate-Methoden ihrer children auf und bilden das Ergebnis der Verknüpfungsoperation. True und False liefern stets den entsprechenden Wahrheitswert.

Die Klasse Op dient wie in 5.4.1.2 beschrieben zur flexiblen Integration weiterer Logikoperatoren und erzeugt diesen Operator anhand des Klassennamens; sie ruft dessen evaluate-Methode auf und gibt deren Ergebnis als ihr eigenes zurück. Der erzeugte Operator muß bei seiner Ausführung auf die Kinder des ursprünglich in der Objekthierarchie vorhandenen Op-Objekts zugreifen. Um ein Kopieren der Kinder zu vermeiden, muß die von Op erzeugte Operation eine Subklasse von ProxyExpression sein, der eine abweichende Quelle für Kinder übergeben werden kann.

Zum Verwenden von benannten Expressions wird das Tag <evaluate> bzw. die Klasse NamedExpressionEvaluator eingesetzt; diese Klasse erwartet eine systemweite Policy, die das Interface PolicyWithNamedExpressions implementiert. PolicyFileXML aus Abbildung 5-12 implementiert dieses Interface und nutzt die Klasse Expressions zum Auffinden der benannten Ausdrücke aller <expression>-Tags, so daß diese von NamedExpressionEvaluator ausgewertet werden kann.

AccessCheck entspricht dem Tag <access> und nutzt zur Auswertung des aktuellen Zugriffs die Klasse AccessContext aus edu.udo.jsec.contexts, die in 5.4.4 vorgestellt wird. ContextEvaluator entspricht dem Tag <context> und nutzt einen beliebigen ContextAspect zum Prüfen des ExecutionContext.

Objekte aller konkreten Expression-Subklassen aus edu.udo.jsec.policy.logic werden gemäß Abbildung 5-11 aus dem XML-Dokument als Knoten der Objekthierarchie automatisch erzeugt und können so sehr einfach auf ihre Kinder zugreifen, die wiederum

tisch erzeugt und können so sehr einfach auf ihre Kinder zugreifen, die wiederum Expressions sind.

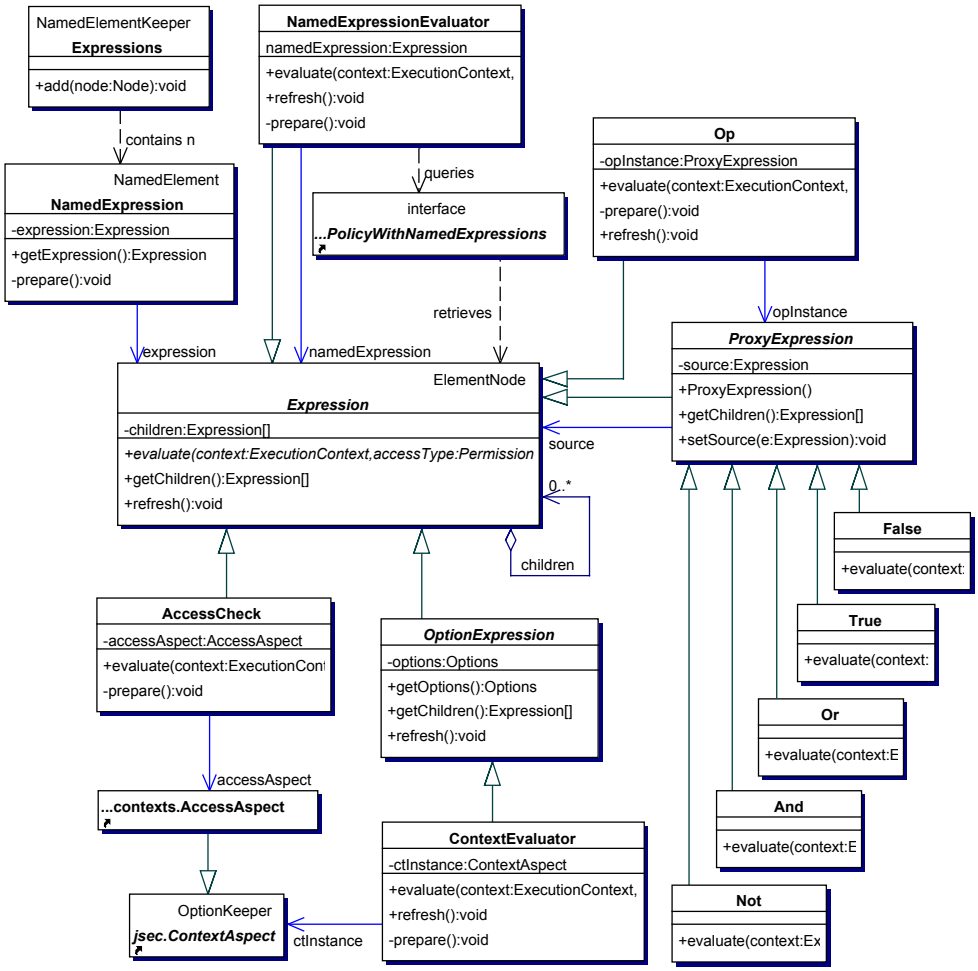

Abbildung 5-13: Klassendiagramm zum Package edu.udo.jsec.policy.logic

5.4.1.3.2 Aktivierung der Engines aus Rules – Package edu.udo.jsec.policy.activation

Die Hauptaufgabe der XML-Policy besteht darin, die passenden Engines aus der XML-Objekthierarchie zu ermitteln. Diese Aufgabe wird mit Hilfe der Klassen in `edu.udo.jsec.policy.activation` erfüllt. In Abbildung 5-14 sind die Klassen `Settings` und `Rules` dargestellt, die von `PolicyFileXML` in Abbildung 5-12 zur Ablage der entsprechenden aus XML erzeugten Objekte der Klassen `Rule` und `Setting` genutzt werden.

`Rules` dient auch zum Ermitteln der zu ExecutionContext und Permission passenden `Appliers`, aus denen anschließend die Engines erzeugt werden. Rules ruft dazu alle vorhandenen `Rule`-Objekte auf, die den Funktionswert ihrer Condition wie in 5.4.1.3.1 beschrieben auswerten lassen.

Ist eine Bedingung erfüllt, so werden alle durch die Implication festgelegten `Appliers` der Rückgabemenge hinzugefügt. Die Appliers können entweder direkt mit dem `<apply>`-Tag angegeben werden, welches der Klasse `Applier` entspricht, oder indirekt durch `<activate>`; die entsprechende Klasse `SettingActivator` bezieht ihre Applier aus einer über den Namen referenzierten `Setting`.

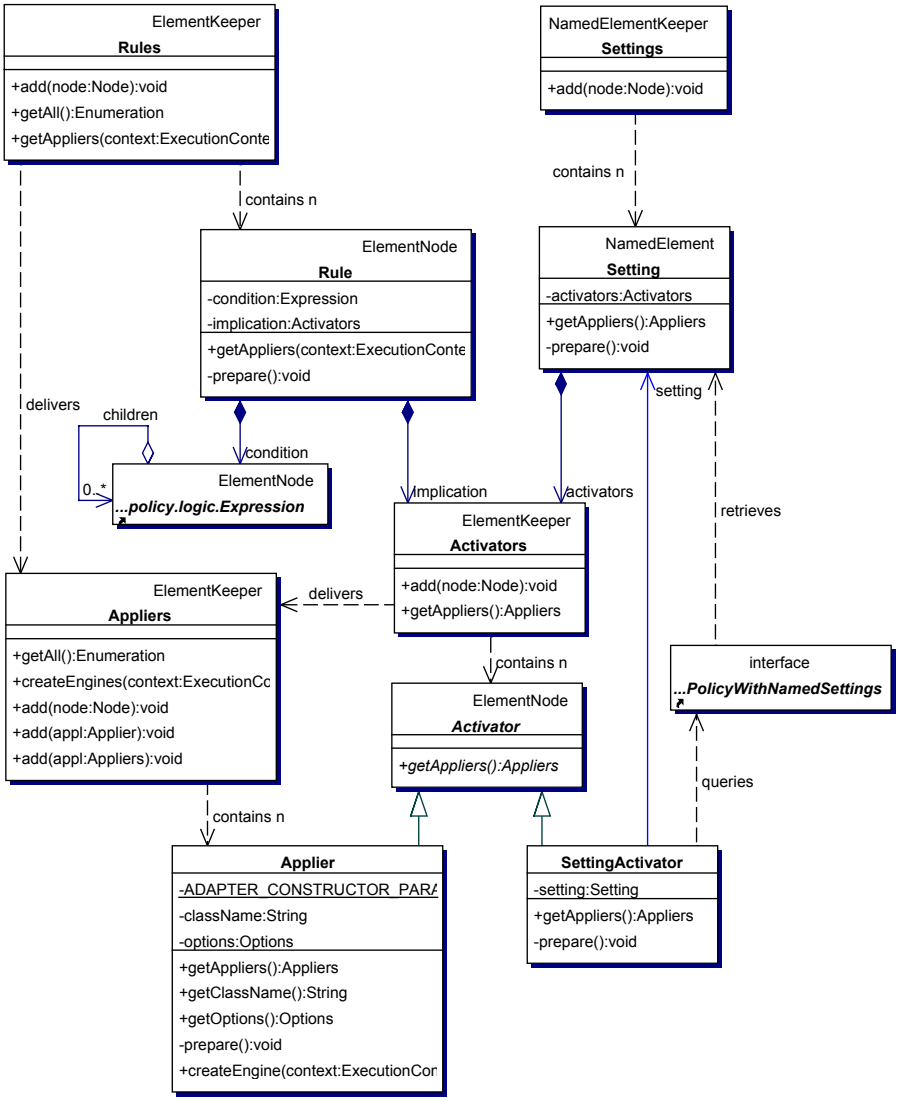

Abbildung 5-14: Klassendiagramm zum Package edu.udo.jsec.policy.activation

`SettingActivator` erwartet eine systemweite Policy, die das Interface `PolicyWithNamedSettings` implementiert, um von ihr die benannte Setting mit den darin definierten `<apply>`-Tags oder weiteren `<activate>`-Tags zu erhalten. `PolicyFileXML` implementiert auch dieses Interface und bezieht die Setting aus der Klasse `Settings`. Vorteil dieser indirekten Kopplung ist die mögliche Zusammenarbeit der Klassen aus `edu.udo.jsec.policy.activation` und `edu.udo.jsec.policy.logic` mit anderen Policy-Implementierungen, die `PolicyWithNamedSettings` und `PolicyWithNamedExpressions` implementieren.

5.4.2 Adapter als API für JSEC – Package edu.udo.jsec.adapters

Die in 5.4.1 vorgestellte JSEC-Policy ermöglicht die Konfiguration von Sicherheitsmechanismen, die unter definierten Bedingungen ausgeführt werden und damit nicht bei der Anwendungsentwicklung implementiert werden müssen. Um die JSEC-Funktionalität in Anwendungen oder Komponenten zu nutzen, ist lediglich die Nutzung der in diesem Abschnitt beschriebenen Adapter statt der ursprünglichen Klassen für Ressourcenzugriffe aus der Java-Klassenbibliothek nötig. Die Adapter führen den gewünschten Zugriff aus und berücksichtigen zusätzlich alle passenden Engines, die in der JSEC-Policy festgelegt sind.

Die Adapter stellen also das *Application Programming Interface (API)* von JSEC dar und können mit denselben Parametern initialisiert werden wie die von ihnen ersetzten Klassen; der Aufwand zur Änderung einer Anwendung beschränkt sich also auf eine Ersetzung der Klassennamen. JSEC kann auch vollständig ohne Modifikationen in bestehende Anwendungen integriert werden, wenn eine Java Virtual Machine mit der in Kapitel 4 vorgestellten Wrapping-Erweiterung zur Verfügung steht. Dazu existieren in JSEC Wrapper-Klassen, die in 5.4.3 vorgestellt werden.

Aus Aufwandsgründen konnten im Rahmen dieser Arbeit nicht für alle in der Java-Klassenbibliothek enthaltenen Klassen mit Ressourcenzugriff Adapter und Wrapper entwikkelt werden; weitere können jedoch unabhängig hinzugefügt werden, wobei es zur Zusammenarbeit mit passenden Engines notwendig sein dürfte, weitere Service Provider Interfaces (SPI) für Engines zu vereinbaren (siehe auch 5.3.3.1). In der Standardimplementierung stehen die nachfolgend beschriebenen Adapter aus Abbildung 5-16 zur Verfügung.

5.4.2.1 MainMethodAdapter

Die Klasse `MainMethodAdapter` dient dazu, konfigurierte Aktionen vor dem Start der Anwendung durchzuführen, beispielsweise ein Login des Benutzers. Dazu kann sie in der Kommandozeile statt der Anwendungsklasse mit der `main`-Methode benutzt werden. Lautet der Programmaufruf ursprünglich zum Beispiel

```
java  package.MainClass  p1 p2 p3,
```

so muß dieses Kommando lediglich durch

```
java  edu.udo.jsec.adapters.MainMethodAdapter  package.MainClass  p1 p2 p3
```

ersetzt werden. Der Klassenname der ursprünglichen Startklasse wird damit als erster Parameter an den `MainMethodAdapter` übergeben, der diese Klasse lädt und mit den restlichen Parametern startet, nachdem alle in der JSEC-Policy für den Programmstart vorgesehenen Aktionen durchgeführt wurden.

Der `MainMethodAdapter` kann statt dessen auch im Quelltext benutzt werden und zu Beginn der `main`-Methode einer Anwendung aufgerufen werden. Als Parameter werden dann einerseits die Kommandozeilenparameter erwartet, die auch der `main`-Methode zur Verfügung stehen, und andererseits die Namen einer Klasse und einer darin enthaltenen statischen

Methode, die vom Adapter anschließend wie eine `main`-Methode mit den Kommandozeilenparametern aufgerufen wird. Die Programmausführung darf nicht nach Ausführung des Adapters gestartet werden, sondern muß in dieser statischen Methode implementiert sein, damit die Ausführung innerhalb von `doPrivileged` aus 2.2.2.2.3 oder `doAs` aus 2.2.3.5 möglich ist, was etwa für eine Login-Engine notwendig ist.

Die Programmstruktur stellt sich damit wie Abbildung 5-15 dar. Wie bei allen Adaptern kann auf diese Änderung verzichtet werden, wenn Wrapping zur Verfügung steht.

```
public class ApplicationStart {

   public static void main(String[] args) {
      new edu.udo.jsec.adapters.MainMethodAdapter().main(
         ApplicationStart.class, "oldMainMethod", argv);
   }

   public static void oldMainMethod(String[] args) {
      // Application executes here
   }
}
```

Abbildung 5-15: Veränderte main-Methode beim Einsatz eines MainMethodAdapter

`MainMethodAdapter` fordert von der JSEC-Policy alle Engines an, die für die Zugriffsbeschreibung `JsecPermission "Name der Main-Klasse", "executeMain"` konfiguriert sind und folgende aus Abbildung 5-4 bekannten Service Provider Interfaces (SPI) implementieren:

- Konfigurierte Engines, die das SPI `AccessControlEngineSPI` implementieren, werden mit der genannten `JsecPermission` aufgerufen, um ihnen die Möglichkeit zur Zugriffskontrolle für den Start dieser Anwendung zu geben.
- `MainMethodEngineSPI`-Engines können die Kommandozeilenparameter verändern.
- Alle `ActionEngineSPI` werden zunächst durch Aufruf von `setDelegation` verkettet und anschließend mit der Ausführung einer `PrivilegedAction` beauftragt, die die angegebene `main`-Methode ausführt. Diese PrivilegedAction wird innerhalb der verketteten Engines nach Ausführung ihrer Funktionalität durch Aufrufe von `doAction` weitergereicht und kann dabei je nach Engine um `doAs` oder andere Schachtelungen erweitert werden.

Engines aus der JSEC-Standardimplementierung, die diese Interfaces implementieren und daher in Verbindung mit `MainMethodAdapter` konfiguriert werden können, werden in den Abschnitten 5.4.5.1 bis 5.4.5.5 vorgestellt. Weitere Engines können implementiert werden und mit einem Adapter zusammenarbeiten, wenn sie die Service Provider Interfaces unterstützen.

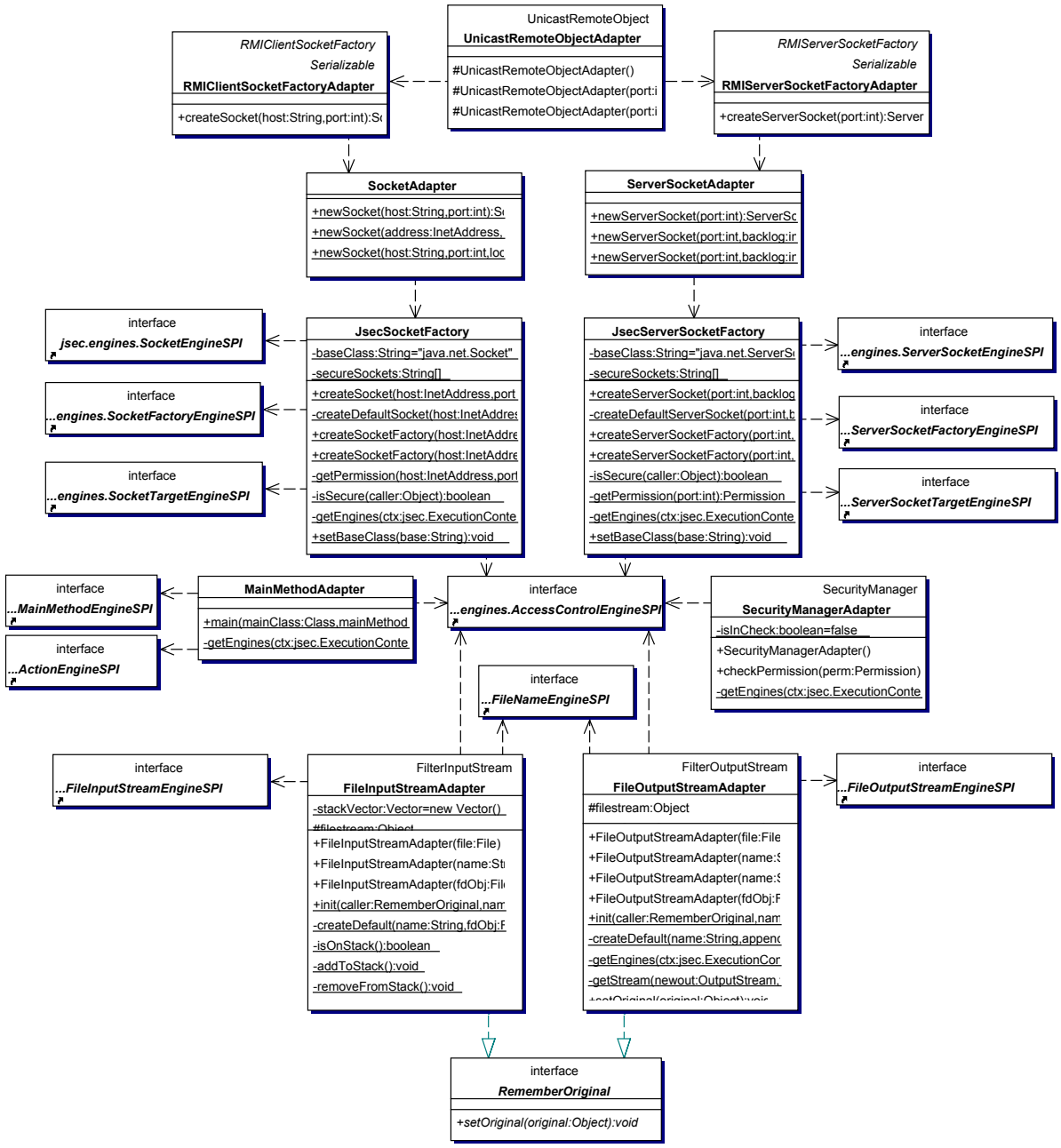

Abbildung 5-16: Klassendiagramm zum Package edu.udo.jsec.adapters

5.4.2.2 SecurityManagerAdapter

Der `SecurityManagerAdapter` erbt von `java.lang.SecurityManager` (siehe auch 2.2.2.2) und definiert die Methode `checkPermission` neu. Statt `java.security.AccessController` aufzurufen, werden alle `AccessControlEngineSPI` für die zu prüfende Permission und den aktuellen Zustand der Laufzeitumgebung aus der JSEC-Policy angefordert und aufgerufen. Alle dafür konfigurierten Engines können so entscheiden, ob sie den Zugriff gestatten oder verbieten. Ebenso könnten sie die geprüften Permissions protokollieren.

Einige implementierte Engines zum Einsatz in Verbindung mit dem `SecurityManager-Adapter` werden in 5.4.5.4 und 5.4.5.5 vorgestellt.

5.4.2.3 SocketAdapter und ServerSocketAdapter

Die Klassen `SocketAdapter` und `ServerSocketAdapter` können zur Erzeugung von Sockets auf der Client-Seite und analog auf der Server-Seite eingesetzt werden. Statt `s = new Socket(...)` wird im Quelltext dazu `s = SocketAdapter.newSocket(...)` verwendet, wobei alle bei den ursprünglichen Socket-Konstruktoren möglichen Parameter übergeben werden können.

Diese Adapter nutzen die `JsecSocketFactory` und `JsecServerSocketFacctory`, die anhand der JSEC-Policy feststellen, welche Engine zur Erzeugung des Sockets für die gewünschte Netzwerkverbindung eingesetzt werden soll.

Wie in Abbildung 5-16 dargestellt, werden die folgenden Service Provider Interfaces beim Aufruf der Engines unterstützt. Die entsprechenden Server-seitigen Engines werden hier nicht explizit erwähnt.

- `SocketTargetEngineSPI`-Engines werden aufgerufen und haben die Möglichkeit, das Verbindungsziel für das Socket zu verändern oder zu protokollieren.
- `AccessControlEngineSPI` können Zugriffskontrollentscheidungen treffen und die Herstellung der Verbindung verhindern.
- Eine `SocketFactoryEngineSPI`-Engine kann eine `SocketFactory` festlegen, mit denen das Socket erzeugt werden soll. Da zur Erzeugung eines Sockets nur eine Factory verwendet werden kann, darf in der JSEC-Policy nicht mehr als eine solche Engine für den aktuellen Verbindungswunsch konfiguriert sein; andernfalls werfen diese Adapter eine `SecurityException`, die zum Abbruch des Programms führt. In der Standardimplementierung der JSEC-Policy können `ApplierCombiner` (siehe 5.4.1.3) integriert werden, um solche Situationen bereits vorher aufzulösen.
- Wird ebenfalls eine `SocketEngineSPI` konfiguriert, so wird die Factory dieser Engine übergeben, um das Socket zu erzeugen und eventuell anschließend weiter zu konfigurieren. Auch diese Engine darf für jede Verbindung nur ein einziges Mal in der Policy vorhanden sein. `SocketEngineSPI` und `SocketFactoryEngineSPI` können getrennt implementiert werden, aber es ist ebenso möglich, beide Interfaces in einer Engine zu vereinen.

Nachdem die Sockets initialisiert wurden, greift JSEC in die Client-Server-Kommunikation nicht mehr ein, so daß ein Laufzeitnachteil zunächst nur beim Aufbau einer Verbindung entsteht, bei dem ohnehin Latenzzeiten auftreten. Die von den Engines erzeugten Sockets können anschließend jedoch eine Verschlüsselung oder andere Mechanismen ausführen; der damit während der Datenübertragung über eine gesicherte Verbindung auftretende Geschwindigkeitsverlust hängt also von der eingesetzten Socket-Implementierung ab und würde auch bei deren Einsatz ohne JSEC auftreten.

In 5.4.5.4 bis 5.4.5.7 werden Engines beschrieben, die diese Interfaces unterstützen und in der Standardimplementierung enthalten sind.

5.4.2.4 UnicastRemoteObject-, RMIClient- und RMIServerSocketFactoryAdapter

Auch für *Remote Method Invocation (RMI)* werden Sockets und ServerSockets benutzt, die allerdings nicht vom Anwendungsentwickler, sondern von der RMI-Bibliothek geöffnet werden. Daher können die Klassen `SocketAdapter` und `ServerSocketAdapter` aus 5.4.2.3 für RMI nicht unmittelbar eingesetzt werden.

RMI bietet aber die Möglichkeit, eine `RMIClientSocketFactory` und `RMIServerSocketFactory` zur Erzeugung der Sockets festzulegen. Klassen, deren Objekte per RMI verfügbar gemacht werden sollen, erben von `UnicastRemoteObject`. Wird statt dessen der `UnicastRemoteObjectAdapter` als Superklasse der Server-Klassen verwendet, so setzt diese zur Erzeugung der Sockets die Adapter `RMIClientSocketFactoryAdapter` und `RMIServerSocketFactoryAdapter` ein.

Diese Adapter nutzen `SocketAdapter` und `ServerSocketAdapter`, um Sockets anhand der Konfiguration aus der JSEC-Policy zu erzeugen. Da `RMIClientSocketFactoryAdapter` auf der Client-Seite der RMI-Verbindung ausgeführt wird, entsprechen die dort erzeugten Sockets der auf dem Client festgelegten JSEC-Policy. Die Konfigurationen von Server und Client müssen jedoch zusammenpassen, um eine RMI-Verbindung zwischen passenden Sockets herstellen zu können; dies trifft jedoch für Socket-Verbindungen ohne RMI ebenfalls zu.

Mit den genannten Mechanismen können RMI-Methodenaufrufe über gesicherte Socket-Verbindungen ausgeführt werden. Dies gilt jedoch nicht für Verbindungen mit der RMI-Registry, die zum Auffinden von Server-Objekten nötig sind. Da RMI hier keine Schnittstelle für andere Socket-Implementierungen zu bieten scheint, hätten diese Teile von RMI in entsprechenden Adaptern vollständig neu implementiert werden müssen. Zudem wird in 5.4.3.3 deutlich, daß bei einem Einsatz von Wrapping auch diese Verbindungen gesichert werden können.

5.4.2.5 FileInputStreamAdapter und FileOutputStreamAdapter

`FileInputStreamAdapter` und `FileOutputStreamAdapter` aus Abbildung 5-16 dienen zur Veränderung von Dateiströmen. Sie bieten alle Konstruktoren und Methoden der Klassen `java.io.FileInputStream` und `java.io.FileOutputStream`, so daß die Verwendung dieser Adapter lediglich eine Änderung der Klassennamen voraussetzt.

Diese Adapter erben allerdings nicht von den genannten Klassen, sondern von `FilterInputStream` bzw. `FilterOutputStream`, um damit eine Ineinanderschachtelung mehrerer anderer Streams zu ermöglichen, die von mehreren Engines erzeugt werden und den Datenstrom jeweils auf ihre Weise verändern. Aus diesem Grund können die Adapter nicht

verwendet werden, wenn beispielsweise ein `FileInputStream` als Parameter benötigt wird; dies wäre mit Subklassen dieser Klassen durch Polymorphie möglich.

Allerdings kann davon ausgegangen werden, daß üblicherweise kein `FileInputStream`, sondern ein Objekt der Superklasse `InputStream` erwartet wird; diese Vermutung wird durch die API-Spezifikation der Java-Klassenbibliothek [Sun00f] bestätigt, in der keine einzige Verwendung von `FileInputStream` oder `FileOutputStream` in Schnittstellen der Klassenbibliothek dokumentiert ist.

`FileInputStreamAdapter` und `FileOutputStreamAdapter` arbeiten mit Engines zusammen, die folgende Interfaces implementieren:

- `FileNameEngineSPI`-Engines könnten den Dateinamen vor dem Zugriff auf die Datei verändern, um eine Umlenkung der Zugriffe in andere Bereiche des Dateisystems zu erreichen, oder die Dateinamen protokollieren.
- Jede `AccessControlEngineSPI` wird mit einer dem gewünschten Zugriff entsprechenden `FilePermission` aufgerufen, um Zugriffskontrollentscheidungen zu unterstützen. Da für den eigentlichen Zugriff ein `FileInputStream` bzw. `FileOutputStream` eingesetzt werden, kommt zusätzlich der in 2.2.2 vorgestellte Java-Zugriffskontrollmechanismus mit Security Manager zum Einsatz.
- Alle für den aktuellen Zugriff konfigurierten `FileInputStreamEngineSPI` bzw. `FileOutputStreamEngineSPI` können eigene Streams um den tatsächlichen Datei-Strom schachteln, um beispielsweise eine Ver- oder Entschlüsselungen durchzuführen.

Durch die einmalige Aktivierung der Streams muß JSEC nach der Initialisierung beim Öffnen einer Datei nicht mehr eingreifen, sondern überläßt die Bearbeitung des Datenstroms den Streams, die von den konfigurierten Engines erzeugt werden. Daher ergibt sich das Laufzeitverhalten beim Lesen und Schreiben von Dateiströmen ebenfalls aus den verwendeten Stream-Implementierungen.

In 5.4.5.4, 5.4.5.5, 5.4.5.7 und 5.4.5.8 werden Engines der Standardimplementierung zu diesen Interfaces beschrieben.

5.4.3 Wrapper für einen Einsatz ohne API – Package edu.udo.jsec.wrappers

Mit den Adaptern aus 5.4.2 können Anwendungen oder Komponenten mit geringem Programmieraufwand auf eine Verwendung der JSEC-Bibliothek umgestellt werden. Die Wrapping-Erweiterung der Java Virtual Machine aus Kapitel 4 spart einerseits auch diesen Aufwand ein und ermöglicht zusätzlich die Aktivierung von JSEC auch für solche Ressourcenzugriffe, die nicht vom Anwendungsentwickler initiiert werden, sondern an anderer Stelle implementiert sind, beispielsweise innerhalb der Java-Klassenbibliothek.

Dazu sind Java-Klassen nötig, um Teile der Java-Klassenbibliothek zu ersetzen, die Ressourcenzugriffe durchführen. In der JSEC-Standardimplementierung sind solche Wrapper für alle

Klassen enthalten, zu denen in 5.4.2 Adapter vorgestellt wurden und für die Wrapping sinnvoll ist.

