Diplomarbeit

Sicherheitsaspekte von Java 2 Enterprise Edition (J2EE)

von

André Luerssen

betreut durch

Prof. Dr. K. Brunnstein

und

Dr. H.-J. Mück

Januar 2003

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Anwendungen der Informatik
in Geistes- und Naturwissenschaften

Eidesstattliche Erklärung

Hiermit erkläre ich meine Diplomarbeit selbständig durchgeführt zu haben. Für diese Arbeit habe ich keine anderen, als die im Anhang angegebenen, Hilfsmittel und Quellen benutzt.

08. Januar 2003 André Luerssen

Danksagung

Ich danke Herrn Prof. Dr. K. Brunnstein und Herrn Dr. H.-J. Mück für die Betreuung der Diplomarbeit.

Ein besonderer Dank geht an Mr. Rob High, Mrs. Katie Barrett und Mr. Peter Birk von IBM in Austin, Texas. Ihre wertvollen Informationen zur Sicherheit in WebSphere haben wesentlich zur Qualität der vorliegenden Arbeit beigetragen.

Zusammenfassung

In Unternehmen werden verstärkt Applikationsserver eingesetzt, die auf Enterprise Java von Sun Microsystems basieren. Ein weit verbreitetes Produkt ist der IBM Websphere Application Server Advanced Edition v4.0. Diese Diplomarbeit beschäftigt sich mit der Sicherheit von Enterprise Java (J2EE) in Websphere. Zum Verständnis der J2EE-Security wird das Sicherheitsmodell von Java 2 (J2SE) und die J2EE-Architektur mit ihren Komponenten konzeptionell dargestellt. Im zweiten Teil dieser Arbeit wird die Architektur und das Sicherheitsmodell von WebSphere beschrieben. Abschließend werden in einer Differentialanalyse die Unterschiede, zwischen den Spezifikationen von SUN und der Implementation von IBM, aufgezeigt.

Inhaltsverzeichnis:

1 Einleitung

Diese Arbeit beschäftigt sich mit dem komplexen Gebiet der Sicherheit in Enterprise Applikationen, welche auf der Java 2 Plattform Enterprise Edition Spezifiktion 1.3 [1] von SUN Microsystems [J2EES13] beruhen.

Die Sicherheit von Java in der Standardversion (J2SE[2]) [J2SES13] wurde bereits in meiner Studienarbeit „Netzwerkprogrammierung mit Java2: Risiken und Lösungsansätze“ im Kapitel „J2SE-Security“ dargestellt. Dieses Kapitel wurde komplett überarbeitet und zur Vollständigkeit in diese Arbeit übernommen.

Ziel dieser Diplomarbeit ist die Darstellung der verschiedenen Technologien sowie ihr komplexes Zusammenwirken. Die daraus gewonnenen Erkenntnisse sollen dem Verständnis des J2EE-Sicherheitsmodells dienen, welches eine Erweiterung des J2SE-Sicherheitsmodells darstellt.

Auf Technologien wie Webservices, XML, CORBA, LDAP, SSL und Kerberos wird in Hinsicht auf den Umfang dieser Arbeit nicht im Detail eingegangen.

Als Beispiel einer Implementation wird der WebSphere Application Server von IBM mit seiner Architektur, seinen Funktionalitäten und seinem Sicherheitsmodell dargestellt.

Eine anschließende Differentialanalyse soll Unterschiede zwischen den Spezifikationen von SUN und der Implementation von IBM aufzeigen.

Das größte Problem, bei der Erstellung dieser Arbeit, war die große Anzahl unterschiedlicher Technologien und der Identifizierung ihrer sicherheitsrelevanten Elemente.

Die Implementation konnte nur konzeptionell überprüft werden, da Implementationsdetails (interne Spezifikation von WebSphere, Dokumentation der Implementation und Source Code) von IBM nicht zur Verfügung gestellt werden.

Diese Arbeit kann als Grundlage für weitere Untersuchungen dienen, die sich mit den einzelnen Komponenten von WebSphere oder anderer, auf J2EE-basierende, Applikationsserver beschäftigen.

[1] Java 2 Plattform Enterprise Edition Spezifiktion 1.3 im folgenden kurz J2EE 1.3
[2] Java 2 Platform Standard Edition (J2SE)

2 Die J2EE-Plattform

Heutzutage findet man in Unternehmen oft heterogene Netzwerke mit Servern verschiedener Hersteller mit unterschiedlichen Betriebssystemen, auf denen Applikationen laufen, die in verschiedenen Programmiersprachen entwickelt wurden. Ziele heutiger Unternehmen sind u.a. die Reichweite durch e-Commerce zu erweitern, dabei Kosten zu senken und ihre Antwortzeiten zu Kunden, Angestellten und Zulieferern zu verringern. Es wird also nach Lösungen gesucht, bestehende Altsysteme miteinander zu kombinieren und mit neuen Applikationen zu ergänzen. Diese sollten möglichst „webfähig“ sein und Schnittstellen für zukünftige Clients wie z.B. Handys ermöglichen, um eine Zukunftssicherheit zu gewährleisten. Üblicherweise kombinieren diese neuen Applikationen existierende „Enterprise Information Systems“ (EIS) mit neuen Geschäftsfunktionen, die Dienste einer vielfältigen Anzahl an Benutzern zur Verfügung stellen und folgende Merkmale erfüllen:

- *Hoch verfügbar*, um die Bedürfnisse des heutigen globalen Geschäftsumfeldes zu erfüllen. Dies setzt eine Skalierbarkeit voraus.