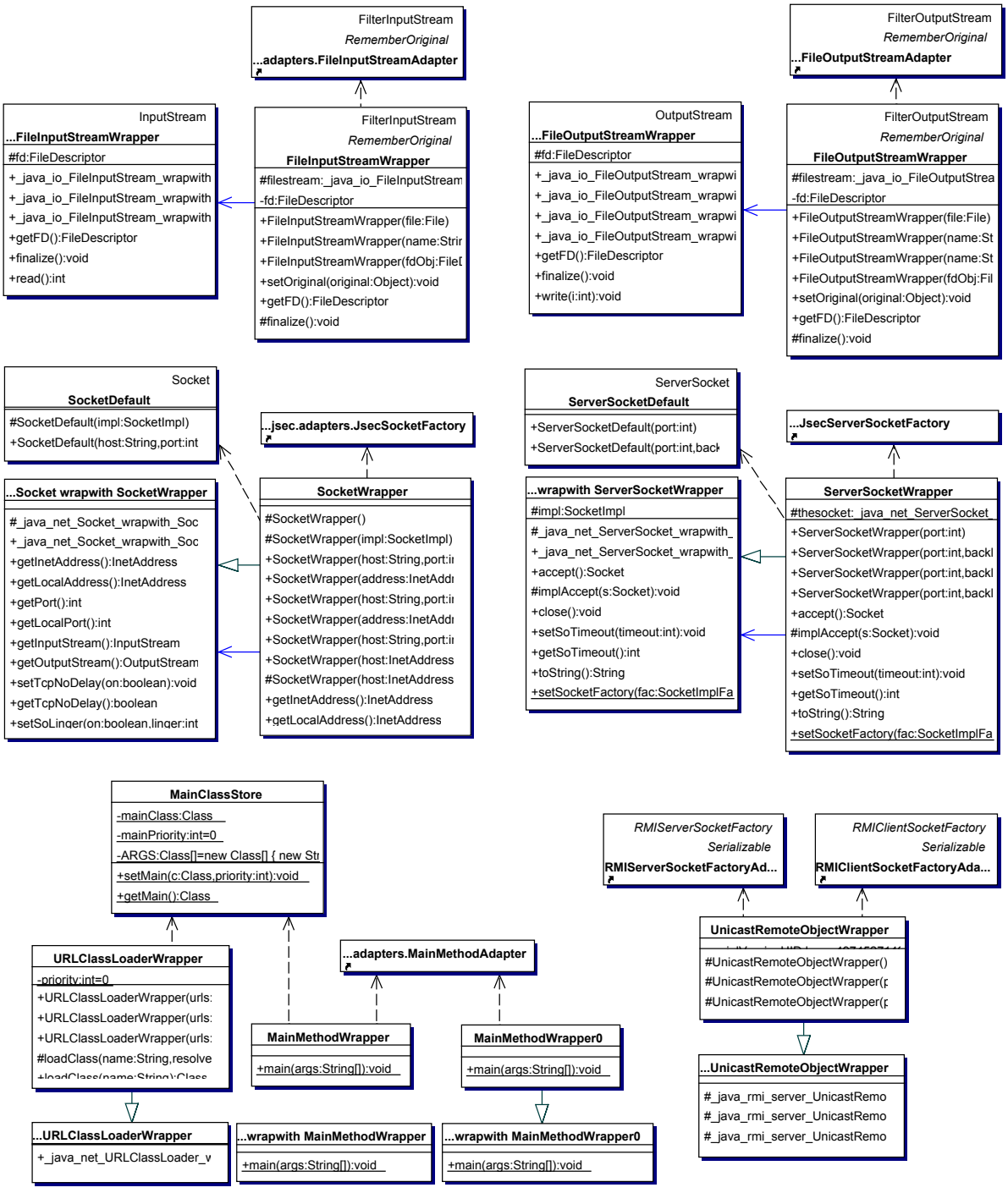

Abbildung 5-17: Klassendiagramm zum Package edu.udo.jsec.wrappers

Die Wrapper erfüllen prinzipiell dieselbe Aufgabe wie die Adapter und leiten lediglich alle Aufrufe an diese weiter; sie ersetzen allerdings jeweils eine Klasse der Java-Laufzeitumgebung und werden daher immer aktiviert, wenn eigentlich die ursprüngliche Klasse verwendet

werden sollte. Die Zusammenarbeit zwischen Wrappern und Adaptern funktioniert auch, wenn Anwendungsteile, in denen Adapter eingesetzt wurden, in einer Virtual Machine mit Wrapping ausgeführt werden.

Einige der Adapter benötigen ein Objekt der von ihnen ersetzten Klasse. Bei aktiviertem Wrapping würden sie allerdings einen Wrapper erzeugen, der wiederum den Adapter aufrufen würde, was zu einer Endlosschleife und einem Stack-Überlauf führen würde. Daher sind die Adapter auf Wrapping vorbereitet, indem sie den Wrappern eine Schnittstelle bieten, mit der ihnen der Name der ersetzten Originalklasse mitgeteilt werden kann. Dieser Name hängt wie in 4.3.1.1 beschrieben vom Wrapper ab und führt beim Adapter zu einer Erzeugung von Objekten der Originalklassen.

Da die von den Wrappern aufgerufenen Adapter durch Anwendung mehrerer Engines teilweise bereits mehrere unabhängige Modifikationen eines Zugriffs durchführen können, wird das in 4.2.3 vorgestellte und in der erweiterten Java Virtual Machine implementierte mehrfache Wrapping, also die Schaltung mehrerer geschachtelter Wrapper für eine Klasse, von den JSEC-Wrappern nicht genutzt. Mehrfaches Wrapping könnte jedoch eingesetzt werden, um unterschiedliche Adapter mehrerer Anbieter auch ohne die in 5.3.3.1 vorgeschlagenen Transformations-Engines gemeinsam einsetzen zu können und separat mit den jeweils eigenen Engines arbeiten zu lassen.

Da die Wrapper außer einem Aufruf der Adapter keine weitere Funktionalität enthalten, stellen die folgenden Abschnitte lediglich kurz die Besonderheiten der einzelnen Wrapper dar.

5.4.3.1 MainMethodWrapper

Wie in Abbildung 5-17 dargestellt ist, gibt es zwei mögliche Wrapper als Ersatz für diejenige Klasse, deren `main`-Methode nach dem Start der Java Virtual Machine aufgerufen wird. `MainMethodWrapper0` ruft lediglich `MainMethodAdapter` auf und übergibt den Namen `_java_main_wrapwith_MainMethodWrapper0` als Klasse der Anwendung, deren `main`-Methode gestartet werden soll. Dieser Name bezeichnet nach den Ersetzungsregeln aus 4.3.1.1 die ursprüngliche Startklasse der Anwendung, die durch `MainMethodWrapper0` ersetzt wurde.

Befinden sich die Wrapper allerdings in einem `wrapclasspath` (siehe 4.3.3), so werden diese nicht von einem herkömmlichen Class Loader geladen, sondern vom Bootstrap Class Loader der Java Virtual Machine (siehe 2.2.1.2.1). Das führt dazu, daß die ersetzte Startklasse der Anwendung nicht vom Wrapper geladen werden kann, wenn diese nicht im `wrapclasspath` zu finden ist. Bei der Übergabe von `_java_main_wrapwith_MainMethodWrapper0` würde daher eine `ClassNotFoundException` auftreten.

Für diesen Fall kann in der Wrapping-Konfiguration mit `wrapmain` statt `MainMethodWrapper0` die Klasse `MainMethodWrapper` festgelegt werden. Diese übernimmt die Startklasse der Anwendung aus `MainClassStore`, wo sie zuvor von `URLClassLoaderWrapper` hinterlegt wird. Dieser ersetzt den Class Loader, der von der Virtual Machine als

erstes zum Laden der Startklasse einer Anwendung eingesetzt wird und speichert eine Referenz auf diese Klasse, nachdem sie geladen wurde. Auf diese Weise kann das beschriebene Problem des Class Loaders umgangen werden.

Gemäß der in 4.2.1 eingeführten Klassifizierung von Wrappern handelt es sich bei `MainMethodWrapper0` um einen *schlanken Wrapper*, da diese Klasse von der Rumpfklasse bzw. zur Laufzeit von der ersetzten Klassen erbt. `MainMethodWrapper` besitzt aufgrund der Class-Loader-Problematik keine direkte Verbindung zu der von ihm ersetzten Klasse. Dies würde zunächst auf einen *autarken Wrapper* hindeuten; die Verbindung wird jedoch durch `URLClassLoaderWrapper` indirekt hergestellt, so daß sich ein *aktiver Wrapper* ergibt, der über eine Referenz auf Klassenebene mit der ersetzten Klasse verbunden ist.

5.4.3.2 SocketWrapper und ServerSocketWrapper

Zwischen `SocketWrapper` bzw. `ServerSocketWrapper` und den von ihnen ersetzten Klassen besteht gemäß Abbildung 5-17 sowohl eine Vererbungs- als auch eine Assoziationsbeziehung. Die Assoziation wird einerseits genutzt, um Aufrufe aller Methoden an ein von JSEC erzeugtes Socket zu delegieren, wenn eine dieser Klassen statt `Socket` bzw. `ServerSocket` instanziiert werden. JSEC erzeugt dabei ein gesichertes Socket, das eigentlich eine Subklasse von `Socket` ist, durch Wrapping aber eine Subklasse von `SocketWrapper`. Dessen Konstruktoren werden vom gesicherten Socket jedoch erneut aufgerufen, so daß eine Endlosschleife mit immer neuen JSEC-Sockets entstehen würde.

Daher wird im gesicherten Socket eine Assoziation mit `SocketDefault` hergestellt und nicht JSEC zur Erzeugung eines anderen Sockets aufgerufen. Diese Klasse erbt von `java.net.Socket`, zur Laufzeit aber wiederum von `SocketWrapper`. Um nun bei der Delegation keine Endlosschleifen zu erzeugen, wird von den einzelnen Methoden geprüft, ob es sich beim assoziierten Objekt um sich selbst handelt und in diesem Fall die Methode der Superklasse aufgerufen, so daß schließlich die Originalmethoden der Klasse `Socket` bzw. `SocketWrapper` innerhalb des davon erbenden, gesicherten Sockets zur Anwendung kommen.

`SocketWrapper` und `ServerSocketWrapper` entscheiden also je nach Situation zwischen dem Verhalten eines *aktiven* und dem eines *schlanken Wrappers*. Die Verbindung zum Objekt der Originalklasse wird im ersten Fall indirekt hergestellt, da die direkte Assoziation an ein gesichertes Socket gerichtet ist, das allerdings aus einer Subklasse der ersetzten Klasse stammt.

5.4.3.3 UnicastRemoteObjectWrapper

`UnicastRemoteObjectWrapper` ist ein sehr einfacher *schlanker Wrapper*, der beim Aufruf der Konstruktoren lediglich `RMIClientSocketFactory` und `RMIServerSocketFactory` zum Erzeugen sicherer Sockets bei RMI-Methodenaufrufen festlegt. Da von JSEC durch `SocketWrapper` und `ServerSocketWrapper` aus 5.4.3.2 bereits alle erzeugten Sockets berücksichtigt werden, also auch die von RMI erzeugten, ist dieser Wrapper nicht nötig, wenn

auf Server- und Client-Seite Wrapping zur Verfügung steht. Können die Clients jedoch nicht in einer erweiterten Java Virtual Machine betrieben werden, so führt der Einsatz dieses Wrappers auf der Server-Seite zu einer Erzeugung von Sockets mit JSEC auch auf der Client-Seite.

Wie bei `UnicastRemoteObjectAdapter` aus 5.4.2.4 kann ohne Wrapping auf beiden Seiten jedoch keine Sicherung der Verbindungen zur RMI-Registry hergestellt werden. Bei Verwendung von Wrapping auf der Server-Seite muß in der JSEC-Policy der Port dieser Registry daher bei der Erzeugung sicherer Sockets explizit ausgeschlossen werden, da sonst keine Verbindung mit den herkömmlichen Sockets der aufrufenden Clients aufgebaut werden kann.

5.4.3.4 FileInputStreamWrapper und FileOutputStreamWrapper

`FileInputStreamWrapper` und `FileOutputStreamWrapper` erben wie die entsprechenden Adapter nicht von den eigentlich ersetzten Klassen, sondern von `FilterInputStream` und `FilterOutputStream`, um die Schachtelung mehrer Streams zu ermöglichen, die Veränderungen an gelesenen bzw. geschriebenen Daten vornehmen. Im Gegensatz zu den Adaptern können sie jedoch wie `FileInputStream` und `FileOutputStream` eingesetzt werden; sie „sind“ zur Laufzeit schließlich diese Klassen.

Die Funktion ist dieselbe wie die der Adapter, dessen Methoden dazu aufgerufen werden: Zunächst wird ein „echter“ `FileInputStream` oder `FileOutputStream` erzeugt – also eine Instanz der ersetzten Klasse. Diese kann dann von Engines mit anderen Streams umhüllt werden, um die Veränderung der Daten vorzunehmen.

Bei diesen Klassen handelt es sich um *aktive Wrapper*, da ein Objekt der ersetzten Klasse erzeugt und wiederverwendet wird.

Die hier vorgestellten Wrapper können mit der in Anhang A.4.4 enthaltenen Wrapping-Konfigurationsdatei gemeinsam genutzt werden, so daß unveränderte Anwendungen die JSEC-Bibliothek bei Zugriffen auf die von den Wrappern behandelten Ressourcen einsetzen und damit eine konfigurierte Sicherheitsspezifikation durchgesetzt wird.

5.4.4 ContextAspects für Conditions – Package edu.udo.jsec.contexts

In den Bedingungen, die in einer Policy spezifiziert werden, können neben dem aktuellen Ressourcenzugriff, der über eine Permission spezifiziert wird, auch weitere Aspekte des Ausführungszustandes berücksichtigt werden. So können je nach Situation unterschiedliche Engines oder dieselben Engines mit anderen Parametern für einen Zugriff aktiviert werden.

Die herkömmliche Java-Zugriffskontrolle prüft lediglich die Code Sources der ausgeführten Klassen (siehe 2.2.2.1.2). JAAS beinhaltet eine neue Policy-Implementierung, mit der zusätzlich die Principals eines angemeldeten Subjects oder Benutzers berücksichtigt werden (siehe 2.2.3.5).

JSEC führt statt dessen den erweiterbaren Mechanismus des ExecutionContext ein, der in 5.3.1 vorgestellt wurde. In der XML-Policy aus 5.4.1 wird zum Prüfen eines ContextAspect das Tag `<context>` eingesetzt (siehe Abbildung 5-7). Die darin angegebene Subklasse von `jsec.ContextAspect` wird mit den gefundenen Optionen instanziiert und beim Auswerten der Bedingungen aufgerufen.

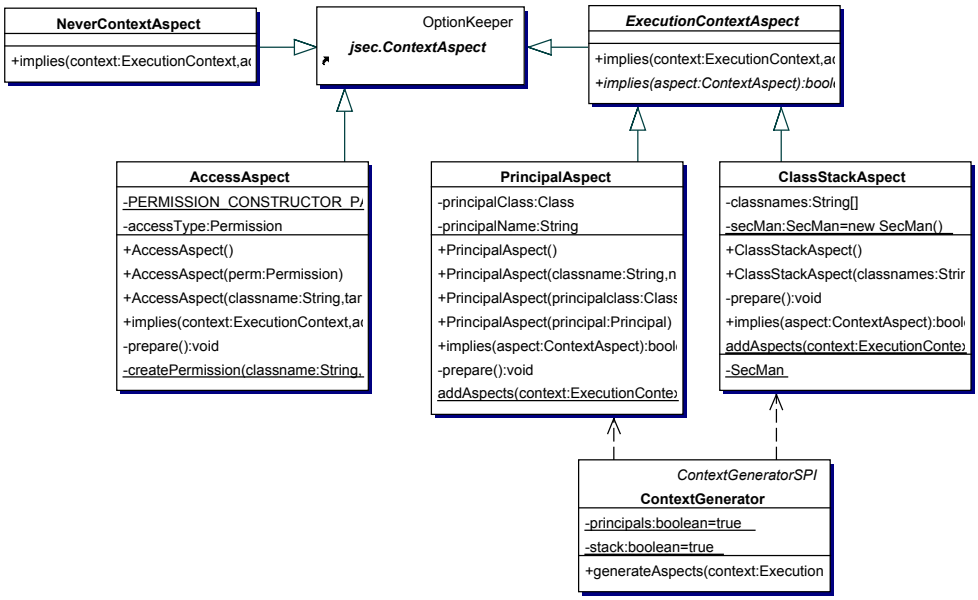

Abbildung 5-18: Klassendiagramm zum Package edu.udo.jsec.contexts

Das Package `edu.udo.jsec.contexts` enthält einige ContextAspects, die in einer JSEC-Policy abgefragt werden können:

- `NeverContextAspect` liefert bei einem Aufruf von `implies` stets `false`. Zusammen mit `AllContextAspect` aus dem Package `jsec` können so beide Wahrheitswerte verwendet werden.
- `AccessAspect` berücksichtigt nicht den an `implies` übergebenen ExecutionContext, sondern lediglich die `Permission`, die den Ressourcenzugriff spezifiziert. `AccessAspect` wertet die Optionen `class`, `target` und `action` aus und erzeugt daraus ein Permission-Objekt; dessen implies-Methode wird zum Vergleichen mit dem aktuellen Ressourcenzugriff eingesetzt. In der XML-Policy kann zur Abkürzung der Schreibweise das Tag `<access>` eingesetzt werden (siehe Abbildung 5-6); die dazugehörige Klasse `AccessCheck` aus Abbildung 5-13 delegiert jedoch wiederum an diese Klasse.
- `PrincipalAspect` dient zur Abfrage von Principals, die demjenigen Subject zugeordnet sind, das den aktuellen Thread ausführt. Einem Subject werden verschiedene Principals durch JAAS-LoginModule zugeordnet. Ein Principal wird anhand seiner Klasse und seines Namens identifiziert, die `PrincipalAspect` aus den Optionen `class` und `name` liest. In Abbildung 5-10 ist ein Beispiel für die Benutzung dieser Klasse in einer XML-Policy zu sehen.

- `ClassStackAspect` prüft, ob alle Klassen ausgeführt werden – d. h. auf dem Ausführungsstack vorhanden sind –, die über die Optionen `class.1`, `class.2` usw. angegeben wurden. Abbildung 5-8 enthält ein Beispiel zur Verwendung dieser Klasse.

Die Prüfung von Code Sources ist in der JSEC-Standardimplementierung nicht enthalten, da sich die Auswahl von Verschlüsselungsverfahren und Schlüsseln im Normalfall an den zugegriffenen Ressourcen und angemeldeten Benutzern orientieren dürfte. Bei Bedarf kann die Prüfung der Quelle und des Unterzeichners von Klassen jedoch durch Implementierung einer zusätzlichen ContextAspect-Subklasse hinzugefügt werden.

Welche ContextAspects abgefragt werden, wird in den Bedingungen einer Policy festgelegt. Zuvor muß bei der Erzeugung des ExecutionContext jedoch bekannt sein, welche ContextAspects installiert sind und evtl. bei der Abfrage benötigt werden. Daher können in `java.security` mit `jsec.context.generator.1` (und 2, 3 usw.) mehrere ContextGeneratoren festgelegt werden, die zum Erzeugen des ExecutionContext aufgerufen werden. Neue ContextAspects benötigen also zusätzlich einen ContextGenerator, der den Zustand der Ausführungsumgebung untersucht und den ExecutionContext mit entsprechenden Objekten füllt.

Die Klasse `ContextGenerator` in `edu.udo.jsec.contexts` wird aufgerufen, wenn kein solcher Eintrag gefunden wird. Sie fügt dem ExecutionContext `PrincipalAspect`- und `ClassStackAspect`-Objekte hinzu, die den Principals des aktuellen Subjects bzw. dem Stack-Inhalt entsprechen. Diese Objekte werden beim Auswerten einer Policy mit den dort instanziierten Objekten über die Methode `implies` verglichen.

5.4.5 Engines für Implications – Package edu.udo.jsec.engines

Wird eine Condition in einer Rule der JSEC-Policy zu `true` ausgewertet, so müssen die Engines instanziiert werden, die in der Implication spezifiziert wurden und zu dem Adapter passen, der einen Ressourcenzugriff durchführen möchte. Ein Adapter fordert daher stets Engines an, die nur Service Provider Interfaces (SPI) aus `jsec.engines` implementieren, die er kennt. So kann er anschließend die ihm bekannten Methoden der Engines aufrufen. Gleichzeitig ist sichergestellt, daß Adapter und Engines verschiedener Anbieter zusammenarbeiten können (siehe auch 5.3.3.1). Die Adapter der Standardimplementierung wurden in 5.4.2 vorgestellt.

Die Engines aus `edu.udo.jsec.engines` stellen also eine Grundausstattung dar, die unabhängig erweitert werden kann. In Abbildung 5-19 sind diese Engines mit den von ihnen implementierten SPIs dargestellt.

5.4.5.1 LoginEngine

`LoginEngine` implementiert das Interface `ActionEngineSPI` und stellt eine Methode `doAction(PrivilegedAction action)` zur Verfügung. Diese Methode ruft JAAS auf und

führt einen `LoginContext` zur Anmeldung eines neuen Subject aus. Welche Art von Login durchgeführt wird, kann in den herkömmlichen JAAS-Konfigurationsdateien festgelegt werden. Mit dieser Konfiguration kann unter Windows NT der angemeldete Benutzer festgestellt und dem Subject als Principal zugeordnet werden:

```
jseclogin {
    com.sun.security.auth.module.NTLoginModule required;
};
```

Mit anderen `LoginModule`n kann ebenso ein Benutzername mit Kennwort abgefragt oder SmartCards, Fingerabdrücke und andere Quellen gelesen werden.

Für LoginModule, die eine Benutzerinteraktion benötigen, kann eine `LoginEngine` über Optionen parametrisiert werden. So kann beispielsweise der eingesetzte `CallbackHandler` definiert werden, der die Benutzeroberfläche erzeugt; in 5.4.5.9.1 werden einige implementierte CallbackHandler vorgestellt. Ebenso kann eine maximale Anzahl von Wiederholungen bei falsch eingegebenen Kennwörtern und die dazwischenliegende Pause festgelegt werden.

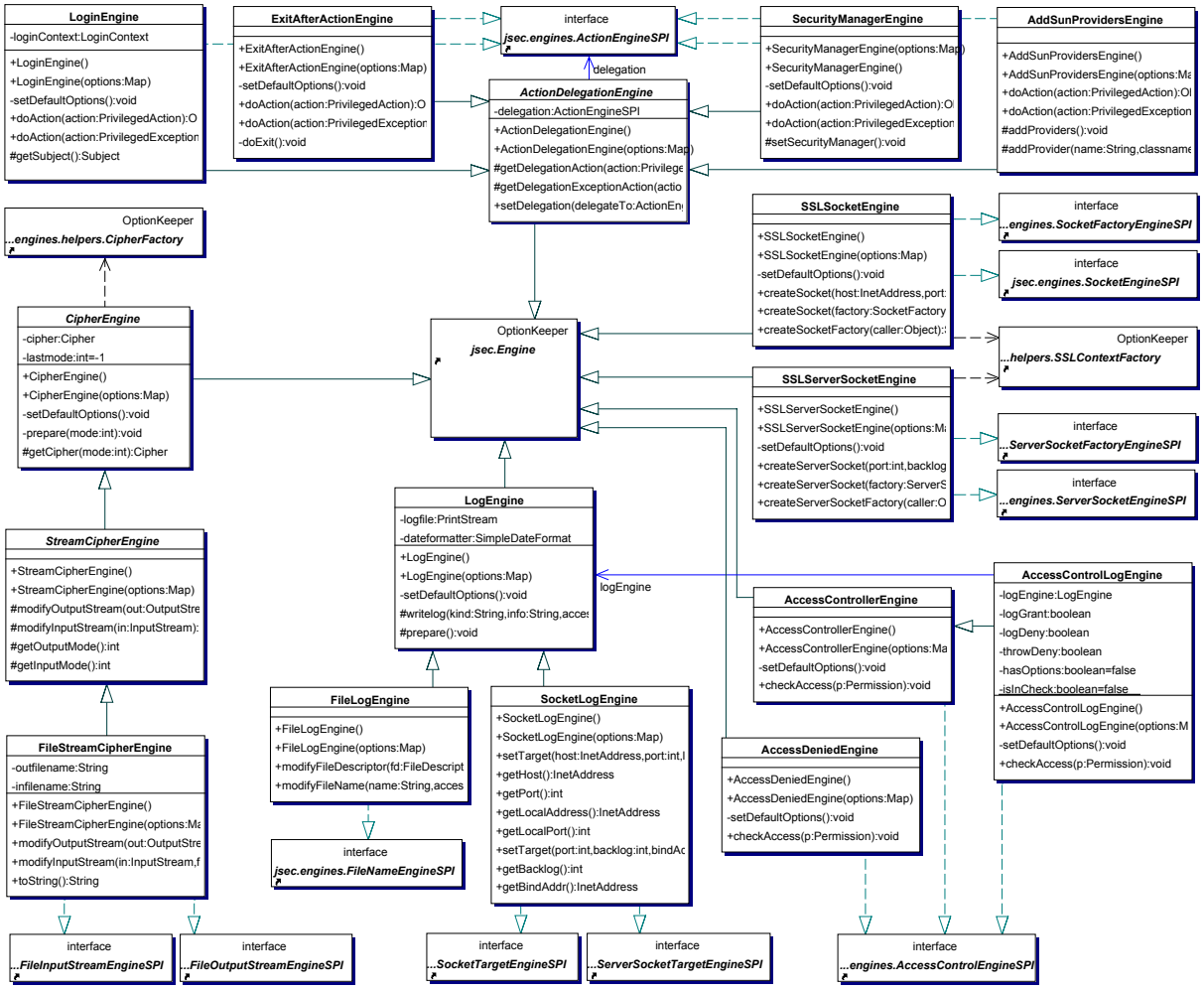

Abbildung 5-19: Klassendiagramm zum Package edu.udo.jsec.engines

5.4.5.2 ExitAfterActionEngine

`ExitAfterActionEngine` ist zur Benutzung beim Programmstart vorgesehen. Nach Ausführung der Action, also der `main`-Methode des Programms, wird `System.exit()` aufgerufen und damit die Java Virtual Machine beendet. Dies kann hilfreich sein, wenn durch die eingesetzten Engines neue Threads gestartet werden, die eine automatische Beendigung nach Ablauf von `main` verhindern würden, beispielsweise wenn die `LoginEngine` eine graphische Benutzeroberfläche in einem zuvor textbasierten Programm startet.

5.4.5.3 AddSunProvidersEngine

`AddSunProvidersEngine` implementiert ebenfalls `ActionEngineSPI` und kann beim Start einer Anwendung mit einem MainMethodAdapter aus 5.4.2.1 eingesetzt werden, um die Sun-Provider für JCA, JCE, JSSE und JAAS in der JCA-Providerverwaltung zu registrieren und sie für andere Engines verfügbar zu machen. Dies ist für Situationen vorgesehen, in denen die Konfigurationsdatei `java.security` nicht verändert werden kann, in der diese Provider üblicherweise eingetragen werden können.

5.4.5.4 SecurityManager-, AccessController- und AccessControlLogEngine

`SecurityManagerEngine` ist ebenfalls für den Start einer Anwendung vorgesehen und setzt vor Ausführung der `action` einen SecurityManager, damit die Java-Zugriffskontrollmechanismen ausgeführt werden. Java-Programme werden nämlich ohne SecurityManager ausgeführt, wenn dieser nicht vom Programm oder in der Kommandozeile gesetzt wird – oder durch Konfiguration dieser Engine. In der JSEC-Policy kann der Klassenname des gewünschten SecurityManagers als Option angegeben werden, sonst wird `java.lang.SecurityManager` benutzt. Hier kann beispielsweise auch der in 5.4.2.2 vorgestellte `SecurityManagerAdapter` festgelegt werden. Zusätzlich kann ein Dateiname für ein Java Policy File angegeben werden, der ansonsten in der Kommandozeile oder in `java.security` angegeben werden muß.

`AccessControllerEngine` implementiert das Interface `AccessControlEngineSPI` und nutzt für seine Zugriffskontrollentscheidung den herkömmlichen `AccessController` aus 2.2.2.2.2 und damit die normale Java Access Control Policy. Wird für den Programmstart eine `SecurityManagerEngine` definiert, die den `SecurityManagerAdapter` als SecurityManager einsetzt, und wird die `AccessControllerEngine` für die Bedingung `<true/>` konfiguriert, so wird das Verhalten des herkömmlichen SecurityManager erreicht.

Wird für `<true/>` statt dessen die `AccessControlLogEngine` festgelegt, so bietet diese zusätzlich die Möglichkeit, erlaubte und verbotene Zugriffe zu protokollieren und wahlweise das Werfen einer `SecurityException` bei verbotenen Zugriffen zu unterbinden. Damit kann eine Anwendung, für die noch keine Java Policy Files existieren, ohne Abbrüche durch die Zugriffskontrolle in einer Testumgebung ausgeführt werden und alle nötigen Permissions protokolliert werden. Anhand dieses Protokolls kann der Systemadministrator dann entschei-

den, welche Rechte die Anwendung erhalten soll und damit die Java Policy Files für die Einsatzumgebung erstellen.

5.4.5.5 AccessDeniedEngine

`AccessDeniedEngine` implementiert ebenfalls das Interface `AccessControlEngineSPI`, das von allen Adaptern aus 5.4.2 vor dem Ressourcenzugriff aufgerufen wird. Die Methode `checkAccess(Permission p)` wirft bedingungslos eine SecurityException, die zu einem Abbruch des Ressourcenzugriffs und des Programms führt. Durch Spezifikation einer `AccessDeniedEngine` in der Implication einer Rule werden also die in der Condition abgefragten Zugriffe verhindert. Im Gegensatz zur üblichen Erlaubnisvergabe können mit einer `AccessDeniedEngine` also Zugriffsverbote konfiguriert werden.

5.4.5.6 SSLSocketEngine und SSLServerSocketEngine

`SSLSocketEngine` implementiert die Interfaces `SocketFactoryEngineSPI` und `SocketEngineSPI`, so daß der SocketAdapter bzw. die JsecSocketFactory aus 5.4.2.3 damit Sockets erzeugen können. Die Sockets stammen aus JSSE, so daß SSL- und TLS-Verbindungen ermöglicht werden (siehe auch 2.2.3.3). Zur Erzeugung der Sockets wird die Klasse `SSLContextFactory` genutzt, die in 5.4.5.9.6 noch beschrieben wird.

In der JSEC-Policy kann per Option definiert werden, ob der SSL-Handshake sofort nach Erzeugen des Sockets stattfinden soll; standardmäßig wird dieser sofort durchgeführt, da ohne spätere Ausführung des Handshake die Identitäten der Verbindungsendpunkte nicht geprüft werden.

`SSLServerSocketEngine` entspricht der zuvor genannten Engine, allerdings für Sockets auf der Server-Seite. Zum Erzeugen der ServerSockets wird ebenfalls eine `SSLContextFactory` aus 5.4.5.9.6 eingesetzt. Diese Engine berücksichtigt eine Option, mit der die Authentifizierung der Clients abgeschaltet werden kann, so daß jeder beliebige Client eine Verbindung mit dem erzeugten Socket herstellen kann.

Die verwendeten JSSE-Sockets können durch ihre Provider-Architektur mit verschiedenen Implementierungen der kryptographischen Algorithmen zusammenarbeiten. Die Parameter zur Auswahl der Algorithmen können als Optionen in der JSEC-Policy festgelegt werden. Durch einen Einsatz effizienter Implementierungen, die beispielsweise Native Code oder Hardware zur Ausführung der kryptographischen Algorithmen verwenden, kann auch die Ausführungsgeschwindigkeit gesteigert werden.

5.4.5.7 SocketLogEngine und FileLogEngine

`SocketLogEngine` stellt die Methoden aus `SocketTargetEngineSPI` und `ServerSocketTargetEngineSPI` zur Verfügung. Diese werden allerdings nicht eingesetzt, um ein

anderes Ziel für die Socket-Verbindung festzulegen, sondern um die Zieladresse in einer Logdatei zu erfassen.

`FileLogEngine` erfaßt Dateizugriffe in einer Logdatei und nutzt dazu das Interface `FileNameEngineSPI`, ohne den Dateinamen zu verändern. Der Name der Logdatei kann per Option in der JSEC-Policy festgelegt werden.

5.4.5.8 FileStreamCipherEngine

`FileStreamCipherEngine` bietet Ver- und Entschlüsselung für Stream-Dateizugriffe. Dazu implementiert es die Interfaces `FileInputStreamEngineSPI` und `FileOutputStreamEngineSPI`, so daß ein `CipherInputStream` aus JCE genutzt werden kann (siehe auch 2.2.3.2). Der dazu nötige und zur Ver- und Entschlüsselung eingesetzte `Cipher` wird durch eine `CipherFactory` erzeugt, die in 5.4.5.9.2 noch beschrieben wird.

Die abstrakten Superklassen `StreamCipherEngine` und `CipherEngine` aus Abbildung 5-19 werden in der JSEC-Standardimplementierung zwar nicht an anderer Stelle eingesetzt; sie wurden aber dennoch als separate Klassen entworfen, da sie für weitere Engines eingesetzt werden können, die einen `Cipher` benötigen.

Die vom Cipher eingesetzten Algorithmen sind durch die Provider-Architektur von JCA und JCE austauschbar. Die nötigen Parameter zur Auswahl der Algorithmen können als Optionen in der JSEC-Policy festgelegt werden. Durch einen Einsatz von Implementierungen, die beispielsweise Native Code oder Hardware zur Ausführung der kryptographischen Algorithmen verwenden, kann zudem die Ausführungsgeschwindigkeit beeinflußt werden.

5.4.5.9 Engine-Unterstützung – Package edu.udo.jsec.engines.helpers

In den vorigen Abschnitten wurde auf die Klassen `CallbackHandler`, `CipherFactory` und `SSLContextFactory` hingewiesen, die wesentliche Funktionalität für einige der Engines bereitstellen. Diese Klassen befinden sich im Package `edu.udo.jsec.engines.helpers`, das in Abbildung 5-20 dargestellt ist.

5.4.5.9.1 CallbackHandler

Das Interface `CallbackHandler` stammt aus JAAS und bietet eine Schnittstelle für Interaktion mit dem Benutzer. JAAS enthält jedoch keine Implementierungen dieses Interface. JAAS definiert verschiedene `Callback`-Klassen, von denen mehrere Objekte an einen CallbackHandler übergeben werden können und zur Darstellung oder Abfrage von Informationen führen. Ein `TextOutputCallback` enthält beispielsweise einen Text, der dargestellt werden soll, während ein `PasswordCallback` zur Abfrage eine Kennworts und ein `NameCallback` zur Abfrage eines Benutzernamens führen soll.

Für JSEC wurden drei CallbackHandler implementiert:

- `TextCallbackHandler` für einen Einsatz in textbasierten Anwendungen.

- `AWTCallbackHandler` nutzt das Abstract Windowing Toolkit (AWT).
- `SwingCallbackHandler` basiert auf der Oberflächenbibliothek Swing. In Abbildung 5-21 und Abbildung 5-22 ist ein Beispiel für den Einsatz dieser Klasse dargestellt.

Die entwickelten CallbackHandler können auch unabhängig von JSEC für JAAS-LoginModule eingesetzt werden. In der JSEC-Standardimplementierung werden sie von `LoginEngine` für JAAS-Logins und von `KeyStoreManager` zur Abfrage von Kennwörtern beim Laden von Schlüsseln eingesetzt (siehe 5.4.5.9.3). In beiden Fällen kann der benutzte CallbackHandler über eine Option definiert werden, so daß nicht nur diese drei, sondern auch andere Klassen eingesetzt werden können, die das Interface `CallbackHandler` aus JAAS implementieren.

Bösartige Programme könnten ein gleich aussehendes Fenster öffnen, um den Benutzer zur Eingabe von Kennwörtern zu bringen und diese anschließend zu mißbrauchen. Dies kann verhindert werden, indem ein SecurityManager verwendet wird und eine `java.awt.AWTPermission "showWindowWithoutWarningBanner"` nur an JSEC und an weitere vertrauenswürdige Klassen vergeben wird. AWT- und Swing-Fenster, die von anderen Programmteilen erzeugt werden, enthalten dann den Hinweis „Java Applet Window“, um den Benutzer auf die möglichen Gefahren hinzuweisen.

5.4.5.9.2 CipherFactory

Die Klasse `CipherFactory` ermöglicht die Erzeugung eines `Cipher`s, der für Ver- und Entschlüsselung benötigt wird. JCA und JCE bieten zwar Unabhängigkeit von konkreten Ver- und Entschlüsselungsalgorithmen und deren Implementierungen, aber bei der Initialisierung müssen Parameter und Schlüssel festgelegt werden, die davon nicht unabhängig sind. Daher steht in JSEC die abstrakte Klasse `CipherFactory` zur Verfügung, deren konkrete Implementierungen die Initialisierung eines `Cipher`s durchführen.

In der JSEC-Standardimplementierung sind zwei CipherFactories enthalten:

- `PBECipherFactory` initialisiert einen Cipher für Password Based Encryption (PBE). Dabei wird kein Schlüssel eingesetzt, sondern alle Operationen aus einem eingegebenen Kennwort berechnet. Dieses Kennwort kann in der JSEC-Policy als Option hinterlegt sein; wird es nicht gefunden, so nutzt die Factory einen per Option festgelegten `CallbackHandler`, um den Benutzer danach zu fragen.
- `CommonCipherFactory` erzeugt einen Cipher für Algorithmen mit öffentlichen und privaten oder mit geheimen Schlüsseln; der Schlüssel wird dabei von einem `KeyManager` bereitgestellt, der per Option wählbar ist (siehe 5.4.5.9.3). Der Algorithmus und seine Implementierung können ebenfalls mit Optionen eingestellt werden, die an JCA und JCE weitergereicht werden. Die Parameter der Algorithmen werden von `CommonCipherFactory` jedoch nicht beachtet, so daß diese mit Standardwerten ausgeführt werden.

Spezifische Parameter für einzelne Algorithmen können in anderen CipherFactories gesetzt werden, die per Angabe eine Option von den Engines benutzt werden können.

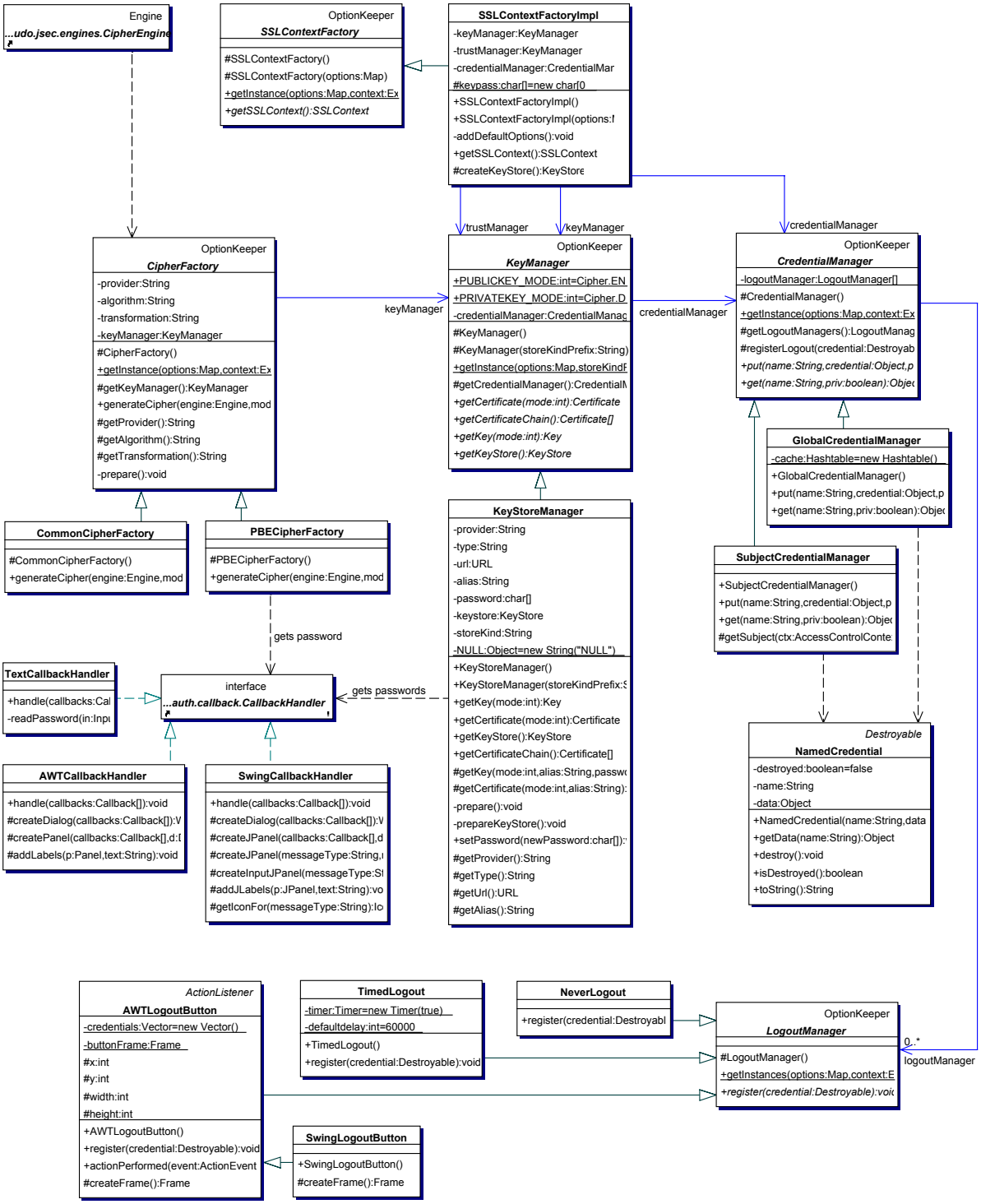

Abbildung 5-20: Klassendiagramm zum Package edu.udo.jsec.engines.helpers

5.4.5.9.3 KeyManager

Die abstrakte Klasse `KeyManager` wird unter anderem von `CipherFactory` eingesetzt, um kryptographische Schlüssel zu beschaffen. Die Art und Weise der Beschaffung ist den konkreten Implementierungen überlassen. Beispielsweise könnten Schlüssel auf Anforderung neu

erzeugt oder über einen Verzeichnisdienst beschafft werden. In JSEC ist jedoch nur die Implementierung `KeyStoreManager` enthalten, die ihre Schlüssel aus einem `KeyStore` bezieht (siehe auch 2.2.2.1.3). Da der KeyStore über JCA bereits austauschbar ist, steht hier bereits eine relativ große Flexibilität zur Verfügung.

Die Zusatzleistung des `KeyStoreManagers` besteht darin, den Inhalt des KeyStore aus einer Datei oder über Internet-Protokolle zu laden und den richtigen Schlüssel auszuwählen. Die URL des KeyStore sowie der Alias des gewünschten Schlüssels wird aus Optionen gelesen. Es sind andere Implementierungen denkbar, bei denen die Wahl des Schlüssels beispielsweise vom aktuellen Benutzer abhängt. In dieser Implementierung sind solche Anforderungen mit separaten KeyStores in den privaten Verzeichnissen der verschiedenen Benutzer erfüllbar, so daß aus `file:${user.home}/.keystore` unterschiedliche Schlüssel geladen werden.

Beim Laden eines KeyStore und beim Lesen eines privaten oder geheimen Schlüssels sind Kennwörter notwendig. Diese können entweder als Optionen in der JSEC-Policy eingegeben werden oder werden zur Laufzeit vom Benutzer erfragt. Dabei wird ein `CallbackHandler` eingesetzt, der per Option wählbar ist. In Verbindung mit dem oben vorgestellten `SwingCallbackHandler` mit Metal Look & Feel stellt sich eine Kennwortabfrage für einen KeyStore wie in Abbildung 5-21 dar.

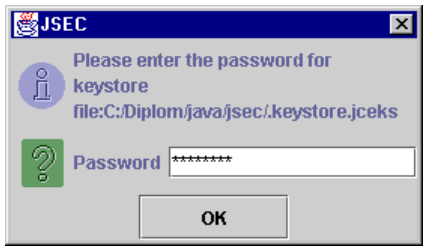

Abbildung 5-21: Kennwortabfrage für einen KeyStore mit dem SwingCallbackHandler

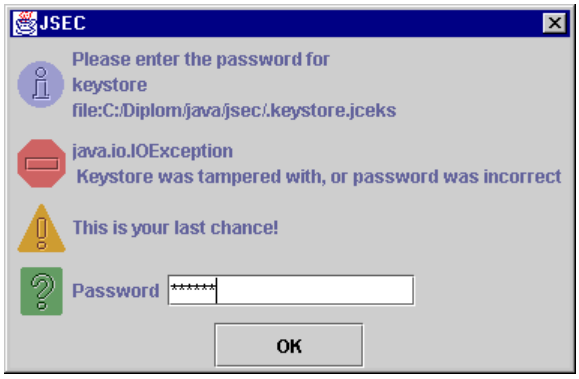

Abbildung 5-22: Kennwortabfrage mit Fehlermeldung und Warnung

Bei Eingabe eines falschen Kennworts wird je nach Optionen mit kurzen Zwangspausen noch mehrmals nachgefragt. Eine Beschreibung des aufgetretenen Fehlers wird als `Error` und eine Warnmeldung beim letzten Versuch als `Warning` an den CallbackHandler übergeben. Diese Bestandteile werden vom SwingCallbackHandler wie in Abbildung 5-22 dargestellt.

5.4.5.9.4 CredentialManager

Um KeyStores und Schlüssel nicht bei jeder Anforderung neu laden zu müssen, werden diese im Arbeitsspeicher zwischengespeichert. Dies delegiert der `KeyManager` an einen `CredentialManager`, der selbst entscheidet, ob und an welcher Stelle die Informationen zwischengespeichert werden. Vor dem Laden eines KeyStore oder eines Schlüssels wird zunächst die `get`-Methode des CredentialManagers aufgerufen, um eventuell bereits geladene Informationen wiederzuverwenden; nach dem Laden wird `put` zum Zwischenspeichern der Informationen aufgerufen.

Auch der eingesetzte `CredentialManager` kann über Optionen festgelegt werden. Wie in Abbildung 5-20 dargestellt, wurden zunächst zwei Implementierungen realisiert.

- `GlobalCredentialManager` speichert die Informationen in einem statischen `Hashtable`, so daß die Informationen in der gesamten Java Virtual Machine verfügbar sind.
- `SubjectCredentialManager` kann eingesetzt werden, wenn mit `doAs` ein JAAS-Subject festgelegt wurde, in dessen Namen das Programm oder ein Programmteil ausgeführt wird, wenn also beispielsweise eine `LoginEngine` beim Programmstart eingesetzt wurde. `Subject` bietet die Möglichkeit zur Speicherung von Credentials, also beliebigen sicherheitsspezifischen Objekten, die einem Subject zugeordnet werden sollen. `SubjectCredentialManager` liest und speichert alle Credentials im jeweils aktuellen Subject, so daß bei Mehrbenutzerumgebungen oder Zuordnung von Server-Prozessen zu Subjects eine klare Trennung der Daten der unterschiedlichen Subjects geleistet werden kann und keine Schlüssel oder sonstige vertrauliche Informationen im Namen anderer Subjects eingesetzt werden.

5.4.5.9.5 LogoutManager

Die Zwischenspeicherung vertraulicher Informationen wie privater oder geheimer Schlüssel kann ein Sicherheitsrisiko darstellen, wenn beispielsweise die Informationen eines Benutzers im Arbeitsspeicher einer laufenden Virtual Machine erhalten bleiben, auch nachdem dieser seinen Arbeitsplatz verlassen hat.

Daher registrieren die `CredentialManager` alle zwischengespeicherten Informationen bei einem oder mehreren `LogoutManager`n, die über Optionen in der JSEC-Policy festgelegt werden können. Diese zerstören nach ihrem individuellen Aktivierungsereignis alle registrierten Credentials; diese sind danach zwar weiterhin in einem Subject oder an anderer Stelle vorhanden, aber ohne Inhalt und damit wertlos.

Diese LogoutManager stehen in der JSEC-Standardimplementierung zur Verfügung:

- `NeverLogout` ruft die `destroy`-Methode der registrierten Credentials nie auf und ist für Schlüssel in Server-Prozessen sinnvoll.
- `TimedLogout` ruft nach Ablauf einer konfigurierbaren Zeitspanne nach der Registrierung eines Credentials `destroy` auf.

- `AWTLogoutButton` öffnet ein Fenster mit einem Button zur manuellen Ausführung des Logout und gibt dem Benutzer so die Möglichkeit, beim Verlassen des Arbeitsplatzes seine sicherheitsrelevanten Informationen sofort zu löschen.
- `SwingLogoutButton` entspricht diesem, basiert aber auf der Bibliothek Swing (siehe Abbildung 5-23).

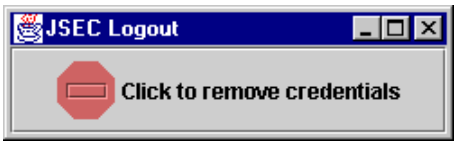

Abbildung 5-23: Der LogoutManager SwingLogoutButton

Zudem können weitere LogoutManager implementiert werden, die auf andere Weise festlegen, wann die sicherheitsrelevanten Informationen gelöscht werden sollen.

Der parallele Einsatz mehrerer LogoutManager bietet beispielsweise die Möglichkeit, die Löschung nach einer gewissen Zeit oder bereits vorher auf Benutzeranforderung zu starten.

5.4.5.9.6 SSLContextFactory

In Abbildung 5-20 ist auch die Klasse `SSLContextFactory` dargestellt, die auf die Erzeugung von `SSLContexts` für JSSE-Sockets spezialisiert ist (siehe auch 2.2.3.3). In JSEC steht eine konkrete Implementierung zur Verfügung, die durch andere Klassen erweitert oder ausgetauscht werden kann.

Ein `SSLContext` dient in JSSE zur Erzeugung von SSLSocketFactories und SSLSockets und muß mit X.509-Schlüsseln oder –Zertifikaten initialisiert werden, die die Identität des Servers oder Clients bestätigen und zur asymmetrischen Verschlüsselung geheimer Schlüssel eingesetzt werden, die wiederum für symmetrische Algorithmen zur Verschlüsselung der übertragenen Nutzdaten verwendet werden (hybrides Verschlüsselungsverfahren, siehe auch 2.1.3.4).

Die Schlüssel und Zertifikate werden von einem `KeyManager` aus 5.4.5.9.3 bezogen und anschließend einem `CredentialManager` aus 5.4.5.9.4 zur Zwischenspeicherung übergeben. Zusätzlich wird auch der erzeugte `SSLContext` als Credential zwischengespeichert, da dessen Wiederverwendung für Verbindungen mit denselben Identitätsinformationen möglich ist und Geschwindigkeitsvorteile bietet.

In diesem Abschnitt wurden einige der Konfigurationsoptionen und die damit verbundenen Fähigkeiten der JSEC-Bestandteile erwähnt; eine vollständige Auflistung an dieser Stelle würde die Übersichtlichkeit verringern und ist daher in Anhang A.4 tabellarisch notiert.

5.5 Anmerkungen zur Implementierung

Die in 5.3 und 5.4 vorgestellten Bestandteile von JSEC wurden im Rahmen der Arbeit implementiert. Dabei wurden sicherheitsrelevante Methoden, die beispielsweise Informationen der JSEC-Policy oder andere kritische Daten wie private Schlüssel und andere Credentials zugänglich machen, durch Nutzung des Security Managers und damit durch die bekannten Java-Zugriffskontrollmechanismen geschützt. Dazu wurde die Klasse `jsec.JsecPermission` als neue Permission-Klasse eingeführt und an den in Anhang A.3 aufgelisteten Stellen in JSEC geprüft.

Da JSEC sowohl bei Nutzung von Adaptern als auch beim Einsatz der Wrapper von Anwendungsklassen aufgerufen wird, befinden sich diese im `AccessControlContext`, wenn JSEC Zugriffe auf geschützte Ressourcen durchführt. Gemäß der in 2.2.2.2.2 vorgestellten Stack Inspection müßten dazu alle von JSEC benötigten Rechte auch an die Anwendungen vergeben werden, beispielsweise die erwähnten `JsecPermissions` sowie Rechte zum Lesen der Policy-Dateien oder zum Verwalten der privaten Credentials im Subject. Anwendungen sollten diese Rechte jedoch keinesfalls besitzen, um einen Mißbrauch auszuschließen.

Innerhalb von JSEC wurde daher Privileged Code eingesetzt (siehe 2.2.2.2.3), so daß die geschützten Zugriffe allein mit den an JSEC vergebenen Rechten durchgeführt werden können. Privileged Code wurde nur eingesetzt, wenn die darin gewonnenen Informationen nicht an die Anwendung gelangen können. Zudem werden die von der Anwendung angeforderten Ressourcenzugriffe selbst nicht unter den Rechten von JSEC durchgeführt, so daß eine Anwendung durch Aufruf von JSEC keine Rechte anwenden kann, die sie nicht schon besitzt.

Zur Fehlerbehandlung wurden in JSEC neue Exception-Klassen wie `jsec.JsecException` oder `edu.udo.jsec.policy.JsecXmlException` eingeführt und im Fehlerfall geworfen, so daß aufrufende Methoden nicht alle eventuell auftretenden Exceptions kennen müssen, was wegen der Erweiterbarkeit von JSEC auch nicht möglich wäre. Die tatsächlich aufgetretene Exception wird aber in der `JsecException` mitgeführt, um bei einem letztendlichen Programmabbruch anhand deren Stack-Aufzeichnung die tatsächliche Fehlerquelle feststellen zu können und nicht nur die Stelle, an der die `JsecException` geworfen wurde.

Während des Starts der Java Virtual Machine wird JSEC durch Wrapper für Systemklassen bereits aktiviert, bevor die Java-Klassenbibliothek ausreichend weit initialisiert ist; in diesem Zustand kann die Funktionsfähigkeit nicht gewährleistet werden. Daher wird JSEC erst aktiv, nachdem `jsec.Policy.setReady()` aufgerufen wurde. Diese Methode kann zu Beginn von Anwendungen oder bei Initialisierung von Komponenten aufgerufen werden, die JSEC benutzen; mit Wrapping kann darauf verzichtet werden, da `MainMethodWrapper` diese Methode vor dem Start der eigentlichen Anwendung ebenfalls aktiviert.

Bei Nutzung des `FileInputStreamWrappers` kann es durch die Aktivierung von JSEC zu einer Situation kommen, in der weitere Klassen für JSEC geladen werden müssen, was wiederum zu einem Aufruf von `FileInputStreamWrapper` führt. Dies würde zu einer erneuten Aktivierung von JSEC führen, was schließlich mit einem Stack-Überlauf beendet würde.

Daher prüft `FileInputStreamWrapper`, ob der aktuelle Thread diesen bereits benutzt; in diesem Fall ist JSEC bereits aktiviert worden, und der aktuelle Zugriff wird mit dem herkömmlichen `FileInputStream` durchgeführt. Dies bewirkt jedoch auch, daß die von JSEC geladenen Dateien – beispielsweise die Policy – nicht durch JSEC-Mechanismen geschützt werden können. Diese Einschränkung ist jedoch ohnehin gegeben, da beim Laden der JSEC-Policy noch nicht festgelegt sein kann, mit welchen Schutzmechanismen diese geladen werden soll.

Wrapper müssen zum Ersetzen von Systemklassen im `wrapclasspath` gefunden werden, da der Class Loader für den `classpath` zum Ladezeitpunkt noch nicht aktiv ist. Nach der in 2.2.1.2.1 vorgestellten Delegationsbeziehung wird derjenige Class Loader mit dem Laden einer referenzierten Klasse beauftragt, der bereits die Klasse mit der Referenz geladen hat. Da der Java-interne Primordial Class Loader die Wrapper aus dem `wrapclasspath` geladen hat, würde dieser auch JSEC und ebenso Project X, JCE, JSSE und JAAS laden müssen. Um nicht alle eingesetzten Bibliotheken im `wrapclasspath` angeben zu müssen, aktiviert JSEC manuell einen anderen Class Loader; dies wird beim Laden der gewählten Policy-Implementierung durchgeführt, so daß diese und alle daraus folgenden Klassen mit dem herkömmlichen Class Loader geladen werden, also auch die von ihr erzeugten Engines und damit die Bibliotheken JCE, JSSE und JAAS.

Daraus ergibt sich bei einem Einsatz von Wrapping folgende Verteilung von JSEC: Adapter (aus dem Package `edu.udo.jsec.adapters`) und Wrapper (aus `edu.udo.jsec.wrappers`) sowie die JSEC-Basis (aus `jsec`) müssen im `wrapclasspath` gefunden werden, während die Policy-Implementierung (aus `edu.udo.jsec.policy`) und alle weiteren JSEC-Bestandteile nicht im `wrapclasspath` vorhanden sein müssen.

5.6 Vorschläge zur Erweiterung von JSEC

In Abschnitt 5.2 wurde bereits auf die Erweiterungsfähigkeit von JSEC hingewiesen, deren Grundlage das Package `jsec` aus 5.3 darstellt. In Abschnitt 5.4 wurden konkrete Implementierungen für einzelne JSEC-Bestandteile vorgestellt, die gemeinsam als JSEC-Standardimplementierung bezeichnet werden. Innerhalb einer einzelnen, abgeschlossenen Arbeit kann die gegebene Erweiterungsfähigkeit nicht demonstriert werden, da die Grenze zwischen Standardbausteinen und Erweiterungen willkürlich sein würde.

Daher werden in diesem Abschnitt einige der Erweiterungsmöglichkeiten anhand konkreter Vorschläge für die Integration weiterführender Konzepte verdeutlicht.

- Wrapping wird als Mechanismus benutzt, um die JSEC-Bibliothek ohne Programmänderungen auch in bestehende Anwendungen integrieren zu können, und steht in der erweiterten Java Virtual Machine aus Kapitel 4 zur Verfügung. Alternativ kann jedoch auch eine herkömmliche Virtual Machine mit einer modifizierten Java-Klassenbibliothek eingesetzt werden. Dazu müssen lediglich die Wrapper statt der von ihnen ersetzten Klassen in die Klassenbibliothek integriert werden und die ursprünglichen Klassen umbenannt wer-

den; die neuen Namen dieser Klassen müssen wie bisher an die Adapter übergeben werden und ermöglichen diesen den Zugriff auf die Originalklassen.

- Die zur Verfügung stehenden Adapter und Wrapper legen fest, welche Ressourcenzugriffe durch JSEC modifiziert werden können. Um zusätzlich zu den vorgestellten Zugriffsarten weitere zu berücksichtigen, beispielsweise Datenbankzugriffe, können weitere Adapter, die in Anwendungen einsetzbar sind, und weitere Wrapper, die in bestehende Anwendungen ohne Änderungen integriert werden können, entwickelt werden. Für nicht verwandte Ressourcentypen ist dazu ebenfalls die Definition zusätzlicher Service Provider Interfaces für Engines und die Implementierung passender Engines notwendig.

- Die eingesetzten kryptographischen Algorithmen sind bereits in JCA und JCE austauschbar und können durch Parameter in der JSEC-Policy gewählt werden. Für andere Algorithmen kann es jedoch notwendig sein, andere Parameter bei der Initialisierung zu verwenden. Diese Erweiterungsmöglichkeit ist bereits in der JSEC-Standardimplementierung enthalten, indem zusätzliche CipherFactories implementiert und per Policy konfiguriert werden können (siehe auch 5.4.5.9.2).

- Um Verzeichnisdienste wie *Lightweight Directory Access Protocol (LDAP)* einzusetzen, kann einerseits ein LoginModule für JAAS implementiert werden, um auch unabhängig von JSEC eine Authentifizierung des Benutzers anhand solcher Informationen durchzuführen. Um auch die kryptographischen Schlüssel des Benutzers aus einem solchen Verzeichnis zu beziehen, kann ein JSEC-KeyManager implementiert werden, der die Schlüssel vom Verzeichnisdienst anfordert und nicht aus einem KeyStore lädt (siehe auch 5.4.5.9.3).

- Sollen mit JSEC nicht unbekannte Anwendungen mit Sicherheitsregeln ausgestattet werden, sondern die Möglichkeiten bereits bei der Anwendungsentwicklung durch Einsatz von Adaptern berücksichtigt werden, so wäre eine größere Kooperation der Anwendung mit JSEC möglich. Beispielsweise könnte die Anwendung bei einem Ressourcenzugriff zusätzlich zu einer Zieladresse oder einem Dateinamen auch einen Bezeichner für den Zugriff verwenden. So könnten die Sicherheitsregeln nicht allein anhand der angesprochenen Ressourcen, sondern auch anhand dieser Zusatzinformation ausgeführt werden.

  Die Übertragung einer Bestellung kann mit der Information „`order`“ gekennzeichnet werden, so daß mit einem `ContextAspect`, der diese Information berücksichtigt, auch eine Regel definiert werden kann, um Bestellungen zu verschlüsseln, egal wohin sie übertragen oder wo sie gespeichert wird. Die Definition der JSEC-Policy wäre damit anwendungsnäher möglich, da sie nicht auf Dateinamen und Netzwerkadressen beruhen müßte.