- *Sicher*, durch den Einsatz kryptographischer Mittel soll der Schutz der Privatsphäre der Benutzer und der Integrität der Unternehmung gewährleistet werden.

- *Zuverlässig und Skalierbar*, um die schnelle und exakte Durchführung von Geschäftstransaktionen zu gewährleisten. Persistenzmechanismen und Transaktionsmonitore bilden hierfür die Grundlage.

Diese Dienste werden in den meisten Fällen als n-tier (mehrschichtige) Applikationen implementiert. Die mittlere Schicht integriert die Geschäftsfunktionen und Daten der neuen Dienste in bestehende EISs. Man spricht von *Middleware* oder auch von *Integrationslösungen*. Fortgeschrittene Web-Technologien versorgen Benutzer der ersten Schicht mit einem einfachen Zugriff auf die komplexen Geschäftsfunktionen und eliminieren bzw. verringern den Aufwand an Administration und Training der Benutzer. In der folgenden Abbildung 1 ist diese Architektur dargestellt.

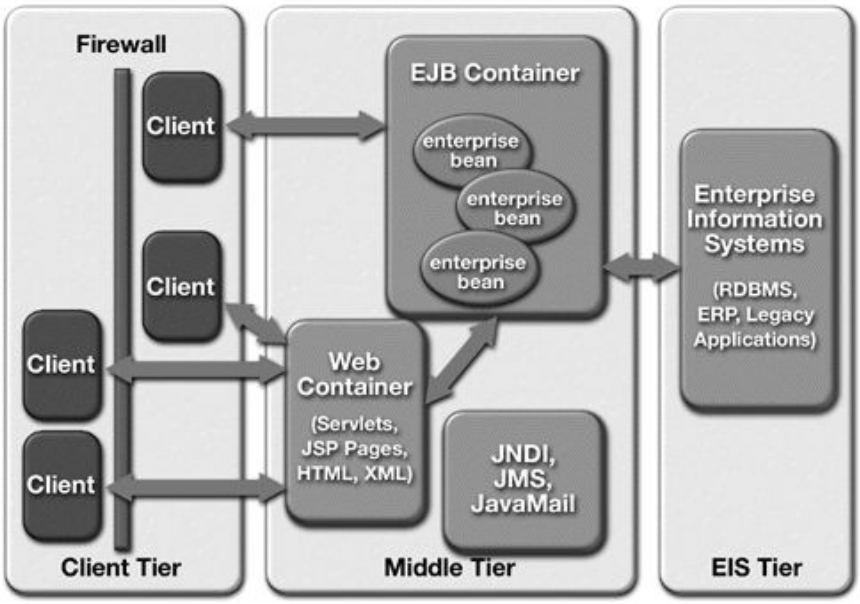

Abbildung 1: J2EE-Umgebung aus [DEAJ2EE]

Die nachträgliche Öffnung geschlossener Systemen zum Internet und anderen Netzen (z.B. Mobilfunk) bringt Gefahren mit sich:

- Die Systeme werden möglicherweise über Kanäle angreifbar, die in ihrer ursprünglichen Form nicht existierten.

- Das Verhalten des Gesamtsystems ist eventuell nicht mehr berechenbar.

- Eine Schwäche in einen System kann sich auf andere Systeme nachteilig auswirken (Beispiel: Schwächstes Glied einer Kette).

- Webservices, bei denen Methoden auf einem Server über das SOAP[1] Protokoll gerufen werden, stellen, nach Meinung des Unternehmens Zapthink [XMLPROXY], ein weiteres Sicherheitsproblem dar. Hierbei findet ein Informationsaustausch mittels XML[2] über HTTP bzw. HTTPS statt, der für herkömmliche Firewalls problematisch ist, da diese i.d.R. nur nach IP-Adressen und Ports filtern. Eine Lösung bieten s.g. XMLProxies, welche bereits als Softwareversionen existieren. Diese sind für größere Datenmengen jedoch nicht geeignet und Hardwareversionen existieren noch nicht.

Es stellt sich die Frage, ob die o.g. Ziele bezüglich Vergrößerung der Reichweite der Unternehmen, Verringerung der Kosten etc. diese Risiken rechtfertigen. Ich vermute, die Entscheider dieser Unternehmen sind sich der Risiken nicht bewusst.

Die Architektur der Java 2 Plattform, Enterprise Edition (J2EE) konzentriert sich auf die Reduktion der Kosten und der Komplexität bei der Entwicklung von mehrschichtigen Enterprise Diensten (Applikationen). So wirbt SUN damit, dass J2EE-Applikationen schnell eingesetzt (deployed) und einfach erweitert werden können, wenn die Unternehmung auf möglichen Wettbewerbsdruck reagieren will.