  Dazu sind einerseits Adapter nötig, die diese zusätzliche Spezifikation des Zugriffs entgegennehmen; und andererseits ein `ContextAspect`, der solche Informationen auswertet und in der Policy eingesetzt werden kann.

- In 1.2.3 wurde ein Vorschlag aus [PSW98] vorgestellt, bei dem Algorithmen und deren Parameter erst beim Aufbau einer Netzwerkverbindung zwischen zwei Rechnern anhand von Sicherheitszielen ausgehandelt werden, die auf einer abstrakten Ebene auch von Nicht-Spezialisten mit einer graphischen Oberfläche festgelegt werden können.

Um diese Funktionalität auch ohne Änderung von Anwendungen und Komponenten und in Verbindung mit den JSEC-Mechanismen verfügbar zu machen, könnte eine solche Entwicklung auch in Form von Sockets implementiert werden, die von einer dazugehörigen Engine erzeugt werden.

- In 5.4.5.5 wurde die `AccessDeniedEngine` vorgestellt, mit der eine negative Zugriffskontrolle mit Zugriffsverboten innerhalb der JSEC-Policy realisiert werden kann. Eine Integration der positiven Zugriffskontrolle auf Basis von Erlaubnissen in die JSEC-Policy würde als Vorteil nicht nur eine einzelne Policy für JSEC und den `AccessController` bieten, sondern auch die Berücksichtigung aller verfügbaren `ContextAspects` und der logischen Ausdrücke, während die in Java integrierte Policy-Implementierung lediglich Code Sources und die JAAS-Implementierung zusätzlich Principals als Bedingungen zulassen.

  Für die Integration einer erlaubnisbasierten Zugriffskontrolle in JSEC müßte zunächst der `SecurityManagerAdapter` aus 5.4.2.2 als SecurityManager eingesetzt werden. Dies kann entweder mit dem Kommandozeilenparameter `-Djava.security.manager=edu.udo.jsec.adapters.SecurityManagerAdapter` beim Programmstart oder innerhalb der JSEC-Policy mit einer `SecurityManagerEngine` aus 5.4.5.4 unter der Bedingung `<access class="jsec.JsecPermission" target="*" action="executeMain"/>` konfiguriert werden. Damit wird erreicht, daß die ursprüngliche Access Control Policy aus 2.2.2 nicht mehr beachtet wird.

  Des weiteren müßte eine `JsecAccessControlEngine` mit dem Interface `AccessControlEngineSPI` implementiert werden, die anhand des `ExecutionContext` in der JSEC-Policy nach den nötigen Permissions für jede der Protection Domains des aktuellen Stack sucht. In der Policy könnten wiederum alle vorhandenen `ContextAspects` als Bedingungen für solche Permissions eingesetzt werden, die in den dazugehörigen Implications als `PermissionEngine` definiert werden könnten, nach denen `JsecAccessControlEngine` sucht. Damit wird die Erweiterungsfähigkeit der JSEC-Policy auf die erlaubnisbasierte Zugriffskontrolle übertragen.

- Die JSEC-Policy ist wie die Access Control Policy von Java zunächst statisch und richtet sich starr nach den einmal konfigurierten Engines. Für dynamische Sicherheitsmodelle, die in 2.1.2.4 vorgestellt wurden, ist es jedoch notwendig, die Entscheidungen über den Einsatz einer Engine anhand von zuvor durchgeführten Ressourcenzugriffen zu treffen.

  Dazu könnte ein `ContextAspect` implementiert werden, der in den Bedingungen der Policy verwendet werden kann und seine Entscheidung anhand vorausgegangener Zugriffe fällt. Die zuvor ausgeführten Zugriffe könnten durch Engines protokolliert werden, die mit einer `<true/>`-Condition bei jedem Zugriff eingesetzt werden.

  Eine solche Protokoll-Engine könnte beispielsweise den Zugriff auf eine sensible Datenbank registrieren. Um in der Policy festzulegen, daß nach diesem Zugriff sämtliche Netzwerkverbindungen nur noch verschlüsselt und an bestimmte Zieladressen durchgeführt werden sollen, kann der erwähnte `ContextAspect` in einer Condition eingesetzt

werden. Dieser aktiviert die Implication, sobald die Protokoll-Engine den kritischen Datenbankzugriff registriert hat.

- Remote Method Invocation (RMI) bietet im Gegensatz zu reiner Socket-Kommunikation die Metapher eines Methodenaufrufs für entfernte Objekte. [Wal99] enthält daher einen Vorschlag zur Erweiterung der Stack Inspection auf entfernte Methodenaufrufe. Dies könnte innerhalb von JSEC mit Wrappern für Klassen der RMI-Bibliothek integriert werden, die den auf dem aufrufenden Client festgestellten `ExecutionContext` an den Server übertragen und dort mit dem Thread assoziieren, in dem die Methode ausgeführt wird. Beim Zusammenstellen eines `ExecutionContext` in diesem Thread könnte ein dafür vorgesehener `ContextGenerator` diese Client-Informationen zusätzlich integrieren, um sie beim Auswerten der Policy berücksichtigen zu können.

- Entfernte Methodenaufrufe könnten auch im Namen und unter den Rechten eines aufrufenden Clients ausgeführt werden. In [BDS00] wird dazu eine Lösung vorgestellt, die das per SSL empfangene Zertifikat des Clients mit dem Server-Thread assoziiert, der die aufgerufene Methode ausführt. So können Server-Methoden herausfinden, wer sie aufgerufen hat. Dies könnte in JSEC in Form eines Wrappers für die Klassen integriert werden, die RMI-Methodenaufrufe auf dem Server initiieren. Um die Vergabe von Rechten und Bedingungen für Engines mit den gegebenen Möglichkeiten vornehmen zu können, sollte dieser Wrapper die RMI-Methoden mittels `Subject.doAs` ausführen, wobei das ausführende Subject mit Informationen über den aufrufenden Client in Form von Principals ausgestattet wird. So können JSEC und die Java-Zugriffskontrolle ihre Entscheidungen anhand dieser Principals fällen, die bereits in den Policies angegeben werden können.

- [HMM00] schlägt zudem eine Methode vor, wie auch an unbekannte Clients solche Principals, die Rollen darstellen, vergeben werden können. Die Rollen werden dabei auf dem Server definiert und anhand eines mittels einer Logik formulierbaren Regelsatzes an die aufrufenden Clients vergeben, wobei Referenzen berücksichtigt werden, die diese unbekannten Clients vorweisen können. Dies stellt eine Erweiterung der beispielsweise bei SSL benutzten Zertifikate von beidseitig getrauten Zertifizierungsstellen dar, die zur Verbindung zweier Rechner zuvor vereinbart werden müssen, was eine Zusammenarbeit mit unbekannten Clients erschwert. Mit diesem Ansatz können also verschiedene Administrationsbereiche verbunden werden, da die lokal definierten Rollen nach ebenfalls definierbaren Regeln an Aufrufer aus der fremden Umgebung vergeben werden können.

  Auch für die Vergabe solcher Rollen an Subjects, die Clients darstellen, bietet sich die Implementierung innerhalb eines JAAS-LoginModules an, so daß diese Implementierung unabhängig von den anderen Bestandteilen einer RMI-Erweiterung ausgetauscht werden kann. Die Authentifizierung des Clients und der `doAs`-Aufruf sollten daher in einer Engine ausgeführt werden, die vom RMI-Wrapper aus der JSEC-Policy angefordert und dann aufgerufen werden kann. So können die verschiedenen genannten Ansätze als Engines entwickelt und auch gemeinsam betrieben werden, ohne jeweils eine eigene RMI-Anpassung zu benötigen, die statt dessen von unabhängigen Wrappern bereitgestellt wird.

5.7 Fazit

Durch die Zusammenschaltung von Policy, Adaptern, Wrappern, ContextAspects und Engines können einzelne dieser Bausteine für spezielle Aufgaben entwickelt und in die Ausführung einbezogen werden.

Die Bestandteile der Standardimplementierung bieten einen Einsatz der für Java verfügbaren Erweiterungsbibliotheken JCE, JSSE und JAAS; deren Flexibilität und Erweiterbarkeit wird durch JSEC unterstützt und bleibt damit erhalten. JSEC ist aber nicht auf bestimmte Sicherheitsmechanismen beschränkt, sondern kann durch neue Bausteine auch zusätzliche Funktionalität für andere Bereiche verfügbar machen.

Mit Wrapping kann JSEC ohne Änderungen an bereits eingesetzter Software in bestehende Systeme integriert werden. Auch ohne Wrapping ist der Programmieraufwand auf die Ersetzung bestimmter Systemklassen durch Adapter beschränkt. In beiden Fällen bietet JSEC die Konfiguration und Durchsetzung von Sicherheitsspezifikationen durch Ausführung zusätzlicher Funktionalität anhand der festgelegten JSEC-Policy. Die Sicherheitspolitik ist damit nicht mehr auf Zugriffskontrolle beschränkt, sondern umfaßt auch die zusätzlich vorhandenen und zukünftigen Sicherheitsmechanismen.

JSEC überträgt die in 2.2.2 genannten Eigenschaften der Java-Zugriffskontrolle auf die Aktivierung weiterer Sicherheitsmechanismen. Diese Eigenschaften sind in JSEC folgendermaßen ausgeprägt:

- *Flexibel:* JSEC enthält kein fest eingebautes Sicherheitsmodell, sondern kann flexibel auf die individuelle Situation bei einem Einsatz abgestimmt werden. Für spezielle Anforderungen können neue Implementierungen einzelner Bausteine integriert werden und mit anderen, bereits bestehenden Teilen zusammenarbeiten.
- *Politikgetrieben:* Die zusätzlich ausgeführte Funktionalität und die Aktivierungsbedingungen werden in einer Sicherheitspolitik festgelegt. Die eingesetzten Sicherheitsmechanismen müssen nicht in der Anwendung implementiert werden, sondern werden anhand dieser Spezifikation zugeschaltet und parametrisiert.
- *Erlaubnisbasiert:* Diese Eigenschaft ist für JSEC nicht ganz zutreffend, da Zugriffserlaubnis oder -verweigerung nicht die Zielsetzung sind. Immerhin werden die Permission-Klassen in der Policy zur Definition von Zugriffen eingesetzt, im Gegensatz zur Access Control Policy von Java jedoch nicht als Folgerung, sondern als Bedingung eingesetzt.
- *Feinkörnig:* Da die bekannten Permissions verwendet werden können, sind die Bedingungen ebenso feinkörnig spezifizierbar wie die Java Access Control Policy. Für die angewendeten Engines können alle benötigten Parameter in der JSEC-Policy festgelegt werden.
- *Erweiterbar:* Die Erweiterbarkeit ist ein Hauptziel der gewählten JSEC-Architektur und wurde in Abschnitt 5.6 bereits an einigen Erweiterungsvorschlägen illustriert. Zudem ist die gewählte Integration der Policy durch den erweiterbaren ExecutionContext auch ohne

vollständigen Austausch der Policy-Implementierung auf die Berücksichtigung zusätzlicher Laufzeiteigenschaften als Bedingungen vorbereitet.

In der Architektur einer Anwendung stellt JSEC eine zusätzliche Schicht dar, deren Implementierung und Parametrisierung für die Anwendungs- und Komponentenentwickler nicht sichtbar ist. Sie können sich in ihren Projekten also auf die Entwicklung der Fachlichkeit konzentrieren, während ein Sicherheitsspezialist des Anbieters oder ein Sicherheitsadministrator des Anwenders anschließend Sicherheitsanforderungen ohne Veränderung der Software berücksichtigen kann.

6 Validierung

Die in den Kapiteln 4 und 5 vorgestellte Lösung wird in diesem Kapitel anhand einer bereits bestehenden Client-Server-Anwendung validiert, um die Einsetzbarkeit unter möglichst realen Bedingungen zu überprüfen. Gleichzeitig werden die Schritte vorgestellt, die zur Konfiguration einer Sicherheitsspezifikation führen, und die Auswirkungen dokumentiert, die zur Laufzeit bei der Durchsetzung dieser Spezifikation beobachtet werden.

Die eingesetzte Anwendung wurde von einer studentischen Projektgruppe der Universität Dortmund entwickelt und stellt ein System zur verteilten Ausführung von Workflows dar [BCC99]. Diese *eCCo VAA++* genannte Anwendung enthält zwei Geschäftsprozesse aus der Versicherungsbranche, die von der Projektgruppe implementiert wurden.

Die Anwendung eignet sich zur Validierung der Ergebnisse dieser Arbeit, da sie Dateizugriffe und Netzwerkverbindungen über Remote Method Invocation (RMI) und über Sockets ausführt, wobei keine Sicherheitsmechanismen zum Schutz der gespeicherten und übertragenen Daten berücksichtigt wurden. Die JSEC-Bibliothek soll die Sicherung dieser Ressourcenzugriffe ermöglichen.

6.1 Szenario

Für die Validierung wird das folgende Szenario angenommen: Die imaginäre *eCCo-Versicherungsgruppe* nutzt die Anwendung innerhalb ihrer zentralen Niederlassung zur Durchführung aller Geschäftsprozesse. Das Fehlen von Sicherheitsmechanismen wird dabei toleriert, da die Anwendung bisher lediglich auf Rechnern innerhalb des internen Netzes ausgeführt wird.

Um auch die Vertriebspartner in die elektronisch durchgeführten Geschäftsprozesse integrieren zu können, sollen nun deren Büro- und Außendienstrechner mit der Client-Software des Workflow-Systems ausgestattet werden und auf die Server in der Zentrale zugreifen. Die Anbindung soll dabei aus Kostengründen über das Internet erfolgen, womit eine Sicherung der Verbindungen notwendig wird.

Da die Anwendung in Java implementiert ist, kann sie in den sehr unterschiedlichen Umgebungen bei den verschiedenen Vertriebspartnern eingesetzt werden. Diese Inhomogenität macht jedoch eine Nutzung von Betriebssystemfunktionalität für Sicherheitsmechanismen wie

beispielsweise das in 1.2.3 erwähnte IPSEC unmöglich, da die Vielzahl von Plattformen eine zentrale Sicherheitsadministration erschwert und einige der eingesetzten Betriebssysteme keine Sicherheitsdienste anbieten.

Daher sollen die Sicherheitsmechanismen in die Java-Anwendung integriert werden, so daß Daten bereits vor dem Verlassen der Virtual Machine verschlüsselt werden und bei Netzwerkverbindungen innerhalb des Programms eine Authentifizierung der Gegenstelle ausgeführt werden kann.

Eine Veränderung der bestehenden Anwendung ist jedoch problematisch, da Komponenten nach VAA++-Spezifikation eingesetzt werden, die von externen Zulieferern entwickelt wurden und zukünftig in neuen Versionen weiterentwickelt werden, was eine dauerhafte Pflege eigener Varianten mit Sicherheitsfunktionalität erschwert. Zudem müßten die Entwickler im Bereich Sicherheit ausgebildet werden und alle Anwendungsteile auf eventuelle Sicherheitslücken untersucht werden. Externe Sicherheitsspezialisten müßten andererseits zunächst die Implementierung des Systems kennenlernen.

Daher soll die Bibliothek JSEC in Verbindung mit der Java Virtual Machine mit Wrapping-Erweiterung eingesetzt werden, um die bestehende Anwendung mit Sicherheitsmechanismen auszustatten. Diese separate Schicht der Anwendung kann leichter auf Sicherheitslücken untersucht werden, da beispielsweise kryptographische Schlüssel nicht von anderen Anwendungsteilen benötigt werden. Die Sicherheitspolitik kann von einem Sicherheitsadministrator ohne weitreichende Kenntnis der Anwendung definiert werden; dies wird in den folgenden Abschnitten beschrieben.

6.2 Untersuchung der Anwendung auf Ressourcenzugriffe

Um Sicherheitsmaßnahmen für Ressourcenzugriffe spezifizieren zu können, muß zunächst bekannt sein, welche Zugriffe die Anwendung durchführt. Kann diese Information nicht von den Entwicklern oder durch Untersuchung der Quelltexte bereitgestellt werden, so können mit JSEC die ausgeführten Ressourcenzugriffe zur Laufzeit in Testläufen protokolliert werden.

Ein solches Protokoll kann nur Zugriffe erfassen, die bei der Ausführung in der Testumgebung tatsächlich ausgeführt werden. Nicht ausgeführte Programmteile oder ein anderer Programmzustand könnten zu abweichenden Ressourcenzugriffen führen. Da die unterschiedlichen Geschäftsprozeßbausteine jedoch keine eigenen Ressourcen verwalten, sondern die Dienste des Workflow-Systems nutzen, sind bei der Ausführung beliebiger Geschäftsprozesse stets dieselben Zugriffe des Systems zu erwarten. Weitere Zugriffe sollen durch den Einsatz eines SecurityManagers verhindert werden, so daß Daten die Virtual Machine nur auf den erlaubten und gesicherten Wegen verlassen können.

Zur Protokollierung von Ressourcenzugriffen stehen in JSEC beispielsweise `FileLogEngine` und `SocketLogEngine` zur Verfügung. Da die Anwendung aber mit SecurityManager ausgeführt werden soll und damit die Konfiguration einer Access Control Policy notwendig

wird (siehe 2.2.2), müssen nicht nur Ressourcenzugriffe, sondern auch alle weiteren benötigten Permissions festgestellt werden. Dazu kann die Anwendung in einer Testumgebung mit dem `SecurityManagerAdapter` aus 5.4.2.2 ausgeführt werden und eine `AccessControlLogEngine` aus 5.4.5.4 zur Aufzeichnung aller nötigen Permissions eingesetzt werden, ohne die Programmausführung durch Zugriffsverletzungen bei jeder fehlenden Permission zu unterbrechen.

In Anhang A.5.1 ist eine JSEC-Policy für Testläufe angegeben, bei denen eine solche Log-Datei mit allen fehlenden Permissions erzeugt wird. Mit den dabei gewonnenen Informationen kann nicht nur die in Anhang A.5.2 dargestellte Access Control Policy erstellt werden; es können auch die Ressourcenzugriffe untersucht werden, um die Sicherheitsziele für JSEC festzulegen.

6.3 Definition der Sicherheitsziele

Die Sicherheitsziele werden anhand der festgestellten Ressourcenzugriffe für das vorgestellte Szenario wie folgt definiert: Alle von der Anwendung aufgebauten Netzwerkverbindungen zwischen externen Clients und den Servern in der Zentrale sollen durch Verschlüsselung und Message Authentication sowie durch Server- und Client-Authentifizierung gesichert werden. So sind einerseits die Daten während der Übertragung über das offene Internet gegen Abhören und Veränderungen geschützt, und andererseits ist sichergestellt, daß die Clients mit dem tatsächlichen Server verbunden werden; zudem können nur solche Clients die Dienste des Servers in Anspruch nehmen und an den Geschäftsprozessen teilnehmen, die einen privaten Schlüssel mit einem Zertifikat besitzen, das von der Versicherung ausgestellt wurde.

Um Geschwindigkeitseinbußen bei Verbindungen im internen Netzwerk zu vermeiden, könnte für solche Verbindungen auf eine Sicherung verzichtet werden. Die vorliegende JSEC-Implementierung bietet diese Möglichkeit jedoch nicht, da für externe und interne Clients nur ein Server-Socket geöffnet wird, das entweder als gesichert oder ungesichert konfiguriert werden kann. Mit einer `ServerSocket`-Implementierung, die beim Verbindungsaufbau je nach Herkunft des Clients entscheidet, ob die Verbindung gesichert werden soll, könnte diese Anforderung jedoch berücksichtigt werden. Eine `ServerSocketFactoryEngineSPI`-Implementierung, die solche Sockets erzeugt, kann dann in der JSEC-Policy konfiguriert werden.

Wegen der Internet-Anbindung der Server sollen die darauf abgelegten Daten zusätzlich gesichert werden. Dabei handelt es sich einerseits um Datenbanken, in denen die Geschäftsobjekte abgelegt werden, und andererseits um Dateien, die den Zustand des Workflow-Systems enthalten. Da eine Verschlüsselung von Datenbankfeldern wenig sinnvoll erscheint, wird die Datenbank statt dessen auf einem Rechner abgelegt, der vom Internet durch eine Firewall abgetrennt ist. Sind andere Sicherungsmaßnahmen für Datenbankzugriffe verfügbar, so sollten diese durch Wrapper für →JDBC-Klassen in JSEC integriert werden können. Die genannten Dateien sollen geschützt werden, da sie auch kritische Daten wie beispielsweise Kennwörter des Mail-Systems enthalten.

6.4 Konfiguration der Sicherheitspolitik

Die Sicherheitsziele sollen durch den Einsatz von JSEC erreicht werden. Die Java Virtual Machine mit Wrapping-Erweiterung wird mit der Wrapping-Definition aus Anhang A.4.4 konfiguriert, so daß die nötigen Wrapper zugeschaltet werden.

Die JSEC-Policy zur Konfiguration der gewünschten Sicherheitsmechanismen besteht aus drei Teilen: Neben separaten XML-Dateien für Server (A.5.4) und Clients (A.5.5) werden gemeinsame Definitionen in einer weiteren Datei vorgenommen (A.5.3).

Beim Programmstart wird `java.lang.SecurityManager` aktiviert und die Sun-Provider für JCA, JCE und JAAS geladen. Zur Überwachung des Systems werden Netzwerk- und Dateizugriffe protokolliert, wobei die Programm- und die JSEC-Konfigurationsdateien dabei ausgenommen sind.

Auf den Clients wird für alle Socket-Verbindungen SSL vorgeschrieben, wobei der Algorithmus TLS und sofortiger Handshake beim Aufbau der Verbindung definiert sind. Auch auf dem Client werden `ServerSockets` verwendet, weshalb auch dafür ein Konfigurationseintrag in der Client-Policy enthalten ist.

Der in beiden Fällen benutzte private Schlüssel mit dem Namen `keyclient` wird im Keystore `jsec.client.jceks` erwartet. Der Truststore `jsec.trust.jceks` ist ebenfalls ein Keystore und enthält das Zertifikat der Versicherung als Zertifizierungsstelle; eine Verbindung wird nur zu Servern aufgebaut, deren Schlüssel mit dem zu diesem Zertifikat gehörenden privaten Schlüssel signiert wurden.

Die zum Öffnen der Keystores und Lesen des privaten Schlüssels notwendigen Kennwörter sind nicht in der Konfigurationsdatei enthalten, sondern werden mit Swing-Dialogen vom Benutzer erfragt. Die Credentials, also Keystores, Schlüssel und Zertifikate, werden von einem `GlobalCredentialManager` im Arbeitsspeicher zwischengespeichert und entweder per Klick auf einen Swing-Logout-Button oder im Beispiel nach 4 Minuten wieder gelöscht.

Auf Server-Seite wird ebenfalls SSL für alle Sockets vorgeschrieben, wobei eine Client-Authentifizierung gefordert wird, so daß nur Clients akzeptiert werden, die ebenfalls SSL einsetzen und das im Truststore des Servers vorhandene Zertifikat für ihren privaten Schlüssel vorweisen können. Die Kennwörter für die Schlüssel sind in diesem Beispiel in der Konfigurationsdatei hinterlegt, so daß der Server ohne Benutzereingriff neu gestartet werden kann. Durch Angabe von `NeverLogout` wird erreicht, daß die Schlüssel während der Lebenszeit der Server-JVM im Arbeitsspeicher bleiben und nicht neu geladen werden müssen.

Zusätzlich werden die auf dem Server vorhandenen Dateien `ParameterSystem.dat` und `WorkflowManager.dat` mit dem Algorithmus DES bzw. Blowfish verschlüsselt. Die dazu nötigen geheimen Schlüssel `keydes1` und `keyblowfish2` werden aus dem Keystore `jsec.jceks` gelesen und ebenso wie die SSL-Credentials dauerhaft im Arbeitsspeicher gehalten. Auch die Kennwörter für diese Schlüssel sind in der Konfigurationsdatei eingetragen.

Die Nutzung der Sicherheitsmechanismen kann durch Änderung der dazu vorgesehenen Expressions aus der gemeinsamen Konfigurationsdatei ein- und ausgeschaltet werden. Die Expression `filesecurity` steuert die Ver- und Entschlüsselung der Dateien, `netsecurity` die Nutzung von SSL und `filelog` bzw. `netlog` die Protokollierung von Datei- bzw. Netzwerkzugriffen.

Da beim ersten Einsatz von JSEC die Dateien noch in unverschlüsselter Form vorliegen, kann beim ersten Lesen keine Entschlüsselung durchgeführt werden. Daher kann die Entschlüsselung durch Setzen von <true/> in der Expression `INSTALL` bei der ersten Ausführung deaktiviert werden; die Verschlüsselung wird jedoch durchgeführt, so daß die Dateien nach der ersten Ausführung verschlüsselt vorliegen. Die Berücksichtigung dieser Einstellung wird durch logische Ausdrücke innerhalb der JSEC-Policy ermöglicht.

Die Kommandozeilenparameter zum Start von Server und Client wurden verändert, um die Wrapper zu aktivieren und die JSEC-Policy-Dateien zu laden. Bei der Ausführung hat sich gezeigt, daß die `main`-Klasse des Clients nicht mit `wrapmain` gewrappt werden kann. Während der Bearbeitungszeit konnte dieses Problem nicht mehr gelöst werden, so daß statt des `MainMethodWrappers` der `MainMethodAdapter` explizit in der Kommandozeile eingesetzt wird. Da die Kommandozeile aber ohnehin verändert wurde, wird dieser Nachteil nicht als gravierend eingeschätzt.

An der oben bereits erwähnten Access Control Policy für den Security Manager (Anhang A.5.2) ist zu sehen, daß die Anwendung auch bei einem Einsatz von JSEC keine Zugriffsrechte auf die Keystores, Log- oder JSEC-Policy-Dateien benötigt (siehe auch 5.5). Diese Rechte müssen lediglich an JSEC vergeben werden (Anhang A.4.7).

6.5 Durchsetzung der Sicherheitspolitik

Die Ausführung der Sicherheitsmechanismen wurde während der Ausführung der Anwendung mit JSEC kontrolliert. Folgende Auswirkungen von JSEC auf die Ausführungsgeschwindigkeit sind zu erwarten und wurden bei der Validierungsanwendung bestätigt:

- Insgesamt geringere Ausführungsgeschwindigkeit durch Verwendung der Classic JVM statt der Hotspot JVM – diese kann durch Übertragung der Änderungen aus Anhang A.1 aber ebenfalls mit Wrapping erweitert werden, um diesen Nachteil zu beheben.
- Geringfügig erhöhte Latenzzeit durch Abfrage der JSEC-Policy bei jedem Öffnen einer Ressource, auch wenn diese nicht gesichert werden soll. Dies konnte nur während des Programmstarts beobachtet werden, wenn viele Klassendateien geladen werden und diese nicht in einer einzelnen JAR-Datei vorliegen. Allerdings führt bereits die Aktivierung des SecurityManagers zu einer Verlangsamung dieser Zugriffe. Um die zweifache Verzögerung durch SecurityManager und JSEC zu vermeiden, könnte die erlaubnisbasierte Zugriffskontrolle eventuell in JSEC integriert werden (siehe auch 5.6).

- Größere Wartezeiten beim ersten Ressourcenzugriff, der mit kryptographischen Algorithmen modifiziert wird. Spätere Zugriffe werden ohne feststellbare Verzögerung durchgeführt. Laut JCE-Dokumentation [Sun00d] liegt dies an der Initialisierung von Zufallszahlen.
- Verzögerung beim Verbindungsaufbau mit einem neuen SSLContext durch Ausführung asymmetrischer Algorithmen und Schlüsselaustausch beim Handshake zwischen Client und Server. Bei weiteren Verbindungen, die aufgrund gleicher Optionen mit einem im CredentialManager bereits vorhandenen SSLContext aufgebaut werden können, ist der Handshake deutlich schneller beendet.
- Geringerer Durchsatz bei der Datenübertragung von langsamen Clients mit schneller Netzanbindung. Die mögliche Übertragungsrate könnte durch die Ausführungsgeschwindigkeit der symmetrischen Algorithmen auf langsamen Rechnern begrenzt werden.
- Höhere Serverbelastung und langsamere Ausführung von Server-Methoden durch zusätzliche kryptographische Algorithmen für alle zugreifenden Clients.

Die letzten vier Punkte können durch andere Algorithmen-Implementierungen ausgeglichen werden, die auch als Native Code oder in Hardware ausgeführt werden können. Gerade auf Server-Seite könnte der Einsatz von Verschlüsselungs-Hardware große Vorteile bei der parallelen Bearbeitung vieler Client-Anfragen bieten. Durch den SSL-Standard können die Algorithmen auf der Client-Seite weiterhin in Software ausgeführt werden und Daten mit der Hardware-Implementierung austauschen.

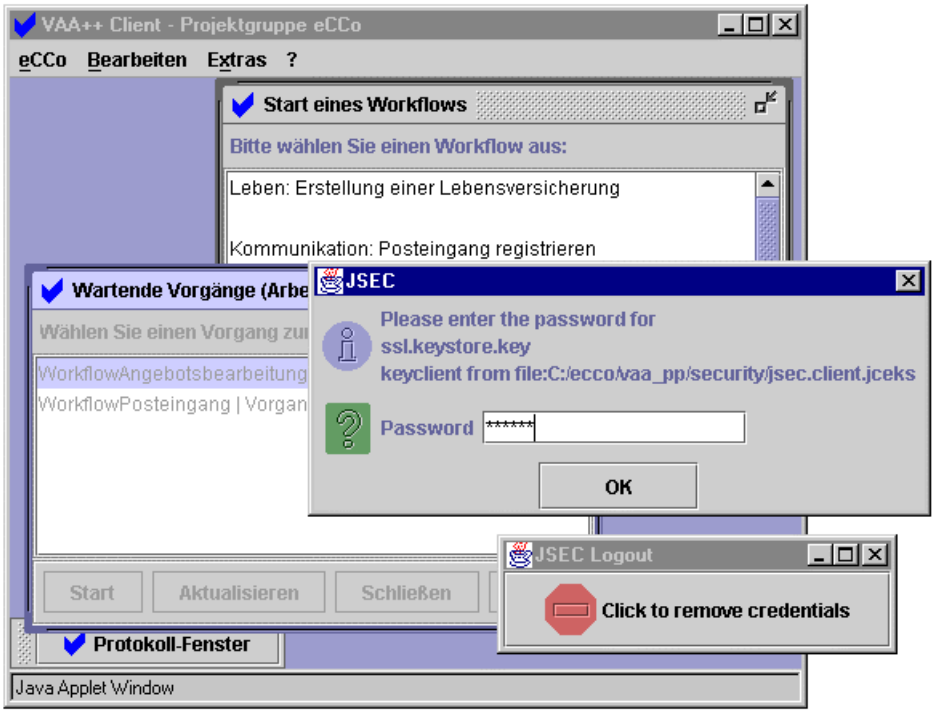

Abbildung 6-1: Validierungsanwendung mit zusätzlichen JSEC-Dialogen

Neben der Ausführungsgeschwindigkeit wurde auch die Anwendung der Verschlüsselungsmechanismen überprüft. Abbildung 6-1 zeigt einen Screenshot der Anwendung auf einem Client mit zwei zusätzlichen Fenstern, die von JSEC zur Eingabe eines Kennworts und zum manuellen Löschen der Credentials erzeugt wurden; diese Fenster wurden wie in A.5.5.1 konfiguriert mit der Swing-Bibliothek erstellt.

Das Fenster der Anwendung enthält den Hinweis „Java Applet Window“, da die Anwendung gemäß A.5.2 keine `java.awt.AWTPermission "showWindowWithoutWarningBanner"` besitzt; JSEC besitzt diese Permission wie in A.4.7 festgelegt und kann seine Fenster daher ohne diesen Hinweis darstellen. Wird die Permission ausschließlich an JSEC vergeben, so kann verhindert werden, daß andere Programmteile die Kennwörter durch Täuschung des Benutzers erschleichen.

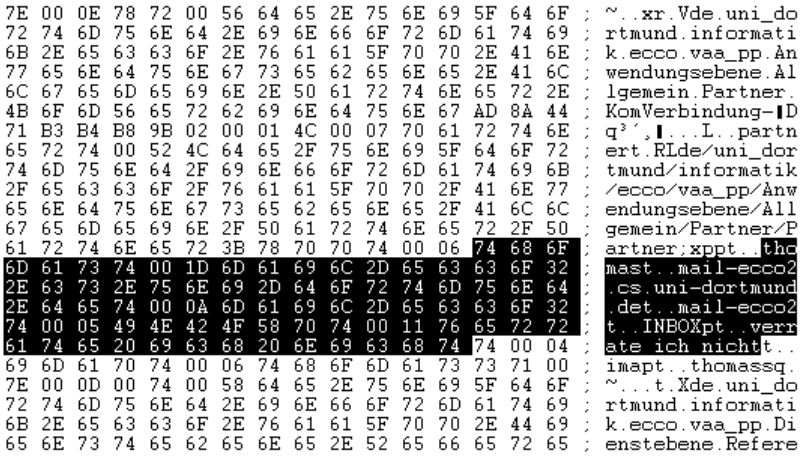

Abbildung 6-2: Serialisierter Zustand des Parametersystems in unverschlüsselter Datei

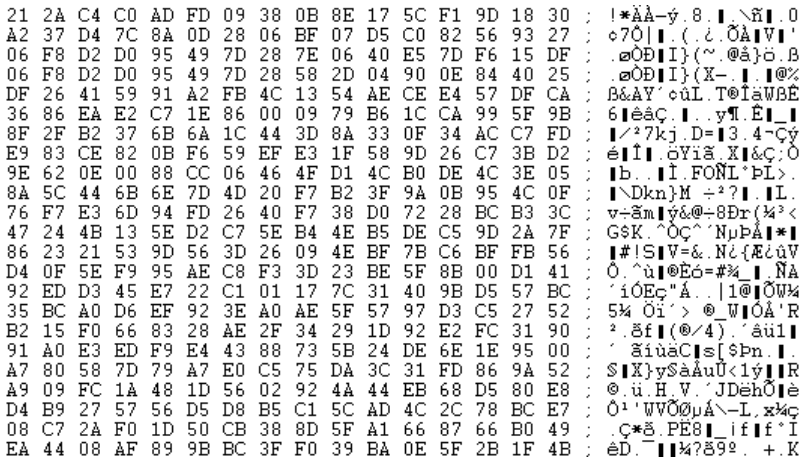

Abbildung 6-3: Serialisierter Zustand des Parametersystems in Datei mit DES-Verschlüsselung

```
6F 72 6D 61 74 69 6B 2E 65 63 63 6F 2E 76 61 61 ; ormatik.ecco.vaa
5F 70 70 2E 53 74 65 75 65 72 75 6E 67 73 65 62 ; _pp.Steuerungseb
65 6E 65 2E 41 6E 70 61 73 73 75 6E 67 2E 56 65 ; ene.Anpassung.Ve
72 73 69 63 68 65 72 75 6E 67 2E 4C 65 62 65 6E ; rsicherung.Leben
2E 56 6F 72 67 61 65 6E 67 65 2E 56 6F 72 67 61 ; .Vorgaenge.Vorga
6E 67 45 72 66 72 61 67 65 55 6E 74 65 72 73 63 ; ngErfrageUntersc
68 72 69 66 74 53 70 61 65 74 65 72 70 70 70 70 ; hriftSpaeterpppp
70 70 70 70 00 74 00 73 64 65 2E 75 6E 69 5F 64 ; pppp.t.sde.uni_d
6F 72 74 6D 75 6E 64 2E 69 6E 66 6F 72 6D 61 74 ; ortmund.informat
69 6B 2E 65 63 63 6F 2E 76 61 61 5F 70 70 2E 53 ; ik.ecco.vaa_pp.S
74 65 75 65 72 75 6E 67 73 65 62 65 6E 65 2E 41 ; teuerungsebene.A
6E 70 61 73 73 75 6E 67 2E 56 65 72 73 69 63 68 ; npassung.Versich
65 72 75 6E 67 2E 4C 65 62 65 6E 2E 56 6F 72 67 ; erung.Leben.Vorg
61 65 6E 67 65 2E 56 6F 72 67 61 6E 67 56 65 72 ; aenge.VorgangVer
73 65 6E 64 65 41 6E 74 72 61 67 71 00 7E 00 4C ; sendeAntragq.~.L
71 00 7E 00 23 70 70 70 70 70 70 70 73 71 00 7E ; q.~.#ppppppsq.~
00 08 00 00 00 00 00 00 00 01 75 71 00 7E 00 0B ; ..........uq.~..
00 00 00 0A 73 71 00 7E 00 1D 00 00 00 1E 73 71 ; ....sq.~......sq
00 7E 00 08 00 00 00 00 00 00 00 01 75 71 00 7E ; .~..........uq.~
00 0B 00 00 00 0A 71 00 7E 00 28 70 70 70 70 70 ; ......q.~.(ppppp
70 70 70 70 73 71 00 7E 00 08 00 00 00 00 00 00 ; ppppsq.~........
00 01 75 71 00 7E 00 0B 00 00 00 0A 71 00 7E 00 ; ..uq.~......q.~.
```

Abbildung 6-4: Serialisierter Zustand des Workflowmanagers in unverschlüsselter Datei

```
19 34 5A D5 9A CE 3E AB AC 14 2A 05 55 96 6D E2 ; .4ZO▮I>«¬.*.U▮mâ
2E 7A E2 05 29 D3 4A A7 EF EF B7 BC FF 05 C8 82 ; .zâ.)ÓJ§ïï·¼ÿ.È▮
40 75 B7 C8 77 44 88 2E 1B D6 D2 16 E3 2C B4 2E ; @u·ÈwD▮..ÖÒ.ã,´.
C6 61 17 E0 7B DC 9C 4D 19 B7 65 83 8E 4A DD AC ; Æa.à{Ü▮M.·e▮▮JÝ¬
9E 72 57 03 6C 7B 21 B4 8E A0 C0 1C 77 8F 9A 4A ; ▮rW.l{!´▮ À.w▮▮J
24 C7 1D EC 53 03 EF 20 66 0D 00 4C 2D C1 BB E5 ; $Ç.iS.ï f..L-Á»å
2F 07 C3 CA 5A 79 7B CF 25 60 14 5A A7 9E C3 DE ; /.ÃÊZy{Ï%`.Z§▮ÃÞ
70 FC 97 2D 3E 26 29 5E 23 9E 54 E9 5B 4A C0 E5 ; pü▮->&)^#▮Té[JÀå
D5 16 BA 1F 89 BF 3B 09 9C 4D 80 5A E2 B2 D9 00 ; Õ.º.▮¿;.▮M▮Zâ²Ù.
8A 77 F3 A3 2F B8 3B D6 AE A6 4B 04 10 CF 97 49 ; ▮wó£/¸;Ö®¦K..Ï▮I
A8 16 E7 CA CC 72 3F C0 EB CC 2C CB 66 18 1F F3 ; ¨.çÊÌr?ÀëÌ,Ëf..ó
C5 DB 13 28 50 72 21 07 20 6B 9C 30 DD EB 87 DC ; ÅÛ.(Pr!. k▮0Ýë▮Ü
C1 EA 73 4A FE 27 FA 25 90 9C 4A E6 C0 DF 43 3C ; ÁêsJþ'ú%▮▮JæÀßC<
EB 8B 26 B7 9A 9E 06 0B 9C B4 CA 62 92 84 6C FA ; ë▮&·▮▮..▮´Êb´▮lú
FD EE 1B 92 89 A4 EA 30 B4 77 1F 07 8D F6 45 24 ; ýî.´▮¤ê0´w..▮öE$
75 01 29 9B E4 D3 A7 8C 1D C6 31 C8 DB AD 11 F1 ; u.)▮äÓ§▮.Æ1ÈÛ-.ñ
A0 F6 A7 83 A7 EE A4 C1 40 74 53 3A 01 B8 AE 69 ;  ö§▮§î¤Á@tS:.¸®i
9F FD EE 16 1A 6C 6B EC B9 20 92 8F 04 C8 0E 38 ; ▮ýî..lkì¹ ´▮.È.8
97 A3 F8 98 B2 B6 87 EE D8 80 8B 4E 33 E3 18 A0 ; ▮£ø▮²¶▮îØ▮▮N3ã. 
3A F0 49 08 72 AD BF DA 3B 4A ED 87 44 9B 88 DA ; :ðI.r-¿Ú;Jí▮D▮▮Ú
40 B9 AE C8 27 FA BA 65 71 0F 04 66 47 D7 05 1E ; @¹®È'úºeq..fG×..
80 79 8B 44 87 DC 4B 25 92 CA AA 72 29 C0 AA B7 ; ▮y▮D▮ÜK%´Êªr)Àª·
```

Abbildung 6-5: Serialisierter Zustand des Workflowmanagers in Datei mit Blowfish-Verschlüsselung

Abbildung 6-2 und Abbildung 6-4 enthalten Ausschnitte der Dateien `ParameterSystem.dat` und `WorkflowManager.dat` in der ursprünglichen Version. In diesen Dateien sind serialisierte Objekte enthalten, die den Zustand des Anwendungssystems enthalten. Darin sind beispielsweise Zugangsdaten für E-Mail-Server und Klassennamen der ausgeführten Workflows zu erkennen. Abbildung 6-3 und Abbildung 6-5 zeigen Ausschnitte dieser Dateien bei einem Einsatz von JSEC mit der in 6.4 vorgestellten Sicherheitspolitik. Die Verschlüsselungsalgorithmen für die Dateizugriffe werden also wie gewünscht ausgeführt.

Um auch die Netzwerkverbindung zu überprüfen, wurden die übertragenen Pakete mit dem Werkzeug WinDump [DVR00] aufgezeichnet, einer Windows-Portierung von TCPdump [JLM89]. Abbildung 6-6 zeigt einen Ausschnitt der ohne JSEC aufgezeichneten Daten. Darin sind ebenso kritische Daten der Versicherungsnehmer enthalten wie Informationen über den Aufbau übertragener Objekte und Anfragen an die RMI-Registry zum Auffinden der Server-Methoden. Die Daten sind Gefahren durch Abhören, Veränderung oder Verbindung mit falschen Servern ausgesetzt. Bei Nutzung von JSEC werden die Nutzdaten nicht mehr im Klartext übertragen. Wie in Abbildung 6-7 zu sehen, werden dann aber zusätzlich Zertifikate von Server und Client in unverschlüsselter Form übertragen.

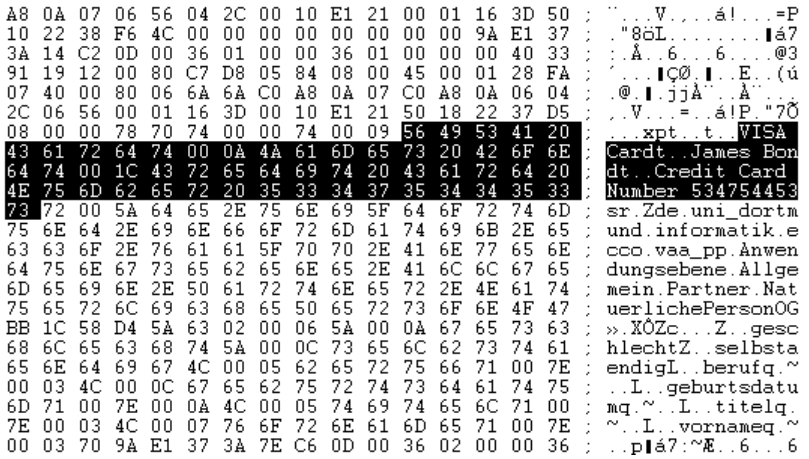

Abbildung 6-6: TCP-Paketaufzeichnung ohne sichere Sockets

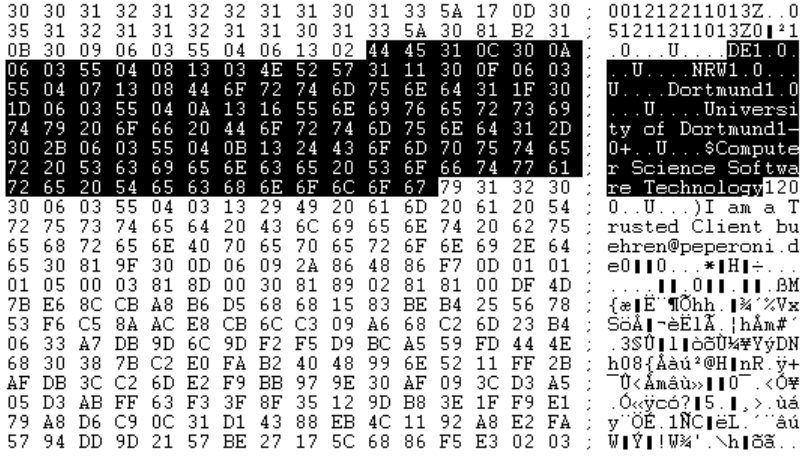

Abbildung 6-7: TCP-Paketaufzeichnung mit SSL-Sockets

Die gegenseitige Authentifizierung wurde durch Änderung der Systemzeit auf Server bzw. Client getestet, wodurch das jeweils andere Zertifikat durch Verlassen des Gültigkeitszeitraums als ungültig erkannt wurde. Bei einer ungültigen Zeiteinstellung auf einem Client wurde auf diesem eine `SSLException` mit der Zusatzinformation „untrusted server cert chain“ geworfen. Der Verbindungswunsch von Clients wurde bei verstellter Server-Zeit von diesem abgewiesen, was auf den Clients zu einer `SocketException` mit der Angabe „Connection aborted by peer“ führte. Wird in der JSEC-Policy des Servers aus A.5.4.1 die Option `ssl.needclientauth` auf `false` gesetzt, so werden auch Clients ohne gültiges Zertifikat akzeptiert.

Dieser exemplarische Einsatz mit einer bestehenden Anwendung hat also gezeigt, daß die in Kapitel 3 gestellten Anforderungen auch unter praxisnahen Bedingungen von der entwickelten Lösung erfüllt werden.

7 Fazit und Ausblick

Im Rahmen dieser Arbeit wurde die Erweiterung der Java Virtual Machine um einen Wrapping-Mechanismus sowie die erweiterbare und konfigurierbare JSEC-Bibliothek mit zusätzlicher Sicherheitsfunktionalität entwickelt. Beide Teile wurden gemeinsam zur Konfiguration und Durchsetzung von Sicherheitsspezifikationen ohne Änderungen an bestehenden Anwendungen eingesetzt. Sie können aber auch unabhängig voneinander einzeln weiterverwendet werden.

Mit der vorgenommenen Erweiterung des Java-Sicherheitsmodells kann die Funktionalität einer Anwendung entwickelt werden, ohne die Sicherheitsmechanismen im Quellcode berücksichtigen zu müssen. Bei der Validierung anhand einer bestehenden Client-Server-Anwendung hat sich gezeigt, daß die Installation der erweiterten Java Virtual Machine und die Definition einer Sicherheitsspezifikation ausreichen, um verschiedene Arten von Zugriffen auf Systemressourcen um die gewünschten Sicherheitsmechanismen zu erweiteren.

Das Wrapping nutzt dabei konsequent die Flexibilität des Lademechanismus der Java Virtual Machine und erweitert die Möglichkeiten der Objektorientierung, um Änderungen an bestehenden Software-Systemen und der Java-Klassenbibliothek vorzunehmen.

Um die der Java-Laufzeitumgebung hinzugefügten Mechanismen auch auf weiteren Betriebssystemplattformen oder in Verbindung mit einer Hotspot Virtual Machine [DG98] einsetzen zu können, ist lediglich eine Portierung der in Anhang A.1 dokumentierten Änderungen an der Java Virtual Machine zum Einbau des Wrapping-Mechanismus nötig. Die übrigen Teile sind darauf aufbauend vollständig in Java und somit plattformunabhängig implementiert.

7.1 Mögliche Erweiterungen der prototypischen Entwicklung

Die JSEC-Sicherheitsbibliothek aus Kapitel 5 wurde anhand der in dieser Arbeit gestellten Anforderungen konzipiert und wendet die für Java verfügbaren Sicherheitsmechanismen an. Bei zusätzlichen Anforderungen kann sie um weitere Funktionalität erweitert werden, was bei der Entwicklung bereits berücksichtigt wurde. In Abschnitt 5.6 wurden bereits Vorschläge zur Nutzung dieser Erweiterungsmöglichkeiten vorgestellt.

Der Wrapping-Mechanismus aus Kapitel 4 kann auch unabhängig vom Thema Sicherheit für weitere Einsatzzwecke verwendet werden, die in Abschnitt 4.5.1 bereits vorgestellt wurden und teilweise eine Erweiterung des vorhandenen Mechanismus voraussetzen.

Neben den in Abschnitt 5.4.3 vorgestellten Wrappern für die sicherheitsspezifische Erweiterung ausgewählter Systemklassen können für die Klassen der Java-Klassenbibliothek weitere universell einsetzbare Wrapper mit verschiedenen Funktionen entwickelt werden. Wrapper könnten auch als Grundlage zur Realisierung anderer Sicherheitsmodelle dienen, die Ressourcen und Klassen oder Objekte um zusätzliche Sicherheitsattribute erweitern müssen, beispielsweise Sicherheitsstufen bei Mandatory Access Control.

Zudem können auch anwendungsspezifische Wrapper angefertigt werden, um einzelne Anwendungsklassen oder Komponenten für den Einsatz in einem unvorhergesehenen Umfeld anzupassen; dazu ist lediglich der Einsatz des bestehenden Wrapping-Mechanismus nötig.

Eine über die Validierung hinausgehende Überprüfung der Vollständigkeit, Robustheit und Sicherheit der gewählten Implementierung von Wrapping und JSEC-Bibliothek war im Rahmen dieser Arbeit nicht möglich. Diese weitere Prüfung ist ebenso vorstellbar wie eine Berücksichtigung des Wrapping in der Spezifikation des Java-Lademechanismus [Gol98] zur formalen Verifikation der gewünschten Eigenschaften.

Ebenfalls denkbar ist eine über den Rahmen dieser Arbeit hinausgehende Untersuchung der Auswirkungen des Wrapping-Konzepts auf den Software-Entwicklungsprozeß sowie auf den Konfigurationsaufwand beim Einsatz der Software.

7.2 Werkzeugunterstützung zur Konfiguration

Die in Java vorhandenen und in anderen Arbeiten vorgestellten Sicherheitsmechanismen können ähnlich wie die hier vorgestellte Lösung in vielen Aspekten konfiguriert werden, was einen Einsatz in unterschiedlichen Situationen und speziell bei JSEC die Entwicklung einzelner, kleiner Bausteine und deren individuelle Verknüpfung ermöglicht.

Zusätzlich zu den Konfigurationen werden Informationen über Benutzer sowie deren Authentifizierungsmerkmale, Rollen und kryptographische Schlüssel benötigt. Auch die Konfiguration der Betriebssystemumgebung, in denen die Java-Anwendungen ausgeführt werden, hat einen Einfluß auf die Sicherheitseigenschaften der Anwendungen und ist teilweise direkt mit diesen gekoppelt, wie beispielsweise bei der Wiederverwendung von Anmeldeinformationen für Single Login.

Die Sicherheitspolitik für ein konkretes Einsatzszenario muß von einem Sicherheitsadministrator entwickelt und eingesetzt werden können. Bei der Vielzahl von Möglichkeiten besteht jedoch die Gefahr, den Überblick zu verlieren und die Auswirkungen gewählter Einstellungen im Zusammenspiel mit anderen Optionen nicht mehr nachvollziehen zu können. Daher

scheint zur Beherrschung der komplexen Sicherheitseigenschaften solcher Systeme eine Werkzeugunterstützung dringend notwendig zu sein.

Auch ein solches Werkzeug muß umfangreiche Erweiterungsmöglichkeiten bieten, um den Administrator bei der Konfiguration der ebenfalls erweiterbaren Sicherheitsmechanismen unterstützen zu können. Dazu wäre eine Zusammenarbeit des Konfigurationswerkzeugs mit den eingesetzten Sicherheitsbausteinen hilfreich, um deren Auswirkungen im Werkzeug automatisiert berücksichtigen zu können. Eventuell sollten die einzelnen Sicherheitskomponenten ähnlich wie Java Beans eine Schnittstelle für solche Werkzeuge erhalten, um eine Zusammenarbeit des Werkzeugs auch mit unbekannten Komponenten zu ermöglichen.

Das Werkzeug sollte als graphischer Konfigurationseditor mit intuitiv verständlicher Bedienung und Visualisierung der netzwerkweiten Folgen von Sicherheitskonfigurationen ausgeführt werden. Ein solcher Editor sollte den Administrator von Details jedes einzelnen Rechners und jedes einzelnen installierten Software-Pakets möglichst weit entlasten und statt dessen die Modellierung einer übergreifenden Sicherheitspolitik unterstützen. Wie in [PSW98] könnte die Politik abstrakte Schutzziele vorgeben, statt die einzelnen Mechanismen konkret festzulegen, die durch das Werkzeug anhand der abstrakteren Sicherheitspolitik automatisch ausgewählt werden könnten. Es sollte die nötigen Konfigurationsdateien aus dem eingegebenen Modell erzeugen und mit einem netzweiten Verteilungsdienst auf die involvierten Rechner verteilen können. In [SP98] wird zudem eine Lösung vorgestellt, mit der die Administration selbst netzwerkweit an mehrere Administratoren verteilt werden kann.

Denkbar ist auch eine Laufzeit-Schnittstelle zwischen Administrationswerkzeug und den Entwicklungen aus dieser Arbeit, so daß Konfigurationsänderungen beispielsweise auch zur Laufzeit ohne Neustart der Java Virtual Machine und damit der ausgeführten Anwendung ermöglicht werden. Mit zeitweiser oder dauerhafter Zuschaltung entsprechender Engines könnten auch relevante Meßdaten auf den Clients erfaßt und an das Administrationswerkzeug übermittelt werden; bei unbekannten Komponenten könnten die davon ausgeführten Ressourcenzugriffe festgestellt und die Definition einer passenden Sicherheitspolitik unterstützt und in Kooperation mit dem Administrationswerkzeug während einer ersten Ausführung durchgeführt werden.

Anhang

A.1 Änderungen an der Java Virtual Machine

Im folgenden wird die Implementierung der Änderungen an der Java Virtual Machine aus Kapitel 4 vorgestellt. Zu jeder Erweiterung wird der damit erreichte Effekt beschrieben und ein Ausschnitt des Quelltextes mit hervorgehobener Änderung aufgelistet. Beides soll einerseits die Implementierung erläutern und andererseits auch die Portierung auf eine andere Version der Java Virtual Machine erleichtern.

Die relevanten Quelltexte liegen im Verzeichnis src/share/javavm/runtime der JDK-Quelltexte. Bei den verwendeten Classic JVMs aus JDK 1.2.2 und 1.3.0 von Sun befinden sich alle Änderungen in C-Quelltexten; die Assembler-Teile der Virtual Machine bleiben unverändert. Sollten in zukünftigen oder anderen Virtual Machines zusätzliche Optimierungen durch Neuimplementierung von Teilen in Assembler vorgenommen werden, könnten sich die Änderungen ganz oder teilweise auch in den Assembler-Bereich verlagern. Da hier auch Kontext und Zweck jeder Änderung kurz erläutert werden, sollte eine Anpassung an solchermaßen veränderte Situationen möglich sein.

Es sind vor allem drei Arten von Änderungen notwendig. Dazu stehen in *TB_util.c* jeweils Funktionen zur Verfügung, die von den veränderten Programmteilen aufgerufen werden.

- *Ersetzung* der Namen von Klassen anhand des Schemas aus Abbildung 4-8. Diese wird durch die Funktion `char* TB_replacename(char* name)` implementiert.
- *Rückwärtsersetzung* der Namen in umgekehrter Richtung. Dazu steht die Funktion `char* TB_replacenamebackward(char* name)` zur Verfügung.
- *Prüfen der Wrapping-Beziehung zweier Klassen* zur Entscheidung über erweiterte Sichtbarkeiten von Methoden der gewrappten Klasse:

```
int TB_allowoverride( ClassClass* sub,
                      struct methodblock *submb,
                      ClassClass* super,
                      struct methodblock *supermb)
```

Neben diesen hauptsächlichen Funktionen werden noch weitere benötigt, die hier im jeweiligen Kontext erläutert werden. Das Präfix TB_ bei allen verwendeten Funktionen soll ein Auf-

finden innerhalb der Quelltexte der Java Virtual Machine erleichtern und die Änderungen eindeutig vom bestehenden Code abheben. Hier folgt nun eine Auflistung der vorgenommenen Erweiterungen.

- Zu Beginn der Initialisierung der Java Virtual Machine werden die zusätzlichen Kommandozeilenparameter zur Abschaltung des Wrapping und Angabe einer separaten Konfigurationsdatei ausgewertet sowie das Wrapping aktiviert. Diese Änderung betrifft die Funktion `Initialize12` in der Datei *javai.c*.

```
static jint Initialize12(JavaVMInitArgs *args, bool_t setDefault)
{
    int i;
    int TB_willwrap = 1;
    int TB_mustwrap = 0;

    TB_readwrappingdefinition(NULL);
...
    } else if (!strcmp(option->optionString, "abort")) {
        abort_hook = (void (JNICALL *)(void))(option->extraInfo);
    } else if (strcmp(option->optionString, "-TB_wrap:no")==0) {
        if ((!TB_mustwrap) && (TB_getwrapoptional())) {
            TB_willwrap = 0;
        } else {
            jio_fprintf(stderr, "-wrap:no is not allowed without \
                                 wrapoptional or with -wrap:yes\n");
            return JNI_EINVAL;
        };
    } else if (strcmp(option->optionString, "-wrap:yes")==0) {
        TB_willwrap = 1;
        TB_mustwrap = 1;
    } else if (!strncmp(option->optionString, "-wrap:", 6)) {
        if (TB_getwrapmore()) {
            TB_willwrap = 1;
            TB_readwrappingdefinition(option->optionString + 6);
        } else {
            jio_fprintf(stderr,
                "-wrap:filename is not allowed without wrapmore\n");
            return JNI_EINVAL;
        };
    } else if (!strncmp(option->optionString, "-Xbootclasspath/a:", …
...
    if (TB_willwrap) {
        TB_enablewrapping();
    };
    return JNI_OK;
}
```

- In der Sun-Implementierung der Java Virtual Machine gibt es die zentrale Funktion `FindClassFromClassLoader`, von der alle Anforderungen nach Referenzen auf andere Klassen bearbeitet werden. Dabei wird zunächst geprüft, ob die angeforderte Klasse bereits im LoaderCache vorhanden ist. Falls nicht, wird schließlich das Dynamic Class Loading in Gang gesetzt, um die Klasse zu beschaffen. Vorher wird aber die Ersetzung

des angeforderten Namens durchgeführt, um statt einer gewrappten Klasse den Wrapper zu laden.

Da vom Dynamic Class Loading eventuell mehrmals wieder auf diese Funktion zurückgegriffen wird, etwa wenn mehrere Class Loader beteiligt sind, muß verhindert werden, daß diese Ersetzung für ein- und dieselbe Klassenanforderung mehrfach durchgeführt wird. Darum wird der ersetzte Name während des Loading-Aufrufs gesperrt, so daß dieser nicht erneut ersetzt wird.

Im Falle eines rekursiven Aufrufs dieser Funktion muß ein gesperrter Name sogar rückwärts ersetzt werden, bevor eventuell im LoaderCache eines anderen Class Loaders danach gesucht wird, denn ein Wrapper wird in diesem Cache bereits unter dem Namen der gewrappten Klasse geführt.

Und so stellt sich diese Änderung in der Datei *classresolver.c* dar:

```
ClassClass *
FindClassFromClassLoader(...)
{
...
    self = EE2SysThread(ee);
    name = TB_replacenamebackward_iflocked(name);
...
    if (cb) {
...
    } else {
        name = TB_replacename(name);
        TB_replace_lock(name);
        if (verboseclassdep) {
            PrintVerboseClassDep(ee, name);
        }
        if (DisableAsyncEvents(ee) == 0) {
...
            if (EnableAsyncEvents(ee)) {
                cb = 0;
            }
        }
        TB_replace_unlock(name);
        if (cb) {
            char buf[MSG_BUF_LEN];
            char *ename;
...
    if (cb) {
        if (resolve) {
            InitClass(cb);
            if (exceptionOccurred(ee))
                return 0;
        }
        if (TB_loadagain(TB_origname)) {
            return FindClassFromClassLoader2(ee, TB_origname, resolve,
                                             loader, throwError, pd);
        } else {
```

```
            return cb;
        };
    } else {
        if (!exceptionOccurred(ee)) {
...
```

- Nach dem Laden wird eine interne Darstellung jeder Klasse erzeugt. Dabei findet eine Rückwärtsersetzung des Klassennamens statt, der in der Klassendatei notiert ist. So wird erreicht, daß die statt der gewrappten Klasse geladene Wrapper-Klasse im System unter dem ursprünglichen Namen geführt wird und damit bei folgenden Anforderungen benutzt wird. Eine gewrappte Klasse wird unter dem konstruierten Namen des Klassenrumpfes abgelegt, der bei Compilierung des Wrappers benutzt wurde. So kann sie vom Wrapper statt dieses Rumpfes angesprochen werden. Der Klassenrumpf selbst wird somit nie geladen.

  Diese Änderung wird an der Funktion `createInternalClass0` in *classload.c* vorgenommen:

```
/* Get the name of the class */
i = get2bytes(context);
ucb->name =  TB_replacenamebackward(
    GetClassConstantClassName(context->constant_pool, i));
context->constant_pool[i].clazz = cb;
...
for (i = ucb->methods_count, mb = ucb->methods; --i >= 0; mb++) {
...
    mb->args_size = Signature2ArgsSize(mb->fb.signature)
                    + ((mb->fb.access & ACC_STATIC) ? 0 : 1);
    if (
        (mb->args_size==1) &&
        (mb->fb.access & ACC_STATIC >0) &&
        (mb->fb.access & ACC_PUBLIC >0) &&
        (strcmp(mb->fb.name,"main")==0)
       )
    {
        char *newname = TB_change_mainclass_name(ucb->name);
        if (strcmp(ucb->name,newname)!=0) {
            ucb->name = newname;
        };
    };
    attribute_count = get2bytes(context);
...
}
```

- Der schon erwähnte LoaderCache enthält in einer offenen Hashing-Datenstruktur Referenzen der bereits geladenen Klassen und wird zuerst nach einer Klasse durchsucht, um mehrfache Anforderungen an den Class Loader nach einer einzigen Klasse zu verhindern. Als Besonderheit wird in dieser Hashing-Tabelle nicht nach dem tatsächlichen Klassennamen gesucht, sondern nach der Speicheradresse dieser Zeichenfolge. Da die von der Ersetzung betroffenen Namen sowohl unter der ursprünglichen Adresse als auch unter der

Adresse des ersetzten Namens auftreten, könnte die Suche nicht korrekt ausgeführt werden.

Als Gegenmaßnahme werden den gesuchten Namen daher zunächst eindeutige Adressen zugewiesen. Dies verlangsamt die Suche durch zusätzliche Funktionsaufrufe, könnte für das Laufzeitverhalten aber noch optimiert werden. In der Praxis hat sich jedoch gezeigt, daß durch diese Lösung keine bemerkenswerte Verlangsamung bei der Programmausführung festgestellt werden kann.

Die beschriebene Änderung befindet sich an zwei Stellen in *classresolver.c*:

```
#define HASH_INDEX(name, loader)  \
    (((unsigned)TB_hashaddress(name) + (unsigned)loader) %  \
     LOADER_CACHE_TABLE_SIZE)

...

static ClassClass *
LookupLoaderCache(...)
{
    int index = HASH_INDEX(hashed_name, loader);
    loader_cache_t *entry = loader_cache[index];
    hashed_name = TB_hashaddress(hashed_name);
    sysAssert(BINCLASS_LOCKED(sysThreadSelf()));
    while (entry) {
        if (TB_hashaddress(cbName(entry->cb))==hashed_name &&
            entry->loader == loader)
        {
          return entry->cb;
        }
        entry = entry->next;
    }
    return NULL;
}
```

- Nach dem Laden einer Klasse wird von `ClassLoaderFindClass` in *classresolver.c* geprüft, ob der Class Loader tatsächlich eine Klasse mit dem richtigen Namen zurückgeliefert hat. Diese Prüfung muß wegen der Namensersetzung angepaßt werden:

```
/* We do not trust the ClassLoader.loadClass method actually returns
 * the right class.
 */
if (!exceptionOccurred(ee) &&
    strcmp(name, TB_replacename(cbName(result_cb))) != 0)
{
    char buf[MSG_BUF_LEN];
    jio_snprintf( buf, sizeof(buf),
                  "Bad class name (expect: %s, get: %s)",
                  name, TB_replacename(cbName(result_cb)));
    ThrowNoClassDefFoundError(ee, buf);
    result_cb = 0;
}
```

- Bei der Anforderung zum Laden einer Superklasse wird geprüft, ob die Subklasse überhaupt auf diese zugreifen und von ihr erben darf. Diese Prüfung bei `LoadSuperclasses0` in *classresolver.c* ist für Wrapper gelockert, damit auch Final- oder Package-Klassen von schlanken Wrappern überschrieben werden können:

```
/* Don't allow a class to have a superclass it can't access */
if ((!VerifyClassAccess(cb, super, FALSE)) &&
    (!TB_allowoverride(cb,0,super,0)))
```

- Von `PrepareMethods` in *classresolver.c* wird eine Tabelle der verfügbaren Methoden für eine Klasse aufgebaut. Darin werden auch die ererbten oder überschriebenen Methoden der Superklasse eingefügt. Schlanke Wrapper können auch Final-Klassen beerben und Final- oder Package-Methoden überschreiben:

```
static void
PrepareMethods(ClassClass *cb)
{
    ClassClass *TB_super = 0;
...
    if (cbSuperclass(cb) != NULL) {
        ClassClass *super = cbSuperclass(cb);
        TB_super = super;
...
    if ((cbAccess(super) & ACC_FINAL) &&
       (!TB_allowoverride(cb,0,TB_super,0))) {
            char details[MSG_BUF_LEN];
            jio_snprintf(details, sizeof(details),
                     "Class %s is subclass of final class %s",
                     cbName(cb), cbName(super));
...
        if ((smb != NULL) &&
             (smb->fb.name == mb->fb.name) &&
             (smb->fb.signature == mb->fb.signature)) {
            int TB_allowov = TB_allowoverride(cb,mb,TB_super,smb);
...
            /* Package-private methods are not inherited outside of the
               package. */
            if ((smb->fb.access & ACC_PROTECTED) ||
                (smb->fb.access & ACC_PUBLIC) ||
                IsSameClassPackage(smb->fb.clazz, cb) ||
                TB_allowov) {
                if ((smb->fb.access & ACC_FINAL) &&
                   (!TB_allowov)) {
                    char details[MSG_BUF_LEN];
                    jio_snprintf(details,
                        sizeof(details),
                        "Class %s overrides final method %s.%s",
...
```

- Die Methoden `VerifyFieldAccess2` und `VerifyClassAccess` in *classinitialize.c* prüfen, ob auf ein Attribut bzw. eine Methode oder auf eine Klasse zugegriffen werden

darf. Bei einer Wrapping-Beziehung zwischen den beiden betroffenen Klassen wird der Zugriff ohne Einschränkung erlaubt:

```
bool_t
VerifyFieldAccess2(...)
{
    if((current_class == 0 ||
        current_class == field_class ||
        IsPublic(access) ||
        (oldjava && classloader_only && (cbLoader(current_class)==0))) ||
        TB_allowoverride(current_class,0,field_class,0))
            return TRUE;
    }
...

bool_t
VerifyClassAccess(...)
{
    if((current_class == NULL ||
        current_class == new_class ||
        IsPublic(cbAccess(new_class)) ||
        (oldjava && classloader_only && (cbLoader(current_class)==0)) ||
        IsSameClassPackage(current_class, new_class)) ||
        TB_allowoverride(current_class,0,new_class,0)) {
            return TRUE;
    }
...
```

- Von `IsSameClassPackage` in *util.c* wird geprüft, ob sich zwei Klassen im selben Pakkage befinden. Da die gewrappte Klasse unter einem anderen Namen, meist aus einem anderen Package, geführt wird, könnte sie nicht mehr auf Non-public-Attribute und -Methoden in Klassen aus ihrem ursprünglichen Package zugreifen und umgekehrt. Daher muß hier vor dem Vergleich der Zeichenketten des Package-Namens der ursprüngliche Name mit dem ursprünglichen Package benutzt werden:

```
bool_t
IsSameClassPackage(ClassClass *class1, ClassClass *class2)
{
    if (cbLoader(class1) != cbLoader(class2))
        return FALSE;
    else {
        char *name1 = TB_getorigname(cbName(class1));
        char *name2 = TB_getorigname(cbName(class2));
...
```

- Enthalten Klassen Native Code, so muß dieser in einem plattformspezifischen Format zur Verfügung stehen. Die Implementierung einer Native-Code-Methode muß als Funktion mit einem Namen vorliegen, der sich aus dem Namen der Klasse und dem der Methode in einer festgelegten Weise zusammensetzt. Dieser Name wird von `mangleUTFString2` in *util.c* erzeugt und anschließend eine Funktion diesen Namens aufgerufen.

Bei gewrappten Klassen enthalten die Namen der außerhalb der Klassendateien implementierten Native-Funktionen unverändert den ursprünglichen Namen der Klasse. Im Arbeitsspeicher liegt die Klasse jedoch unter dem Namen des vom innersten Wrapper benutzten Klassenrumpfes vor. Beim Zugriff auf den Native Code muß daher die Namensersetzung durchgeführt werden, um wieder den ursprünglichen Klassennamen zu erhalten.

```
int
mangleUTFString2(...)
{
    char *ptr = TB_replacename(name);
    char *bufptr = buffer;
...
}
```

- Beim Aufruf von Private-Methoden wird der spezielle Opcode *invokespecial* benutzt, wie in 4.3.2.2 beschrieben. Um Private-Methoden überschreiben zu können, wird bei dessen Ausführung in den Wrappern der betroffenen Klasse nach einer Methode mit derselben Signatur wie die der eigentlich aufgerufenen Private-Methode gesucht. Statt des ursprünglichen invokespecial wird im Bytecode der aufrufenden Methode nun einer der verschiedenen *invokevirtual*-Opcodes zusammen mit einer Referenz auf die gefundene Methode eingesetzt.

  Somit tritt anschließend der herkömmliche Java-Mechanismus wie bei anderen überschriebenen Methoden in Kraft. Bei der Ausführung des invokevirtual muß allerdings im Fall des Wrapping zusätzlich die Prüfung des Private-Zugriffsrechts umgangen werden.

  Das Laufzeitverhalten wird nur bei der jeweils ersten Ausführung dieser Opcodes an den verschiedenen Stellen im Bytecode verschlechtert, da anschließend die Sun-spezifischen *quick*-Varianten der Opcodes dort eingesetzt werden und somit die hinzugefügten Anweisungen nicht mehr ausgeführt werden. Diese Änderungen befinden sich in *interpreter.c:*

```
} else if (opcode == opc_invokevirtual) {
    unsigned int offset = mb->fb.u.offset;
    if ((mb->fb.access & ACC_PRIVATE) &&
        (!TB_allowoverridelater(mb->fb.clazz,mb))) {
        rewriteByte(quick_buffer, 0, opc_invokenonvirtual_quick);
    } else if (offset <= 0xFF && !UseLosslessQuickOpcodes) {
    ...
    } else {
        rewriteByte(quick_buffer, 0, opc_invokevirtual_quick_w);
    }
} else if (opcode == opc_invokespecial) {
    struct methodblock *TB_mb =
        TB_selectmethodblock_special(ee,ee->current_frame,mb);
    if (mb!=TB_mb) {
        unsigned int offset = TB_mb->fb.u.offset;
        mb = TB_mb;
        if (offset <= 0xFF && !UseLosslessQuickOpcodes) {
            rewriteByte(quick_buffer, 1, mb->fb.u.offset);
            rewriteByte(quick_buffer, 2, mb->args_size);
            rewriteByte(quick_buffer, 0,
```

```
                fieldclass(&mb->fb) == classJavaLangObject
                ? opc_invokevirtualobject_quick
                : opc_invokevirtual_quick);
          } else {
            rewriteByte(quick_buffer, 0, opc_invokevirtual_quick_w);
          }
      } else {
          struct methodblock *new_mb;
          ...
          mb = new_mb;
      }
    };
}
```

- Um statische Klassenmethoden überschreiben zu können, wird bei der Ausführung von *invokestatic* ebenso wie bei invokevirtual nach einer entsprechenden Methode in den Wrappern gesucht. Anschließend kann jedoch nicht wie zuvor ein invokevirtual eingesetzt werden, da bei statischen Methoden keine Objektreferenz auf dem Stack vorliegt, anhand derer bei invokevirtual die Klasse der aufgerufenen Instanz festgestellt werden könnte.

  Die aufzurufende Klasse und Methode sind in einer Tabelle der aufrufenden Klasse, dem sogenannten Constantpool, statisch gelinkt. Um das Aufrufziel dieses invokestatic-Opcodes zu ändern, wird daher in diesem Constantpool die gefundene Methode eingetragen. Der aktuelle und alle folgenden Aufrufe aus dieser Klasse richten sich danach direkt an die gefundene Methode des Wrappers.

  Bei der Suche nach dieser Methode in eventuell mehreren Wrappern einer Klasse ist zu beachten, daß bei einem Aufruf aus einem Wrapper heraus nur die weiter innen liegenden Wrapper durchsucht werden dürfen. Sonst könnten die Wrapper die statischen Methodenaufrufe nicht an die nächstinneren Wrapper bzw. die gewrappte Klasse weiterreichen, sondern es würde immer die Methode des äußersten Wrappers aufgerufen, was zu einer ungewollten Rekursion führen würde.

  Diese Änderung wurde ebenfalls in *interpreter.c* vorgenommen:

```
} else if (opcode == opc_invokestatic) {
    struct methodblock *TB_mb =
        TB_selectmethodblock_static(ee,ee->current_frame,mb);
    if (TB_mb!=mb) {
        mb = TB_mb;
        ee->current_frame->constant_pool[
                                GET_INDEX(&quick_buffer[1])].mb = mb;
    };
    rewriteByte(quick_buffer, 0, opc_invokestatic_quick);
} else if (opcode == opc_invokevirtual) {
...
```

Bereits diese überschaubare Anzahl von Aufrufen der hier nicht abgedruckten Hilfsfunktionen ermöglicht die Integration der vorgestellten Wrapping-Mechanismen in eine Java Virtual Machine. Daher sollte auch eine Migration zu anderen Virtual Machines oder neuen Java-Versionen mit vertretbarem Aufwand möglich sein. Voraussetzung ist allerdings, daß die Quelltexte der Virtual Machine vorliegen.

A.2 Format der JSEC Policy Files

A.2.1 XML Document Type Definition (DTD)

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % classname "CDATA">
<!ENTITY % operator "(access | context | evaluate | op | and | or | not | true |
false)">
<!ENTITY % activator "(apply | activate)">
<!ENTITY % options "(option*)">
<!ELEMENT policy (expression+ | setting+ | rule+)+>
<!ELEMENT expression (%operator;)>
<!ATTLIST expression
      id ID #REQUIRED
>
<!ELEMENT access EMPTY>
<!ATTLIST access
      class %classname; #REQUIRED
      target CDATA #IMPLIED
      action CDATA #IMPLIED
>
<!ELEMENT context %options;>
<!ATTLIST context
      class %classname; #REQUIRED
>
<!ELEMENT evaluate EMPTY>
<!ATTLIST evaluate
      id IDREF #REQUIRED
>
<!ELEMENT op (%operator;)*>
<!ATTLIST op
      class %classname; #REQUIRED
>
<!ELEMENT and (%operator;)+>
<!ELEMENT or (%operator;)+>
<!ELEMENT not (%operator;)>
<!ELEMENT true EMPTY>
<!ELEMENT false EMPTY>
<!ELEMENT setting (%activator;)+>
<!ATTLIST setting
      id ID #REQUIRED
>
<!ELEMENT apply %options;>
<!ATTLIST apply
      class %classname; #REQUIRED
>
<!ELEMENT rule (condition, implication)>
<!ELEMENT activate EMPTY>
<!ATTLIST activate
      id IDREF #REQUIRED
>
<!ELEMENT condition (%operator;)>
<!ELEMENT implication (%activator;)+>
<!ELEMENT option (#PCDATA)>
<!ATTLIST option
      name CDATA #REQUIRED
>
```

A.2.2 Graphische Darstellung der DTD

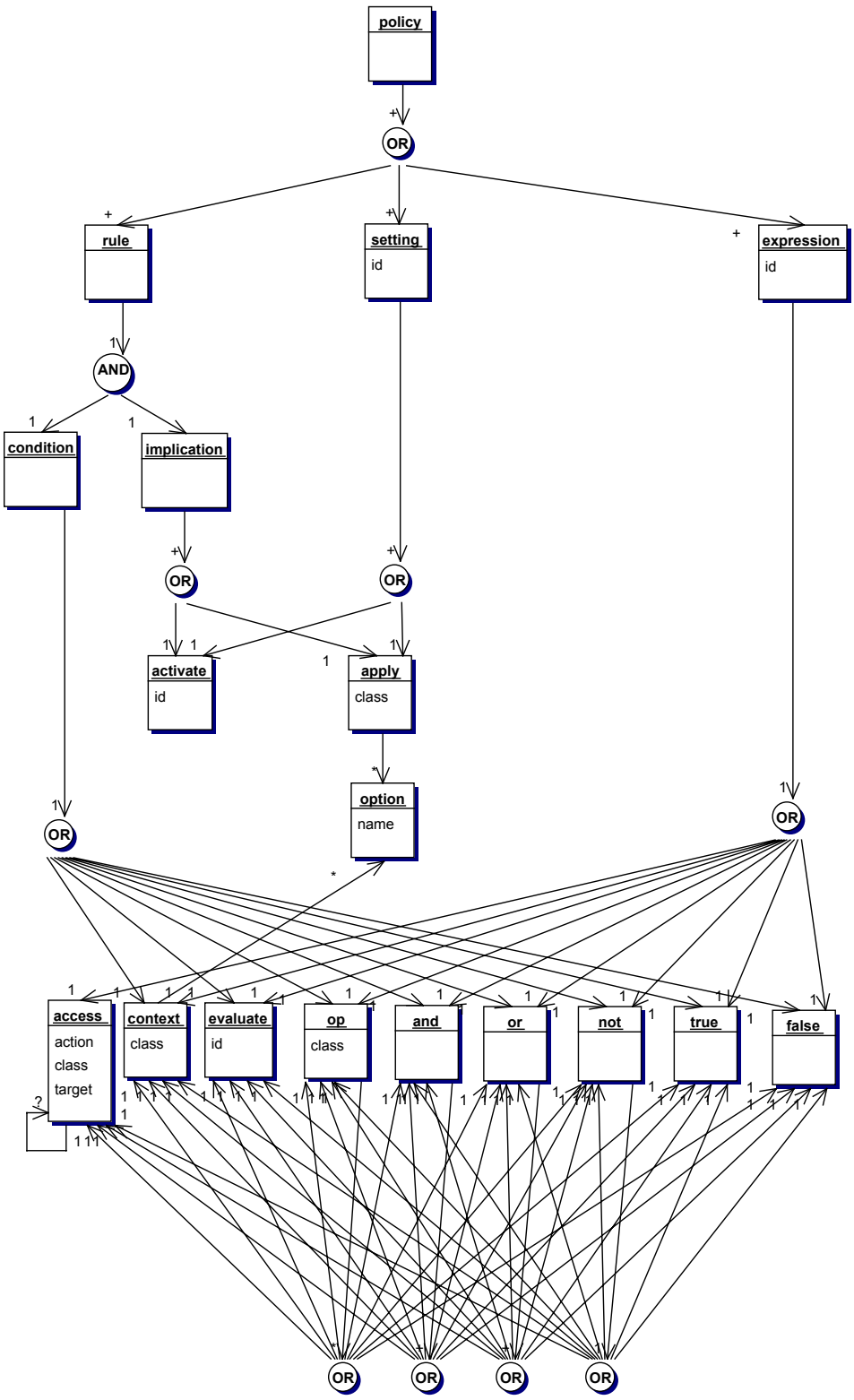

Abbildung A-1: Graphische Darstellung der DTD

A.3 JsecPermission

Die Klasse `jsec.JsecPermission` wird nicht nur von `MainMethodAdapter` aus 5.4.2.1 beim Programmstart als Zugriffsspezifikation mit `JsecPermission "<main class name>", "executeMain"` eingesetzt, sondern auch zum Schutz sicherheitsrelevanter Methoden innerhalb von JSEC verwendet.

Die hier aufgelisteten Permissions müssen an die aufrufenden Klassen vergeben sein, wenn die darunter aufgeführten Methoden aufgerufen werden sollen.

JsecPermission "**getEngines**"
Policy.getEngines

JsecPermission "**getPolicy**"
Policy.getEngines
Policy.getPolicy
Policy.refresh

JsecPermission "**setPolicy**"
Policy.setPolicy

JsecPermission "**refreshPolicy**"
Policy.refresh

JsecPermission "**getNamedExpression**"
PolicyFileXML.getNamedExpression

JsecPermission "**getNamedSetting**"
PolicyFileXML.getNamedSetting

JsecPermission "**setOption.***"
OptionKeeper.setOptions

JsecPermission "**setOption.**<option name>"
OptionKeeper.setOption

JsecPermission "g**etOption.***"
OptionKeeper.getOptions

JsecPermission "**getOption.**<option name>"
OptionKeeper.getOption

JsecPermission "**setExecutionContext**"
OptionKeeper.setExecutionContext

JsecPermission "**getExecutionContext**"
OptionKeeper.getExecutionContext

JsecPermission "**putPrivateCredential**"
GlobalCredentialManager.put (bei priv == true)
SubjectCredentialManager.put (bei priv == true)

JsecPermission "**getPrivateCredential**"
GlobalCredentialManager.get (bei priv == true)
SubjectCredentialManager.get (bei priv == true)

JsecPermission "**getKey**"
KeyStoreManager.getKey

JsecPermission "**getKeyStore**"
KeyStoreManager.getKeyStore

Diese Permissions müssen nicht an Anwendungsteile vergeben werden, die JSEC lediglich über Adapter oder Wrapper aufrufen, da innerhalb von JSEC dazu Privileged Code eingesetzt wurde (siehe auch 5.5).

A.4 Einsatz der JSEC-Standardimplementierung

Die JSEC-Standardimplementierung kann neben den eigentlichen JSEC-Policy-Dateien mit vielfältigen Optionen konfiguriert werden. Bei der Vorstellung von JSEC in Abschnitt 5.4 konnten nicht alle Optionen ausführlich berücksichtigt werden. Daher werden sie in diesem Abschnitt aufgelistet, um eine Nutzung von JSEC zu ermöglichen.

Bei der Ausführung müssen `jsec.jar`, `jsecwrap.jar` und `jsecimpl.jar` im Klassenpfad vorhanden sein, bei Nutzung von Wrapping lediglich `jsecimpl.jar`. Im Verzeichnis `%JAVA_HOME%/lib/ext` werden die Dateien aus den `lib`-Verzeichnissen von JCE, JSSE und JAAS sowie XML-TR2 erwartet.

A.4.1 Optionen in java.security

Die Datei `%JAVA_HOME%/lib/security/java.security` enthält Optionen zur globalen Konfiguration der Java-Sicherheitsbibliotheken. Daher werden auch die folgenden Optionen für JSEC in dieser Datei erwartet und gelten für alle Anwendungen, die mit JSEC ausgeführt werden. Die in der rechten Spalte angegebenen Standardwerte werden benutzt, wenn keine abweichende Option in `java.security` gefunden wird.

jsec.debug=<true/false> false
Mit true erzeugt JSEC Debug-Ausgaben, die bei einer Analyse der Aktivität oder bei Fehlersuche helfen können.

jsec.policy.provider=<classname> edu.udo.jsec.policy.PolicyFileXML
Auswahl einer alternativen Policy-Implementierung.

jsec.policy.url.1(2,3 usw.)**=<url>**
Vorgabe von JSEC-Policy-Dateien für alle Anwendungen, die JSEC nutzen.

policy.allowSystemProperty=<true/false> false
Muß true sein, um weitere Policy-Dateien in der Kommandozeile angeben zu können (siehe A.4.2). Gilt auch für die Access Control Policy.

jsec.combiner.provider.1(2,3 usw.)**=<classname>**
Auswahl von ApplierCombinern (siehe 5.4.1.3)

jsec.preload.1(2,3 usw.)**=<classname>**
Angegebene Klassen werden bei der Initialisierung von PolicyFileXML geladen, um Probleme aufgrund nicht erreichbarer Class Loader zum Zeitpunkt der ersten Nutzung dieser Klassen zu vermeiden. Einige Klassen der JSEC-Standardimplementierung werden auch ohne diese Angabe bereits auf diese Weise geladen.

jsec.secure.socket.1(2,3 usw.)**=<classname>** com.sun.net.ssl.internal.ssl.SSLSocketImpl
com.sun.net.ssl.internal.ssl.SSLServerSocketImpl
Socket-Implementierungen, die als sicher betrachtet werden und nicht zur Ausführung von SokketEngineSPI führen. Dient zur Vermeidung von Endlosschleifen bei der Instanziierung der eingesetzten, sicheren Socket-Klassen.

jsec.context.generate.stack=<true/false> true
Sollen ClassStackAspects vom ContextGenerator (siehe 5.4.4) erzeugt werden?
Falls sie definitiv nicht benötigt werden, bringt die Abschaltung Geschwindigkeitsvorteile.

jsec.context.generate.principals=true/false true
Sollen PrincipalAspects vom ContextGenerator (siehe 5.4.4) erzeugt werden?
Falls sie definitiv nicht benötigt werden, bringt die Abschaltung Geschwindigkeitsvorteile.

jsec.context.debug.stack=<true/false> false
Debug-Ausgaben bei der Erzeugung von ClassStackAspects erzeugen?
(Nur in Verbindung mit jsec.debug=true)

jsec.context.debug.principals=<true/false> false
Debug-Ausgaben bei der Erzeugung von PrincipalAspects erzeugen?
(Nur in Verbindung mit jsec.debug=true)

Neben diesen Optionen können Standardwerte für `<option>`-Tags in `java.security` festgelegt werden. Diese Tags werden in A.4.6 vorgestellt.

A.4.2 Optionen in der Kommandozeile

Um JSEC auch für jede Anwendung einzeln konfigurieren zu können, werden die folgenden System Properties berücksichtigt. Mit `java -Dproperty=value MainClassName` können diese Properties in der Kommandozeile beim Start einer Anwendung angegeben werden.

-Djsec.debug=<true/false> false
Diese Option überschreibt die entsprechende Einstellung aus java.security.

-Djsec.policy=(=)<filename>
Wahl einer JSEC-Policy für diese Anwendung. Bei Verwendung von == wird ausschließlich diese Policy-Datei gelesen und nicht die in java.security oder mit jsec.policy.1(2,3 usw.) angegebenen.

-Djsec.policy.1(2,3 usw.)**=<filename>**
Wahl mehrerer JSEC-Policy-Dateien.

Neben diesen Optionen können Standardwerte für `<option>`-Tags in der Kommandozeile angegeben werden und überschreiben eventuell vorhandene Werte aus `java.security`. Diese Tags werden in A.4.6 vorgestellt.

A.4.3 Adapter-Übersicht (API)

Die folgenden Adapter stehen als API für JSEC im Package `edu.udo.jsec.adapters` bzw. in der Datei `jsecwrap.jar` zur Verfügung und können statt der Klassen aus der rechten Spalte eingesetzt werden. Statt dieser Veränderung der Quelltexte kann auch Wrapping mit der Konfiguration aus A.4.4 verwendet werden.

MainMethodAdapter Klasse mit der main-Methode einer Anwendung

Nutzung entweder in der Kommandozeile mit

```
java edu.udo.jsec.adapters.MainMethodAdapter <class name> <arguments>
```

oder im Quellcode mit

```
new MainMethodAdapter().main(<class>, <method name>, <arguments>);
```

(siehe 5.4.2.1)

SecurityManagerAdapter java.lang.SecurityManager

Nutzung entweder in der Kommandozeile mit

```
java -Djava.security.manager=edu.udo.jsec.adapters.SecurityManagerAdapter ...
```

oder im Quellcode mit

```
System.setSecurityManager(new SecurityManagerAdapter());
```

(siehe 5.4.2.2)

SocketAdapter / ServerSocketAdapter java.net.Socket / java.net.ServerSocket

Statt `s = new Socket(...)` *im Quelltext* `s = SocketAdapter.newSocket(...)` *verwenden. Analoge Ersetzung von ServerSocket (siehe 5.4.2.3).*

UnicastRemoteObjectAdapter UnicastRemoteObject

Objekte, die über RMI verfügbar gemacht werden sollen, erben von UnicastRemoteObject und können zur Nutzung von JSEC statt dessen von UnicastRemoteObjectAdapter erben. Alternativ kann auch weiterhin von UnicastRemoteObject geerbt werden und dessen Konstruktor mit `super(port, new RMIClientSocketFactoryAdapter(), new RMIServerSocketFactoryAdapter());` *aufgerufen werden (siehe 5.4.2.4).*

FileIn(Out)**putStreamAdapter** java.io.FileIn(Out)putStream

Statt `f=new FileInputStream(...)` *im Quelltext* `f=new FileInputStreamAdapter(...)` *verwenden. Analoge Ersetzung von FileOutputStream (siehe 5.4.2.5).*

A.4.4 Wrapper-Konfiguration

Die folgende Wrapping-Konfigurationsdatei kann entweder in `%JAVA_HOME%/lib/wrapping.properties` abgelegt werden und gilt damit für alle Anwendungen, die mit der erweiterten Java Virtual Machine ausgeführt werden, oder in einer Datei, die mit einem Kommandozeilenparameter beim Start einer Anwendung angegeben wird, wie beispielsweise

```
java -classic -wrap:c:\Diplom\java\jsec\jsec.wrap ...
```

Im zweiten Fall muß `wrapping.properties` aber zumindest `wrapmore` enthalten, da ansonsten keine Definition von Wrappern in der Kommandozeile erlaubt ist.

```
wrapclasspath c:\diplom\java\jsec\bin\jsec.jar
wrapclasspath c:\diplom\java\jsec\bin\jsecwrap.jar

wrap java.net.Socket           with edu.udo.jsec.wrappers.SocketWrapper
wrap java.net.ServerSocket     with edu.udo.jsec.wrappers.ServerSocketWrapper

wrap java.io.FileOutputStream  with edu.udo.jsec.wrappers.FileOutputStreamWrapper
wrap java.io.FileInputStream   with edu.udo.jsec.wrappers.FileInputStreamWrapper

# This is not necessary when server and clients both run with wrapping:
# wrap java.rmi.server.UnicastRemoteObject with
#                              edu.udo.jsec.wrappers.UnicastRemoteObjectWrapper

# MainMethodWrapper0.class may NOT be found in wrapclasspath
# wrapmain with edu.udo.jsec.wrappers.MainMethodWrapper0

# This allows a MainMethodWrapper.class in the wrapclasspath
# wrap java.net.URLClassLoader with edu.udo.jsec.wrappers.URLClassLoaderWrapper
# wrapmain with edu.udo.jsec.wrappers.MainMethodWrapper
```

A.4.5 Context-Übersicht

Die folgenden ContextAspects wurden in Abschnitt 5.4.4 vorgestellt und können in einer JSEC-Policy-Datei mit einem `<context>`-Tag in einer Implication benutzt werden:

```
<context class="context class name">
  <option name="option name">option value</option>
  ...
</context>
```

Dies sind die möglichen Klassennamen und die dabei verwendbaren Optionen:

jsec.AllContextAspect
Ein `<context>`*-Tag mit AllContextAspect wird stets zu* `true` *ausgewertet und erwartet keine Optionen.*

edu.udo.jsec.contexts.NeverContextAspect
Ein `<context>`*-Tag mit NeverContextAspect wird stets zu* `false` *ausgewertet und erwartet keine Optionen.*

edu.udo.jsec.contexts.AccessAspect
AccessAspect erzeugt eine Permission aus den Optionen `class`, `target` *und* `action` *und vergleicht den aktuellen Zugriff mit dieser Permission über ihre* `implies`*-Methode. Statt* `AccessAspect` *mit einem* `<context>`*-Tag zu verwenden, kann in einer kürzeren Schreibweise auch das Tag* `<access>` *verwendet werden:*

```
<access class="permission class" target="target" action="action"/>
```

edu.udo.jsec.contexts.PrincipalAspect
PrincipalAspect stellt das Subject fest, das den aktuellen Thread ausführt und vergleicht es mit dem Subject, das über die Optionen `class` *und* `name` *angegeben wurde. Abbildung 5-8 zeigt ein Beispiel für die Verwendung von PrincipalAspect.*

edu.udo.jsec.contexts.ClassStackAspect
ClassStackAspect prüft, ob alle Klassen, die in den Optionen `class.1` *(2,3 usw.) angegeben wurden, auf dem Stack des aktuellen Thread vorliegen. Der Stack eines Threads, von dem der aktuelle Thread erzeugt wurde, wird dabei nicht berücksichtigt. Abbildung 5-8 zeigt ein Beispiel für die Verwendung von ClassStackAspect.*

A.4.6 Engine-Übersicht

Die in 5.4.5 vorgestellten Engines benötigen bei ihrem Einsatz in den `<apply>`-Tags einer JSEC-Policy Optionen zur Festlegung der von ihnen ausgeführten Operationen. Für alle Optionen sind Standardwerte aus der rechten Spalte festgelegt (s. u.), die durch Angabe von `<option>`-Tags für jede einzelne Verwendung einer Engine modifiziert werden können:

```
<apply class="engine class name">
  <option name="option name">option value</option>
  ...
</apply>
```

Der Standardwert für eine Option `option.name` kann zudem überschrieben werden, und zwar zum einen in der Konfigurationsdatei `java.security` mit `jsec.option.name=value` (siehe auch A.4.1) und zum anderen in der Kommandozeile mit `-Djsec.option.name=value` (siehe auch A.4.2).

Setzt eine Engine weitere Engines oder Hilfsklassen aus 5.4.5.9 ein, so müssen auch deren Optionen mit `<option>`-Tags innerhalb des `<apply>`-Tags der Engine spezifiziert werden. Beispielsweise ist in A.5.4.1 für die beiden Anwendungen von `edu.udo.jsec.engines.FileStreamCipherEngine` nicht nur die Option `cipher.factory` angegeben, die direkt von dieser Engine ausgewertet wird, sondern auch Optionen für CipherFactory, KeyStoreManager usw.

Werden von einer Engine weitere Klassen eingesetzt, die eigene Optionen erfordern, so wird in der folgenden Auflistung auf diese Klassen hingewiesen.

- ***LoginEngine***

  login.config
  In welcher Datei ist der Login-Context für JAAS definiert? Wird diese Option nicht angegeben, so nutzt JAAS seine Standardwerte zum Auffinden dieser Datei.

  login.context jseclogin
  Welcher Login-Context für JAAS aus der o. a. Datei soll benutzt werden? In 5.4.5.1 ist ein Beispiel angegeben, das ein Single Login für Windows NT ermöglicht und den Standardwert `jseclogin` *als Bezeichner für den Login-Context nutzt.*

login.privileged false
Soll Subject.doAs als Privileged Code, also mit AccessController.doPrivileged ausgeführt werden? Dies kann nötig sein, wenn die aufrufende Anwendung keine AuthPermission "doAs" besitzt. Damit geht zwar während der Ausführung der AccessControlContext verloren, was aber bei einem Einsatz beim Programmstart keine Auswirkung hat.

login.handler edu.udo.jsec.engines.helpers.TextCallbackHandler
CallbackHandler, der zur Abfrage der Zugangsdaten genutzt wird und an die im LoginContext definierten LoginModule übergeben wird. In der Standardimplementierung stehen TextCallbackHandler, AWTCallbackHandler und SwingCallbackHandler zur Verfügung.

login.handler.retries 3
Anzahl der Nachfragen bei nicht akzeptierten Zugangsdaten.

login.handler.delay 3000
Zwangspause zwischen einzelnen Nachfragen in Millisekunden.

- ***ExitAfterActionEngine***

exit.value 0
Welcher Rückgabewert soll beim Verlassen der JVM geliefert werden?

- ***AddSunProvidersEngine***

Keine Optionen. Die Provider SUN (sun.security.provider.Sun), SunRsaSign (com.sun.rsajca.Provider), SunJCE (com.sun.crypto.provider.SunJCE) und SunJSSE (com.sun.net.ssl.internal.ssl.Provider) werden aktiviert, falls sie noch nicht geladen sind.

- ***SecurityManagerEngine***

security.manager java.lang.SecurityManager
Welcher SecurityManager soll installiert werden?
(Nur möglich, wenn noch kein anderer vorhanden ist.)

security.policy
Die Property java.security.policy wird auf den angegebenen Wert gesetzt, falls er nicht leer ist.

- ***AccessDeniedEngine***

message
Diese Nachricht wird beim Werfen der Exception ihrer Beschreibung hinzugefügt.

- ***AccessControllerEngine***

Keine Optionen. java.security.AccessController wird mit der Prüfung aller Permissions beauftragt, für die diese Engine in der JSEC-Policy konfiguriert ist.

- ***AccessControlLogEngine***

  log.engine edu.udo.jsec.engines.LogEngine
  Welche Engine wird zum Protokollieren der Permissions eingesetzt?
  Die LogEngine kennt weitere Optionen, siehe dort!

  log.security.grant false
  Sollen geprüfte Permissions aufgezeichnet werden, bei denen der AccessController nicht abbricht?

  log.security.deny true
  Sollen nicht erlaubte Zugriffe aufgezeichnet werden, bei denen der AccessController mit einer SecurityException abbricht?

  log.security.deny.throw true
  Sollen SecurityExceptions weitergeworfen werden? Bei false wird die Exception verschluckt und das Programm trotz Zugriffsverweigerung durch den Access Controller fortgesetzt.

  log.security.deny.stack false
  Sollen bei der Aufzeichnung eines nicht erlaubten Zugriffs alle Klassen auf dem Stack ebenfalls protokolliert werden?

- ***LogEngine***

  log.file log.txt
  In welche Log-Datei wird geschrieben?

  log.dateformat yyyy-MM-dd HH:mm:ss zzz
  Die protokollierten Ereignisse werden mit einem Datum versehen. Das Format des Datums kann hier angegeben werden.

- ***SocketLogEngine und FileLogEngine***

  Diese Engines erben von LogEngine und beachten nur die dort angegebenen Optionen.

- ***SSLSocketEngine***

  ssl.starthandshake true
  Soll der SSL-Handshake nach der Erzeugung des Sockets durchgeführt werden?

  ssl.context.factory edu.udo.jsec.engines.helpers.SSLContextFactoryImpl
  Welche SSLContextFactory soll zum Erzeugen des SSLContext verwendet werden, der wiederum die Sockets erzeugt?
  SSLContextFactoryImpl wertet weitere Optionen aus, siehe dort!

- ***SSLServerSocketEngine***

  ssl.needclientauth true
  Müssen Clients ein Zertifikat vorweisen können, das im Truststore des ServerSockets vorhanden ist? Bei false wird jeder Client akzeptiert.

  ssl.context.factory edu.udo.jsec.engines.helpers.SSLContextFactoryImpl
  Welche SSLContextFactory soll zum Erzeugen des SSLContext verwendet werden, der wiederum die ServerSockets erzeugt?
  SSLContextFactoryImpl wertet weitere Optionen aus, siehe dort!

- ***SSLContextFactoryImpl***

  ssl.algorithm TLS
  Welcher Algorithmus soll verwendet werden? Diese Option wird an den Provider weitergereicht. Beim Provider SunJSSE sind SSL und TLS wählbar.

  ssl.provider
  Mit dieser Option kann ein abweichender Provider bestimmt werden, ansonsten wird der Standard-Provider für SSLContexts mit dem angegebenen Algorithmus verwendet.

  ssl.key.manager edu.udo.jsec.engines.helpers.KeyStoreManager
  Welcher KeyManager wird damit beauftragt, den privaten Schlüssel für den Handshake und die Authentifizierung zu liefern?
  KeyStoreManager wertet weitere Optionen aus, siehe dort!

  ssl.trust.manager edu.udo.jsec.engines.helpers.KeyStoreManager
  Welcher KeyManager wird mit dem Laden des Truststore beauftragt, von dessen Zertifikaten der Verbindungspartner mindestens eines vorweisen muß?
  KeyStoreManager wertet weitere Optionen aus, siehe dort!

  credential.manager edu.udo.jsec.engines.helpers.GlobalCredentialManager
  Welchem CredentialManager werden die einmal geladenen oder erzeugten vertraulichen Informationen wie Keystore, Truststore, Schlüssel und SSLContext übergeben?
  Ein CredentialManager wertet weitere Optionen aus, siehe dort!

- ***CipherEngine, StreamCipherEngine, FileStreamCipherEngine***

  cipher.factory edu.udo.jsec.engines.helpers.CommonCipherFactory
  Welche CipherFactory wird zum Erzeugen des notwendigen Ciphers eingesetzt?
  CipherFactory und PBECipherFactory werten weitere Optionen aus, siehe dort!

- ***CipherFactory und CommonCipherFactory***

  cipher.algorithm UNDEFINED
  Welcher Algorithmus soll zur Verschlüsselung eingesetzt werden? Dieser Wert wird an die Klasse Cipher übergeben, aber nur verwendet, wenn cipher.transformation leer ist.

  cipher.transformation
  Die Transformation zur Festlegung der Arbeitsweise des Algorithmus wird statt `cipher.algorithm` *an Cipher übergeben, falls diese Option angegeben ist.*

cipher.provider
Mit dieser Option kann ein abweichender Provider bestimmt werden, ansonsten wird der Standard-Provider für Cipher mit dem angegebenen Algorithmus verwendet.

key.manager edu.udo.jsec.engines.helpers.KeyStoreManager
Welcher KeyManager soll für den Cipher einen benötigten Schlüssel liefern?
(Je nach Algorithmus und Modus den geheimen, privaten oder öffentlichen Schlüssel.)
KeyStoreManager wertet weitere Optionen aus, siehe dort!

- ***PBECipherFactory***

cipher.transformation PBEWithMD5AndDES
Die Transformation zur Festlegung der Arbeitsweise des Algorithmus wird an Cipher übergeben.

cipher.provider
Mit dieser Option kann ein abweichender Provider bestimmt werden, ansonsten wird der Standard-Provider für Cipher mit der angegebenen Transformation verwendet.

cipher.pbe.salt S-A-L-T.
Die Parameter zur Initialisierung des PBE-Algorithmus enthalten ein sogenanntes Salt, das aus 8 Bytes besteht.

cipher.pbe.count 10
Die Parameter zur Initialisierung des PBE-Algorithmus enthalten einen Zähler.

cipher.pbe.password
Das Passwort für die Password Based Encryption (PBE) kann mit dieser Option in der JSEC-Policy angegeben werden. Dies macht nur Sinn, wenn die JSEC-Policy geschützt ist.

cipher.pbe.password.handler.1 edu.udo.jsec.engines.helpers.TextCallbackHandler
Ist kein Passwort angegeben, so wird dieser CallbackHandler aufgerufen, um das Passwort vom Benutzer zu erfragen. Da nicht geprüft werden kann, ob das Passwort korrekt ist, wird diese Abfrage nur einmal durchgeführt und nicht wiederholt. In der Standardimplementierung stehen TextCallbackHandler, AWTCallbackHandler und SwingCallbackHandler zur Verfügung.

- ***KeyStoreManager***

credential.manager edu.udo.jsec.engines.helpers.GlobalCredentialManager
Welchem CredentialManager wird der einmal geladene Keystore und der Schlüssel übergeben und später wieder abverlangt?
Ein CredentialManager wertet weitere Optionen aus, siehe dort!

keystore.provider / truststore.provider
Mit der Option keystore.provider kann ein abweichender Provider bestimmt werden, ansonsten wird der Standard-Provider für Keystores verwendet.

Wird KeyStoreManager zum Laden eines Truststore verwendet, so lauten alle hier aufgeführten Optionen nicht **`keystore.`**`option.name`*, sondern* **`truststore.`**`option.name`*. In manchen Fällen wird von einer Engine sowohl ein Keystore als auch ein Truststore benötigt, so daß getrennte Optionen für zwei KeyStoreManager nötig sind. Beispiel: siehe A.5.5.1*

keystore.type KeyStore.getDefaultType()
Der Typ des Keystore muß angegeben werden, wenn er nicht dem Standard-Typ entspricht.

keystore.url file:{$user.home}/.keystore
Von dieser Adresse wird der Keystore geladen.

keystore.key.alias mykey
Der Schlüssel mit dem angegebenen Namen wird geladen. (Nicht nötig bei Truststores.)

keystore.password
Das Passwort zum Laden des Keystore oder Truststore kann in der JSEC-Policy hinterlegt werden. Dies macht nur Sinn, wenn die JSEC-Policy geschützt ist.

keystore.password.handler.1 (2,3...) edu.udo.jsec.engines.helpers.TextCallbackHandler
Wird das Passwort nicht angegeben, so werden die konfigurierten CallbackHandler aufgerufen, um das Passwort vom Benutzer zu erfragen. In der Standardimplementierung stehen TextCallbackHandler, AWTCallbackHandler und SwingCallbackHandler zur Verfügung.

keystore.password.handler.1.retries 3
Jeder CallbackHandler wird bei falscher Eingabe maximal so oft aufgerufen wie hier angegeben. Danach wird der nächste CallbackHandler benutzt, falls ein weiterer konfiguriert ist. Ansonsten wird eine SecurityException geworfen.

keystore.password.handler.1.delay 3000
Zwangspause in Millisekunden nach jeder fehlerhaften Eingabe.

keystore.key.password
Das Passwort des Schlüssels ist nur für private und geheime Schlüssel (nicht für öffentliche Schlüssel, Zertifikate oder bei Truststores) notwendig und kann mit dieser Option in der JSEC-Policy hinterlegt werden. Dies macht nur Sinn, wenn die JSEC-Policy geschützt ist.

keystore.key.password... ...
Die Optionen `keystore.key.password.handler.1`*(2,3...),* `keystore.key.password.handler.1`*(2,3...)*`.retries` *und* –`delay` *steuern die Abfrage des Kennworts für den Schlüssel und werden wie die o. a. Optionen für das Kennwort des Keystore verarbeitet. (Nicht nötig bei Truststores.)*

- ***CredentialManager***

logout.manager.1 (2,3...) edu.udo.jsec.engines.helpers.TimedLogout
Bei welchen LogoutManagern sollen die zwischengespeicherten Credentials zum späteren Löschen angemeldet werden? TimedLogout, NeverLogout, AWTLogoutButton und SwingLogoutButton können –auch gemeinsam – eingesetzt werden.
Die genannten LogoutManager werten weitere Optionen aus, siehe dort!

- ***TimedLogout***

  logout.delay 60000
  Nach dieser Zeit in Millisekunden wird ein angemeldetes Credential gelöscht.

- ***SwingLogoutButton und AWTLogoutButton***

  Alle Optionen für die LogoutButtons sollten stets nur global in `java.security` *mit* **`jsec`**`.option.name=value` *oder in der Kommandozeile mit* **`-Djsec`**`.option.name=value` *definiert werden, da die graphische Oberfläche nur bei der ersten Verwendung dieser LogoutManager erzeugt wird. Abweichende Optionen in weiteren Engines würden also ohnehin ignoriert.*
  An dieser Stelle sei noch einmal darauf hingewiesen, daß auch die Standardwerte für die Optionen aller anderen Engines auf diese Weise geändert werden können.

  logout.button.label Click to remove credentials
  Dieser Text wird als Beschriftung des Buttons angezeigt.

  logout.window.title JSEC Logout
  Dieser Text wird als Titel des Fensters angezeigt, in dem sich der Button befindet.

  logout.x / logout.y 550 / 80
  Die horizontale / vertikale Position des Fensters auf dem Desktop.

  logout.window.width / height 250 / 75
  Die Breite / Höhe des Fensters.

- ***NeverLogout***

  NeverLogout hat keine Optionen und löscht die Credentials nicht.

A.4.7 Nötige Permissions

JSEC kann als Extension installiert werden [Sun99c], indem die drei JAR-Dateien in das Verzeichnis `%JAVA_HOME%/lib/ext` kopiert werden. Mit der Standard-Policy erhält JSEC dabei eine `java.security.AllPermission` und benötigt keine separate Rechtevergabe.

Befindet sich JSEC jedoch in einem anderen Verzeichnis, so muß entweder eine AllPermission oder die folgenden Permissions vergeben werden:

```
keystore "file:/c:/Diplom/java/jsec/bin/jsec.jks", "JKS";

grant codeBase "file:/c:/diplom/java/jsec/bin/*", signedBy "jsec" {
    permission java.io.FilePermission "<<ALL FILES>>", "read,write";
    permission java.net.SocketPermission "*", "accept,connect,resolve,listen";
```

```
    permission jsec.JsecPermission "*";
    permission java.security.SecurityPermission "*";

    permission javax.security.auth.AuthPermission "*";
    permission javax.security.auth.PrivateCredentialPermission "* * \"*\"", "read";

    permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
    permission java.awt.AWTPermission "accessEventQueue";

    permission java.util.PropertyPermission "*", "read,write";

    permission java.lang.RuntimePermission "accessClassInPackage.sun.misc";
    permission java.lang.RuntimePermission "loadLibrary.nt";
    permission java.lang.RuntimePermission "createSecurityManager";
    permission java.lang.RuntimePermission "setSecurityManager";
};
```

A.4.8 Zertifikat der JSEC-Dateien

Die JSEC-Dateien `jsec.jar`, `jsecimpl.jar` und `jsecwrap.jar` wurden mit einem privaten Schlüssel signiert, dessen zugehöriger öffentlicher Schlüssel in `jsec.jks` vorliegt. Die folgenden Angaben dienen zur Überprüfung des Schlüssels in der elektronisch vorliegenden Datei:

```
Keystore type: jks
Keystore provider: SUN

Alias name: jsec
Creation date: Thu Dec 14 15:53:39 GMT+01:00 2000
Entry type: keyEntry

Certificate chain length: 1
Certificate[1]:
Owner:  CN=Thomas Buehren, OU=Computer Science Software Technology,
        O=University of Dortmund, L=Dortmund, ST=NRW, C=DE
Issuer: CN=Thomas Buehren, OU=Computer Science Software Technology,
        O=University of Dortmund, L=Dortmund, ST=NRW, C=DE
Serial number: 3a38deee
Certificate fingerprints:
         MD5:  E6:07:A7:E1:9D:E7:B6:36:BC:07:6A:68:9B:71:0D:9F
         SHA1: B5:06:DA:FA:1C:96:BE:14:AA:5C:E3:99:82:74:A1:5C:2F:AA:C6:7F
```

A.5 Konfigurationsdateien der Validierungsanwendung

Neben den in A.4.4 und A.4.7 vorgestellten Standardeinstellungen für JSEC wurden die folgenden Konfigurationsdateien für die Validierungsanwendung aus Kapitel 6 erstellt. In Abschnitt 6.4 wurde die Sicherheitspolitik zu der hier aufgelisteten Konfiguration erläutert.

A.5.1 JSEC-Policy zum Feststellen der nötigen Access Control Permissions

Mit dieser JSEC-Policy wird die Anwendung ohne Abbruch durch den AccessController durchgeführt. Alle Permissions, die zu einem Abbruch geführt hätten, werden in einer Log-Datei protokolliert. Anhand dieser Aufzeichnung kann eine Access Control Policy für den tatsächlichen Einsatz der Anwendung definiert werden, wobei die hier dargestellte JSEC-Policy dann nicht mehr verwendet wird.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE policy SYSTEM "file:C:/Diplom/java/jsec/jsecpolicy.dtd">
<policy>

   <rule>
      <condition>
         <access class="jsec.JsecPermission" target="*" action="executeMain"/>
      </condition>
      <implication>
         <apply class="edu.udo.jsec.engines.SecurityManagerEngine">
            <option name="security.manager">
               edu.udo.jsec.adapters.SecurityManagerAdapter
            </option>
         </apply>
      </implication>
   </rule>

   <rule>
      <condition>
         <true/>
      </condition>
      <implication>
         <apply class="edu.udo.jsec.engines.AccessControlLogEngine">
            <option name="log.file">c:/ecco/vaa_pp/security/log/logaccess.txt</option>
            <option name="log.security.grant">false</option>
            <option name="log.security.deny">true</option>
            <option name="log.security.deny.throw">false</option>
            <option name="log.security.deny.stack">true</option>
         </apply>
      </implication>
   </rule>

</policy>
```

A.5.2 Access Control Policy File

Die folgenden Grant-Einträge in `vaa_pp.policy` wurden anhand der mit A.5.1 aufgezeichneten Permissions vorgenommen. An JSEC wurden die Permissions aus A.4.7 vergeben.

```
grant codeBase "file:c:/ecco/vaa_pp/classes.jar" {
   permission java.net.SocketPermission "*", "accept,connect,resolve,listen";

   permission java.io.FilePermission "ParameterSystem.dat", "read,write";
   permission java.io.FilePermission "WorkflowManager.dat", "read,write";
   permission java.io.FilePermission "C:\\ecco\\vaa_pp\\images\\*", "read";
   permission java.io.FilePermission "C:\\ecco\\vaa_pp\\html\\*", "read";

   permission java.util.PropertyPermission "*", "read,write";
   permission java.lang.RuntimePermission "setIO";
   permission java.awt.AWTPermission "accessEventQueue";
};

grant {
   permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
   permission java.io.SerializablePermission "enableSubstitution";

   permission java.net.NetPermission "specifyStreamHandler";

   permission java.lang.RuntimePermission "queuePrintJob";
   permission java.lang.RuntimePermission "readFileDescriptor";
   permission java.lang.RuntimePermission "writeFileDescriptor";
   permission java.lang.RuntimePermission "modifyThread";
   permission java.lang.RuntimePermission "modifyThreadGroup";

   permission java.lang.RuntimePermission "setFactory";
   permission java.lang.RuntimePermission "shutdownHooks";
   permission java.lang.RuntimePermission "setContextClassLoader";
   permission java.lang.RuntimePermission "createClassLoader";
   permission java.lang.RuntimePermission "accessDeclaredMembers";

   permission java.lang.RuntimePermission "loadLibrary.awt";
   permission java.lang.RuntimePermission "loadLibrary.fontmanager";
   permission java.lang.RuntimePermission "loadLibrary.net";
   permission java.lang.RuntimePermission "loadLibrary.zip";
   permission java.lang.RuntimePermission "loadLibrary.JdbcOdbc";
   permission java.lang.RuntimePermission "accessClassInPackage.sun.jdbc.odbc";

   permission java.io.FilePermission "JdbcOdbcSecurityCheck", "write";
   permission java.sql.SQLPermission "setLog";
   permission java.util.PropertyPermission "file.encoding", "read";
   permission java.util.PropertyPermission "browser", "read";
};
```

A.5.3 JSEC-Policy: Basis für Server und Clients

Die folgende XML-Datei enthält gemeinsame Definitionen, die sowohl auf dem Server als auch den Clients eingesetzt werden sollen. Auch der Truststore muß auf Servern und Clients vorhanden sein.

A.5.3.1 vaa_pp_base.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE policy SYSTEM "file:C:/Diplom/java/jsec/jsecpolicy.dtd">
<policy>
   <expression id="INSTALL">
      <false/>
   </expression>
   <expression id="filesecurity">
      <true/>
   </expression>
   <expression id="filelog">
      <true/>
   </expression>
   <expression id="netsecurity">
      <true/>
   </expression>
   <expression id="netlog">
      <true/>
   </expression>

   <expression id="usefilesecurity">
      <and>
         <evaluate id="filesecurity"/>
         <not>
            <and>
               <evaluate id="INSTALL"/>
               <access class="java.io.FilePermission"
                       target="&lt;&lt;ALL FILES&gt;&gt;"
                       action="read"/>
            </and>
         </not>
      </and>
   </expression>

   <expression id="usenetsecurity">
      <and>
         <evaluate id="netsecurity"/>
<!--
   Port 1099 (RMI Registry) kann nur mit Wrapping auf Server + Client
   gesichert werden. Wuerde es Clients ohne Wrapping geben, so duerfte die
   RMI Registry nicht per SSL gesichert werden:
         <not>
            <or>
               <access class="java.net.SocketPermission"
                       target="localhost:1099" action="connect,resolve,listen"/>
               <access class="java.net.SocketPermission"
                       target="127.0.0.1:1099" action="connect,resolve,listen"/>
            </or>
         </not>
-->
```

```
      </and>
   </expression>

   <rule>
      <condition>
         <access class="jsec.JsecPermission" target="*" action="executeMain"/>
      </condition>
      <implication>
         <apply class="edu.udo.jsec.engines.SecurityManagerEngine">
            <option name="security.manager">java.lang.SecurityManager</option>
         </apply>
         <apply class="edu.udo.jsec.engines.AddSunProvidersEngine"/>
      </implication>
   </rule>

   <rule>
      <condition>
         <and>
            <evaluate id="filelog"/>
            <not>
               <or>
                  <access class="java.io.FilePermission"
                          target="c:/ecco/vaa_pp/security/-" action="read,write"/>
                  <access class="java.io.FilePermission"
                          target="c:/ecco/vaa_pp/classes.jar" action="read"/>
                  <access class="java.io.FilePermission"
                          target="c:/Diplom/java/jsec/bin/-" action="read"/>
                  <access class="java.io.FilePermission"
                          target="c:/Programme/JavaSoft/-" action="read"/>
                  <access class="java.lang.RuntimePermission"
                          target="readFileDescriptor" action=""/>
                  <access class="java.lang.RuntimePermission"
                          target="writeFileDescriptor" action=""/>
               </or>
            </not>
         </and>
      </condition>
      <implication>
         <apply class="edu.udo.jsec.engines.FileLogEngine">
            <option name="log.file">c:/ecco/vaa_pp/security/log/logfiles.txt</option>
         </apply>
      </implication>
   </rule>

   <rule>
      <condition>
         <evaluate id="netlog"/>
      </condition>
      <implication>
         <apply class="edu.udo.jsec.engines.SocketLogEngine">
            <option name="log.file">
               c:/ecco/vaa_pp/security/log/logsockets.txt
            </option>
         </apply>
      </implication>
   </rule>
</policy>
```

A.5.3.2 Truststore jsec.trust.jceks

```
Keystore type: JCEKS
Keystore provider: SunJCE
Keystore password: jsec

Alias name: certserver
Entry type: trustedCertEntry

Owner:  CN=I am a Trusted Server and Signing Authority,
        OU=Computer Science Software Technology, O=University of Dortmund,
        L=Dortmund, ST=NRW, C=DE
Issuer: CN=I am a Trusted Server and Signing Authority,
        OU=Computer Science Software Technology, O=University of Dortmund,
        L=Dortmund, ST=NRW, C=DE
```

A.5.4 JSEC-Policy: Server-Seite

Zusätzlich zur Konfiguration aus A.5.3 sind auf dem Server die folgenden Dateien nötig.

A.5.4.1 vaa_pp_server.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE policy SYSTEM "file:C:/Diplom/java/jsec/jsecpolicy.dtd">
<policy>
   <rule>
      <condition>
         <and>
            <evaluate id="usenetsecurity"/>
         </and>
      </condition>
      <implication>
         <apply class="edu.udo.jsec.engines.SSLSocketEngine">
            <option name="ssl.starthandshake">true</option>
            <option name="ssl.algorithm">TLS</option>
            <option name="ssl.context.factory">
               edu.udo.jsec.engines.helpers.SSLContextFactoryImpl
            </option>
            <option name="ssl.keystore.type">jceks</option>
            <option name="ssl.keystore.url">
               file:c:\ecco\vaa_pp\security\jsec.server.jceks
            </option>
            <option name="ssl.keystore.key.alias">keyserver</option>
            <option name="ssl.keystore.password">jsec</option>
            <option name="ssl.keystore.key.password">server</option>
            <option name="ssl.truststore.password">jsec</option>
            <option name="ssl.truststore.type">jceks</option>
            <option name="ssl.truststore.url">
               file:c:\ecco\vaa_pp\security\jsec.trust.jceks
            </option>
            <option name="credential.manager">
               edu.udo.jsec.engines.helpers.GlobalCredentialManager
            </option>
```

```
            <option name="logout.manager.1">
               edu.udo.jsec.engines.helpers.NeverLogout
            </option>
         </apply>
         <apply class="edu.udo.jsec.engines.SSLServerSocketEngine">
            <option name="ssl.needclientauth">true</option>
            <option name="ssl.algorithm">TLS</option>
            <option name="ssl.context.factory">
               edu.udo.jsec.engines.helpers.SSLContextFactoryImpl
            </option>
            <option name="ssl.keystore.type">jceks</option>
            <option name="ssl.keystore.url">
               file:c:\ecco\vaa_pp\security\jsec.server.jceks
            </option>
            <option name="ssl.keystore.key.alias">keyserver</option>
            <option name="ssl.keystore.password">jsec</option>
            <option name="ssl.keystore.key.password">server</option>
            <option name="ssl.truststore.password">jsec</option>
            <option name="ssl.truststore.type">jceks</option>
            <option name="ssl.truststore.url">
               file:c:\ecco\vaa_pp\security\jsec.trust.jceks
            </option>
            <option name="credential.manager">
               edu.udo.jsec.engines.helpers.GlobalCredentialManager
            </option>
            <option name="logout.manager.1">
               edu.udo.jsec.engines.helpers.NeverLogout
            </option>
         </apply>
      </implication>
   </rule>

   <rule>
      <condition>
         <and>
            <evaluate id="usefilesecurity"/>
            <access class="java.io.FilePermission"
                    target="c:/ecco/vaa_pp/ParameterSystem.dat" action="read,write"/>
         </and>
      </condition>
      <implication>
         <apply class="edu.udo.jsec.engines.FileStreamCipherEngine">
            <option name="cipher.algorithm">DES</option>
            <option name="cipher.factory">
               edu.udo.jsec.engines.helpers.CommonCipherFactory
            </option>
            <option name="keystore.type">jceks</option>
            <option name="keystore.url">
               file:c:/ecco/vaa_pp/security/jsec.jceks
            </option>
            <option name="keystore.password">jsec</option>
            <option name="keystore.key.alias">keydes1</option>
            <option name="keystore.key.password">des1</option>
            <option name="credential.manager">
               edu.udo.jsec.engines.helpers.GlobalCredentialManager
            </option>
            <option name="logout.manager.1">
               edu.udo.jsec.engines.helpers.NeverLogout
            </option>
```

```
        </apply>
      </implication>
   </rule>

   <rule>
      <condition>
         <and>
            <evaluate id="usefilesecurity"/>
            <access class="java.io.FilePermission"
                    target="c:/ecco/vaa_pp/WorkflowManager.dat" action="read,write"/>
         </and>
      </condition>
      <implication>
         <apply class="edu.udo.jsec.engines.FileStreamCipherEngine">
            <option name="cipher.algorithm">Blowfish</option>
            <option name="cipher.factory">
               edu.udo.jsec.engines.helpers.CommonCipherFactory
            </option>
            <option name="keystore.type">jceks</option>
            <option name="keystore.url">
               file:c:/ecco/vaa_pp/security/jsec.jceks
            </option>
            <option name="keystore.password">jsec</option>
            <option name="keystore.key.alias">keyblowfish2</option>
            <option name="keystore.key.password">blowfish2</option>
            <option name="credential.manager">
               edu.udo.jsec.engines.helpers.GlobalCredentialManager
            </option>
            <option name="logout.manager.1">
               edu.udo.jsec.engines.helpers.NeverLogout
            </option>
         </apply>
      </implication>
   </rule>
</policy>
```

A.5.4.2 Keystore jsec.server.jceks

Der private Schlüssel des Servers entspricht dem Schlüssel, der zur Zertifizierung der Client-Schlüssel verwendet wird; daher enthält der Schlüssel des Servers lediglich ein Selbstzertifikat und wird damit von den Clients akzeptiert. Der öffentliche Schlüssel des Servers ist im Truststore aus A.5.3.2 auf allen Clients vorhanden.

Zur Zertifizierung könnte jedoch auch ein abweichendes Schlüsselpaar verwendet werden; in diesem Fall müßte auch der Server-Schlüssel ein zweites Zertifikat besitzen.

```
Keystore type: JCEKS
Keystore provider: SunJCE
Keystore password: jsec

Alias name: keyserver
Entry type: keyEntry
Password:   server
Certificate chain length: 1
```

```
Certificate[1]:
Owner:  CN=I am a Trusted Server and Signing Authority,
        OU=Computer Science Software Technology, O=University of Dortmund,
        L=Dortmund, ST=NRW, C=DE
Issuer: CN=I am a Trusted Server and Signing Authority,
        OU=Computer Science Software Technology, O=University of Dortmund,
        L=Dortmund, ST=NRW, C=DE
```

A.5.4.3 Keystore jsec.jceks

```
Keystore type: JCEKS
Keystore provider: SunJCE
Keystore password: jsec

Alias name: keydes1
Entry type: keyEntry
Password:   des1

Alias name: keyblowfish2
Entry type: keyEntry
Password:   blowfish2
```

A.5.4.4 Parameter der Kommandozeile

```
java
   -classic
   -wrap:c:\ecco\vaa_pp\security\vaa_pp.wrap
   -wrap:yes
   -Djsec.debug=false
   -Djsec.policy.1=c:\ecco\vaa_pp\security\vaa_pp_base.xml
   -Djsec.policy.2=c:\ecco\vaa_pp\security\vaa_pp_server.xml
   -Djava.security.manager
   -Djava.security.policy=c:\ecco\vaa_pp\security\vaa_pp.policy
   edu.udo.jsec.adapters.MainMethodAdapter
   de.uni_dortmund.informatik.ecco.vaa_pp.Systemstart.VAAServer
```

A.5.5 JSEC-Policy: Client-Seite

Zusätzlich zur Konfiguration aus A.5.3 sind auf den Clients die folgenden Dateien nötig.

A.5.5.1 vaa_pp_client.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE policy SYSTEM "file:C:/Diplom/java/jsec/jsecpolicy.dtd">
<policy>
<!-- for Windows NT (configured in vaa_pp.login)
   <rule>
      <condition>
          <access class="jsec.JsecPermission" target="*" action="executeMain"/>
      </condition>
      <implication>
```

```
        <apply class="edu.udo.jsec.engines.LoginEngine">
           <option name="login.context">vaappclient</option>
           <option name="login.config">c:\ecco\vaa_pp\security\vaa_pp.login</option>
        </apply>
     </implication>
  </rule>
-->

  <rule>
     <condition>
        <and>
           <evaluate id="usenetsecurity"/>
        </and>
     </condition>
     <implication>
        <apply class="edu.udo.jsec.engines.SSLSocketEngine">
           <option name="ssl.starthandshake">true</option>
           <option name="ssl.algorithm">TLS</option>
           <option name="ssl.context.factory">
              edu.udo.jsec.engines.helpers.SSLContextFactoryImpl
           </option>
           <option name="ssl.keystore.type">jceks</option>
           <option name="ssl.keystore.url">
              file:c:\ecco\vaa_pp\security\jsec.client.jceks
           </option>
           <option name="ssl.keystore.key.alias">keyclient</option>
           <option name="ssl.keystore.key.password.handler.1">
              edu.udo.jsec.engines.helpers.SwingCallbackHandler
           </option>
           <option name="ssl.keystore.password.handler.1">
              edu.udo.jsec.engines.helpers.SwingCallbackHandler
           </option>
           <option name="ssl.truststore.password.handler.1">
              edu.udo.jsec.engines.helpers.SwingCallbackHandler
           </option>
           <option name="ssl.truststore.type">jceks</option>
           <option name="ssl.truststore.url">
              file:c:\ecco\vaa_pp\security\jsec.trust.jceks
           </option>
           <option name="credential.manager">
              edu.udo.jsec.engines.helpers.GlobalCredentialManager
           </option>
           <option name="logout.manager.1">
              edu.udo.jsec.engines.helpers.SwingLogoutButton
           </option>
           <option name="logout.manager.2">
              edu.udo.jsec.engines.helpers.TimedLogout
           </option>
           <option name="logout.delay">240000</option>
        </apply>
        <apply class="edu.udo.jsec.engines.SSLServerSocketEngine">
           <option name="ssl.needclientauth">true</option>
           <option name="ssl.algorithm">TLS</option>
           <option name="ssl.context.factory">
              edu.udo.jsec.engines.helpers.SSLContextFactoryImpl
           </option>
           <option name="ssl.keystore.type">jceks</option>
```

```
            <option name="ssl.keystore.url">
               file:c:\ecco\vaa_pp\security\jsec.client.jceks
            </option>
            <option name="ssl.keystore.key.alias">keyclient</option>
            <option name="ssl.keystore.key.password.handler.1">
               edu.udo.jsec.engines.helpers.SwingCallbackHandler
            </option>
            <option name="ssl.keystore.password.handler.1">
               edu.udo.jsec.engines.helpers.SwingCallbackHandler
            </option>
            <option name="ssl.truststore.type">jceks</option>
            <option name="ssl.truststore.url">
               file:c:\ecco\vaa_pp\security\jsec.trust.jceks
            </option>
            <option name="ssl.truststore.password.handler.1">
               edu.udo.jsec.engines.helpers.SwingCallbackHandler
            </option>
            <option name="credential.manager">
               edu.udo.jsec.engines.helpers.GlobalCredentialManager
            </option>
            <option name="logout.manager.1">
               edu.udo.jsec.engines.helpers.SwingLogoutButton
            </option>
            <option name="logout.manager.2">
               edu.udo.jsec.engines.helpers.TimedLogout
            </option>
            <option name="logout.delay">240000</option>
         </apply>
      </implication>
   </rule>
</policy>
```

A.5.5.2 Keystore jsec.client.jceks

Der private Schlüssel in den Client-Keystores muß vom Server-Schlüssel zertifiziert sein, damit die Clients vom Server akzeptiert werden.

```
Keystore type: JCEKS
Keystore provider: SunJCE
Keystore password: jsec

Alias name: keyclient
Entry type: keyEntry
Password:   client

Certificate chain length: 2

Certificate[1]:
Owner:  CN=I am a Trusted Client,
        OU=Computer Science Software Technology, O=University of Dortmund,
        L=Dortmund, ST=NRW, C=DE
Issuer: CN=I am a Trusted Client,
        OU=Computer Science Software Technology, O=University of Dortmund,
        L=Dortmund, ST=NRW, C=DE
```

```
Certificate[2]:
Owner:  CN=I am a Trusted Client,
        OU=Computer Science Software Technology, O=University of Dortmund,
        L=Dortmund, ST=NRW, C=DE
Issuer: CN=I am a Trusted Server and Signing Authority,
        OU=Computer Science Software Technology, O=University of Dortmund,
        L=Dortmund, ST=NRW, C=DE
```

A.5.5.3 Parameter der Kommandozeile

```
java
   -classic
   -wrap:c:\ecco\vaa_pp\security\vaa_pp.wrap
   -wrap:yes
   -Djsec.debug=false
   -Djsec.policy.1=c:\ecco\vaa_pp\security\vaa_pp_base.xml
   -Djsec.policy.2=c:\ecco\vaa_pp\security\vaa_pp_client.xml
   -Djava.security.manager
   -Djava.security.policy=c:\ecco\vaa_pp\security\vaa_pp.policy
   edu.udo.jsec.adapters.MainMethodAdapter
   de.uni_dortmund.informatik.ecco.vaa_pp.Systemstart.VAAClient
   //ip.of.the.server/
```

Glossar

→*Abstrakte Klasse*	Eine als abstrakt deklarierte Klasse kann nicht als Objekt instanziiert werden, wohl aber deren →Subklassen, falls diese nicht abstrakt sind.
→*Abstrakte Methode*	Eine abstrakte Methode enthält keine Implementierung, sondern nur die Signatur. →Subklassen können die abstrakte Methode überschreiben und eine Implementierung hinzufügen. Klassen mit abstrakten Methoden sind →abstrakte Klassen.
→*Anwendung*	Als Anwendungen werden hier alle Java-Programme, also →Applikationen und →Applets bezeichnet.
→*Applet*	Ein Applet ist ein Java-Programm, das in eine HTML-Seite integriert ist und in der →Java-Laufzeitumgebung des Browsers ausgeführt wird. Meist handelt es sich bei Applets um →Mobile Code. Durch das sogenannte Java Plug-in kann ein Applet jedoch auch in einer separaten Java-Laufzeitumgebung ausgeführt und dennoch innerhalb des Browser-Fensters dargestellt werden.
→*Applikation*	Eine Applikation wird im Gegensatz zu einem →Applet nicht in einem Browser ausgeführt, sondern in einer separaten →Java-Laufzeitumgebung. Durch →Remote Method Invocation und →Class Loader kann jedoch auch in einer Applikation →Mobile Code ausgeführt werden.
→*Bytecode*	Compilierte Form einer Java-Klasse, enthält ähnlich wie ein Maschinenprogramm einzelne Instruktionen (s. 2.2.1.1.1)
→*Class Loader*	Java-Klasse, die andere Klassen lädt (siehe 2.2.1.2)
→*Gewrappte Klasse*	Hier: Klasse, die durch einen →Wrapper ersetzt und eventuell von diesem wiederverwendet wird (siehe 4.2).
→*Java Virtual Machine*	Eine *JVM* ist Teil der →Java-Laufzeitumgebung und führt →Bytecode aus. Mit JVMs für verschiedene Betriebssysteme und Prozessoren können Java-Programme auf diesen verschiedenen Plattformen ausgeführt werden (s. 2.2.1.1.1)

→*Java-Klassenbibliothek*	Auch *Java Class Library (JCL)*. Sammlung von Klassen, die in jeder Java-Laufzeitumgebung enthalten sind und grundlegende Funktionalität bereitstellen, beispielsweise auch Klassen für Ressourcenzugriffe [Sun00f].
→*Java-Laufzeitumgebung*	Auch *Java Runtime Environment (JRE)*. Ausführungsumgebung, in der Java-Programme ausgeführt werden können. Besteht unter anderem aus einer →Java Virtual Machine und der →Java-Klassenbibliothek. Die Java-Laufzeitumgebung enthält im Gegensatz zum *Java Development Kit (JDK)* keinen Compiler, Debugger oder andere Entwicklungswerkzeuge.
→*JDBC*	Programmierschnittstelle für Datenbankzugriffe in Java.
→*Klassenpfad*	Auch *Classpath*. Liste von Verzeichnissen und Archivdateien, in denen der reguläre →Class Loader nach Klassen sucht (siehe 2.2.1.2)
→*Komponente*	Eine Komponente ist ein funktional abgeschlossener, binärer Software-Baustein mit wohldefinierten Schnittstellen und kann in unterschiedlichen, nicht vorhersagbaren Anwendungsumgebungen eingesetzt werden [SGC99]
→*Message Authentication Code*	Siehe 2.1.3.3.1.
→*Mobile Code*	Siehe 2.1.5.
→*Native Code*	Native Code wird nicht von der →Java Virtual Machine ausgeführt, sondern direkt vom Prozessor des Ausführungssystems. Native Code kann in anderen Programmiersprachen entwickelt und für eine bestimmte Plattform compiliert werden. Auf dieser Plattform kann Native Code aus Java-Programmen aufgerufen werden.
→*Polymorphie*	Siehe →Superklasse.
→*Reflection API*	Das Reflection Application Programming Interface ermöglicht zur Laufzeit einen Zugriff auf den Aufbau auch unbekannter Klassen sowie auf die Methoden und Attribute einer Klasse und ihrer Objektinstanzen.
→*Remote Method Invocation*	RMI ermöglicht Methodenaufrufe bei Objekten auf entfernten Rechnern und das Versenden von Objekten als Parameter oder Rückgabewert dieser Methoden. Die Objekte auf dem Server melden sich bei der RMI-Registry an und können so von Clients aufgefunden werden. Der Programmierer muß durch automatische Erzeugung von sogenannten Stubs

	und Skeletons keine Socket-Befehle auf Server- und Client-Seite implementieren.
→*Stack*	Die bekannte Datenstruktur verwaltet als Programm-Stack Übergabeparameter, Rückgabewerte und Rücksprungadressen von Methoden, die eine andere Methode aufgerufen haben und auf deren Rückkehr warten.
→*Statische Methode*	Eine statische Methode ist in einer Klasse definiert und unabhängig von den Objekten dieser Klasse. Sie kann auch ohne ein Objekt der Klasse aufgerufen werden. Statische Methoden können nur auf →statische Attribute und andere statische Methoden zugreifen, da ihnen bei der Ausführung keine Objektinstanz der Klasse bekannt ist.
→*Statisches Attribut*	Ein statisches Attribut ist eine Eigenschaft der Klasse, nicht die eines Objekts dieser Klasse. Das statische Attribut existiert, sobald die Klasse geladen ist, auch wenn keine Objekte dieser Klasse instanziiert wurden.
→*Subklasse*	Erbende Klasse in einer Vererbungsbeziehung. Die Subklasse erbt alle Methoden und Attribute ihrer einzigen →Superklasse. In Java erbt jede Klasse zumindest von der Klasse `Object`.
→*Superklasse*	Klasse, von der andere Klassen erben. Die →Subklassen können Methoden der Superklasse überschreiben und so die Funktionalität verändern. Auch Objekte der Subklassen entstammen der Superklasse und können daher durch *Polymorphie* wie Objekte der Superklasse eingesetzt werden.
→*Thread*	Mit Threads kann innerhalb des Betriebssystemprozesses einer →Anwendung Nebenläufigkeit erzeugt werden. Threads werden daher auch als *Lightweight Processes* bezeichnet.
→*Wrapper*	Hier: Java-Klasse, die zur Laufzeit eine andere Klasse ersetzt und die Funktionalität ändern kann. Diese andere Klasse wird als →gewrappte Klasse bezeichnet und kann für die Grundfunktionalität wiederverwendet werden (siehe 4.2).

Literaturverzeichnis

[BCC99] T. Bühren, M. Cakir, E. Can, A. Dombrowski, G. Geist, V. Gruhn, M. Gürgen, S. Handschumacher, M. Heller, C. Lüer, D. Peters, G. Vollmer, U. Wellen, J. v. Werne: *Endbericht der Projektgruppe eCCo (PG315).* Internes Memorandum Nr. 99 des Lehrstuhls Software-Technologie, Universität Dortmund, ISSN 0933-7725, Februar 1999

[BDS00] Dirk Balfanz, Drew Dean, Mike Spreitzer: *A Security Infrastructure for Distributed Java Applications.* Proceedings of 2000 IEEE Symposium on Security and Privacy, Mai 2000

[BG97] Dirk Balfanz, Li Gong: *Experience with Secure Multi-Processing in Java.* Proceedings of the 18th International Conference on Distributed Computing Systems, Seiten 398-405, Amsterdam, Mai 1998

[BPS98] T. Bray, J. Paoli, C. M. Sperberg-McQueen: *Extensible Markup Language (XML) 1.0.* W3C Recommendation REC-xml-19980210, 1998

[Cal98] Peter H. Callaway: *SecureWay Technologies*. http://www.ibm.com/security/technologies/index.html, Mai 1998

[Cha81] David Chaum: *Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms*. Communications of the ACM 24/2, 1981

[DG98] David Griswold: *The Java HotSpot Performance Engine Architecture*. http://www.javasoft.com/products/hotspot/whitepaper.html, April 1999

[DVR00] Loris Degioanni, Piero Viano, Fulvio Risso: *WinDump 2.02.* Politecnico di Torino, http://netgroup-serv.polito.it/windump/, März 2000

[FBF99] T. Fraser, L. Badger, M. Feldman: *Hardening COTS Software with Generic Software Wrappers.* Proceedings of the 1999 IEEE Symposium on Security and Privacy, ISBN 0-7695-0176-1, Seiten 2-16, 1999

[FS97] M. Fowler, K. Scott: *UML Distilled. Applying the Standard Object Modeling Language.* Addison-Wesley, 1997

[GHJ95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns.* Addison-Wesley, 1995

[GJM91] C. Ghezzi, M. Jazayeri, D. Mandrioli: *Fundamentals of Software Engineering*. Prentice-Hall, 1991

[GJS00] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: *The Java Language Specification, Second Edition*. Addison-Wesley, ISBN 0201310082, 2000

[Gol98] Allen Goldberg: *A Specification of Java Loading and Bytecode Verification.* 5th ACM Conference on Computer and Communications Security, Association for Computing Machinery, ISBN 1-58113-007-4, Seiten 49-58, November 1998

[Gon98a] Li Gong: *Java 2 Platform Security Architecture*. Sun Microsystems, Inc., http://java.sun.com/j2se/1.3/docs/guide/security/spec/security-spec.doc.html, 02.10.1998

[Gon98b] Li Gong: *Secure Java Class Loading.* IEEE Internet Computing, Seiten 56-61, November, Dezember 1998

[Gon99] Li Gong: *Inside Java 2 Platform Security – Architecture, API Design and Implementation.* Addison Wesley Longman, Inc., ISBN 0-201-31000-7, 1999

[GS98] L. Gong, R. Schemers: *Signing, Sealing, and Guarding Java Objects.* Lecture Notes in Computer Science (LNCS), Vol. 1419, Springer-Verlag, Juni 1998

[GW90] Li Gong, David J. Wheeler: *A Matrix Key Distribution Scheme.* Journal of Cryptology 2, Seiten 51-59, Springer-Verlag, 1990

[GWT96] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer: *A Secure Environment for Untrusted Helper Applications.* Proceedings of the 6th USENIX Security Symposium, Seiten 1-13, 1996

[Hag98] Masami Hagiya: *On a New Method for Dataflow Analysis of Java Virtual Machine Subroutines.* Proc. 1998 Static Analysis Symposium, Springer-Verlag LNCS, 1998

[HMM00] A. Herzberg, Y. Mass, J. Mihaeli, D. Naor, Y. Ravid: *Access Control meets Public Key Infrastructure or: Assigning Roles to Strangers.* Paper was accepted for the 2000 IEEE Symposium on Security and Privacy, 2000

[IBM99] IBM Corporation (Hg.): *Virtual Private Networking ITSO Workshop.* http://iws.as400.ibm.com/tcpip/vpn/itsodocs/itsovpnwork/html/workshop1.htm, 1999

[JH98] Christian Jensen, Daniel Hagimont: *Protection Wrappers – A Simple and Portable Sandbox for Untrusted Applications*. 8th ACM European Special Interest Group on Operating Systems (SIGOPS) Workshop: Support for Composing Distributed Applications, Sintra, Portugal, September 1998

[JLM89] V. Jacobson, C. Leres, S. McCanne: *TCPdump.* Lawrence Berkeley Laboratory, Berkeley, CA,. http://www-nrg.ee.lbl.gov/, Juni 1989

[Ker91] Heinrich Kersten: *Einführung in die Computersicherheit.* R. Oldenbourg Verlag, München, ISBN 3-486-21873-5, 1991

[KNN98] L. Koved, A. J. Nadalin, D. Neal, T. Lawson: *The evolution of Java security.* http://www-4.ibm.com/software/developer/library/javaevol/javaevol.html, 20.03.1998

[LB98] Sheng Liang, Gilad Bracha: *Dynamic Class Loading in the Java Virtual Machine*. Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications 1998 (OOPSLA '98), veröffentlicht als ACM SIGPLAN Notices, Volume 33, Number 10, Seiten 36-44, Oktober 1998

[LGK99] C. Lai, L. Gong, L. Koved, A. Nadalin, R. Schemers: *User Authentication and Authorization in the Java Platform.* Proceedings of the 15th IEEE Annual Computer Security Applications Conference, Dezember 1999

[Lip90] W.-M. Lippe: *Computer-Sicherheit: Übersicht über Schwachstellen im Computer-System, insbesondere in Rechnernetzen.* ONLINE '90, 13. Europäische Congressmesse für Technische Kommunikation, Kolloqium A – Computer-Sicherheit: Aktuelle Gefahren und Schutzmöglichkeiten, Februar 1990

[LY99] Tim Lindholm, Frank Yellin: *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, ISBN 0201432943, 1999

[MF99] Gary McGraw, Edward W. Felten: *Securing Java: Getting Down to Business with Mobile Code, Second Edition*. John Wiley & Sons, Inc., 1999

[MRW99] F. Monrose, M. K. Reiter, S. Wetzel: *Password Hardening Based on Keystroke Dynamics.* 6th ACM Conference on Computer and Communications Security, Association for Computing Machinery, ISBN 1-58113-148-8, Seiten 73-82, November 1999

[MS01] Vlada Matena, Beth Stearns: *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform*, ISBN 0-201-70267-3, Addison-Wesley, 2001

[NCS93] National Computer Security Center (Hg.): *A guide to understanding covert channel analysis of trusted systems*. Technical Report NCSC-TG-030 Version-1 (Light Pink Book), National Security Agency, Fort George G. Meade, Maryland, 1993

[Pfl97] Charles P. Pfleeger: *Security in Computing, Second Edition*. Prentice Hall International Editions, 1997

[PRG99] Marco Pistoia, Duane F. Reller, Deepak Gupta, Milind Nagnur, Ashok K. Ramani: *Java 2 Network Security, Second Edition*. Prentice Hall, ISBN 0-13-015592-6, 1999

[PSW98] A. Pfitzmann, A. Schill, A. Westfeld, G. Wicke, G. Wolf, J. Zöllner (Dresden University of Technology): *A Java-based distributed platform for multilateral security*. IFIP Working Conference on Trends in Distributed Systems / Electronic Commerce, Juni 1998

[RC97] Phillip Rogaway, Don Coppersmith: *A Software-Optimized Encryption Algorithm*. 05.09.1997. Vorherige Version erschienen in Cambridge Security Workshop Proceedings, Seiten 56-63, Springer-Verlag, 1994

[RJB98] James Rumbaugh, Ivar Jacobson, Grady Booch: *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998

[Sch96] Bruce Schneier: *Applied Cryptography, Second Edition*. John Wiley & Sons, Inc., 1996

[Sch98] Fred B. Schneider: *Enforceable Security Policies.* Cornell University Technical Report TR 98-1664, Januar 1998

[SGC99] Michael Stal, Christian Gengenbach, Krzysztof Czarnecki: *Des Knaben Wunderhorn.* In: OBJEKTspektrum, 1/99, Seite 18, 1999

[SP98] Ravi Sandhu, Joon S. Park: *Decentralized User-Role Assignment for Web-based Intranets.* Proceedings of the Third ACM Workshop on Role-Based Access Control, Association for Computing Machinery, New York, ISBN 1-58113-113-5, Seiten 1-12, Oktober 1998

[Sta95] William Stallings: *Network and internetwork security: principles and practice.* Prentice-Hall, Inc., ISBN 0-02-415483-0, 1995

[Sun97] Sun Microsystems, Inc. (Hg.): *Java Native Interface Specification*. http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/jniTOC.doc.html, 16.05.1997

[Sun98a] Sun Microsystems, Inc. (Hg.): *Security Managers and the Java 2 SDK*. http://java.sun.com/j2se/1.3/docs/guide/security/smPortGuide.html, 19.10.1998

[Sun98b] Sun Microsystems, Inc. (Hg.): *X.509 Certificates and Certificate Revocation Lists (CRLs)*. http://java.sun.com/j2se/1.3/docs/guide/security/cert3.html, 20.05.1998

[Sun98c] Sun Microsystems, Inc. (Hg.): *API for Privileged Blocks*. http://java.sun.com/j2se/1.3/docs/guide/security/doprivileged.html, 22.09.1998

[Sun98d] Sun Microsystems, Inc. (Hg.): *Permissions in the Java 2 SDK*. http://java.sun.com/j2se/1.3/docs/guide/security/permissions.html, 1998

[Sun98e] Sun Microsystems, Inc. (Hg.): *Default Policy Implementation and Policy File Syntax*. http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html, 30.10.1998

[Sun98f] Sun Microsystems, Inc. (Hg.): *Summary of JDK 1.2 Security Tools.* http://java.sun.com/j2se/1.3/docs/guide/security/SecurityToolsSummary.html, 29.10.1998

[Sun99a] Sun Microsystems, Inc. (Hg.): *Java Cryptography Architecture API Specification & Reference*. http://java.sun.com/j2se/1.3/docs/guide/security/CryptoSpec.html, 06.12.1999

[Sun99b] Sun Microsystems, Inc. (Hg.): *How to Implement a Provider for the Java Cryptography Architecture*. http://java.sun.com/j2se/1.3/docs/guide/security/HowToImplAProvider.html, 08.07.1999

[Sun99c] Sun Microsystems, Inc. (Hg.): *Extension Mechanism Architecture*. http://java.sun.com/j2se/1.3/docs/guide/extensions/spec.html, 1999

[Sun00a] Sun Microsystems, Inc. (Hg.): *Java Authentication and Authorization Service (JAAS) 1.0 Developer's Guide*. http://java.sun.com/security/jaas/doc/api.html, 14.01.2000

[Sun00b] Sun Microsystems, Inc. (Hg.): *Security Code Guidelines*. http://java.sun.com/security/seccodeguide.html, 02.02.2000

[Sun00c] Sun Microsystems, Inc. (Hg.): *Java Secure Socket Extension (JSSE)*. http://java.sun.com/products/jsse/doc/apidoc/index.html, 05.04.2000

[Sun00d] Sun Microsystems, Inc. (Hg.): *Java Cryptography Extension (JCE)*. http://java.sun.com/security/JCE1.2/spec/apidoc/index.html, 26.04.2000

[Sun00e] Sun Microsystems, Inc. (Hg.): *Java API for XML Processing (JAXP)*. http://java.sun.com/xml, 17.11.2000

[Sun00f] Sun Microsystems, Inc. (Hg.): *Java 2 Platform, Standard Edition, v 1.3, API Specification.* http://java.sun.com/j2se/1.3/docs/api/index.html, 2000

[Ven99] Bill Venners: *Inside the Java 2 Virtual Machine, Second Edition*. McGraw-Hill Professional Publishing, 1999

[Wal99] Dan Seth Wallach: *A New Approach to Mobile Code Security.* PhD Thesis, Princeton University, Januar 1999

[WBD97] Dan S. Wallach, Dirk Balfanz, Drew Dean, Edward W. Felten: *Extensible Security Architectures for Java.* Proceedings of the 16th ACM Symposium on Operating System Principles, Saint-Malo, France, Seiten 116–128, Oktober 1997

[Wec90] G. Weck: *Informationszugriff durch nicht autorisierte Personen? Zugangs- und Zugriffskontrolle als zentrale Sicherungsmaßnahmen.* ONLINE '90, 13. Europäische Congressmesse für Technische Kommunikation, Kolloqium A – Computer-Sicherheit: Aktuelle Gefahren und Schutzmöglichkeiten, Februar 1990

[WF98] Dan S. Wallach, Edward W. Felten: *Understanding Stack Inspection*. Proceedings of 1998 IEEE Symposium on Security and Privacy, Oakland, California, Mai 1998

Hinweis zu WWW-Adressen

Einige der Literaturstellen standen ausschließlich im World Wide Web zur Verfügung. Adressen im Internet besitzen jedoch leider keine dauerhafte Gültigkeit. Die betroffenen Dokumente wurden daher in elektronischer Form gesichert und können vom Autor zur Verfügung gestellt werden. Die Datumsangaben bezeichnen die Zeitpunkte der Veröffentlichung.