Die Kostensenkung und Geschwindigkeit kann ich mit den Erfahrungen aus meiner beruflichen Praxis bestätigen, doch Langzeitstudien existieren noch nicht, da diese Technologien zu neu sind. Laut SUN erreicht die J2EE-Architektur diese Vorteile, indem sie eine Standardarchitektur mit folgenden Elementen bietet:

- *J2EE Platform*, eine Standard-Plattform zum Hosting von J2EE-Applikationen.

- *J2EE Compability Test Suite*, eine Suite mit Kompatibilitätstests zur Verifizierung von J2EE-Plattform Produkten, gemäß der *J2EE Platform Specification*.

- *J2EE Reference Implementation*, eine Referenzimplementation zur Herstellung von Prototypen von J2EE-Applikationen und zur operativen Definition der J2EE-Plattform.

- *J2EE BluePrints*, eine Sammlung von so genannten „best-practices“ für die Entwicklung von multi-tier, thin-client Diensten.

- *J2EE Platform Specification*, eine Sammlung von Anforderungen, die eine J2EE-Plattform erfüllen muss.

[1] SOAP = Simple Object Access Protocol
[2] XML = Extensible Markup Language

In diesem Kapitel werden die Anforderungen der „Java™ 2 Platform Enterprise Edition Specification, v1.3“ von SUN Microsystems dargestellt. Sie bildet die Grundlage für diese Untersuchung. In der J2EE-Spezifikation wird auf weitere Spezifikationen, wie z.B. die JSP[1]-, Servlet- und die EJB[2]-Spezifikation verwiesen, so dass diese ebenfalls Gegenstand der Untersuchung sind.

Der WebSphere Application Server v4.0 von IBM beruht auf der J2EE-Spezifikation Version 1.2 und erweitert diese. Ich habe mich für die Beschreibung der aktuellen Version 1.3 der J2EE-Spezifikation entschieden, da sie zum Zeitpunkt des Beginns meiner Diplomarbeit, die aktuellste Version darstellte (mittlerweile ist Version 1.4 aktuell). Der WebSphere Application Server v5.0 implementiert die J2EE-Spezifikation Version 1.3 vollständig, ist aber noch nicht erhältlich und es sind nur wenige Informationen über ihn verfügbar.

Zur Beherrschung der Komplexität werden in der Spezifikation alle beteiligten Komponenten (Software genauso wie Menschen) klar definiert und ihr Zusammenwirken durch Verträge festgelegt. Diese werden im folgenden Abschnitt beschrieben.

2.1 Die Plattform Rollen

Die J2EE-Spezifikation definiert für alle beteiligten Entitäten so genannte *Java 2 Platform Enterprise Edition Roles*, denen Menschen oder auch Unternehmen entsprechen können. Die EJB-, JSP- und Servlet-Spezifikationen beschreiben weitere Untergruppen. Die Einteilung ist nicht bindend. Eine Organisation kann die Funktionalität anders aufteilen, um ihre Workflows bei der Applikationsentwicklung und beim Deployment[3] besser abbilden zu können.

2.1.1 J2EE Product Provider

Ein *J2EE-Produkt* kann ein Container für Komponenten, eine J2EE-Plattform API oder ein anderes Feature sein, das in der J2EE-Spezifikation definiert ist. Der *J2EE Product Provider* implementiert und liefert das J2EE-Produkt. Üblicherweise ist der J2EE Product Provider der Hersteller eines Betriebssystems, einer Datenbank, eines Applikationsservers oder eines Webservers.

Jeder J2EE Product Provider muss:

- die J2EE APIs den Applikationskomponenten durch Container verfügbar machen. Oft basiert die Implementation auf einer existierenden Infrastruktur des Product Providers.

- eine Abbildung (Mapping) der Applikationskomponenten zu den Netzwerkprotokollen bereitstellen, wie es in der Spezifikation beschrieben ist.

- Tools für das Deployment und Management der Applikationen zur Verfügung stellen. Deployment Tools ermöglichen es dem Deployer (s.u.), Applikationskomponenten auf dem J2EE-Produkt zu deployen. Management Tools erlauben es dem Systemadministrator (s.u.), das J2EE-Produkt und die darauf deployten Applikationen zu verwalten. Die Form dieser Tools wird durch die Spezifikation nicht vorgeschrieben.

[1] JSP = Java Server Pages
[2] EJB = Enterprise JavaBeans
[3] Um den Unterschied zur Installation zu verdeutlichen, werden „Deployment“ und „deploy“ in dieser Arbeit nicht übersetzt. Der Application Server wird installiert, aber Komponenten und Dienste werden deployed!

Einem J2EE-Produkt ist es freigestellt, Interfaces zu implementieren, die in der Spezifikation nicht enthalten sind.

Die EJB-Spezifikation [EJBS20] ordnet dem J2EE Product Provider die Untergruppen *J2EE-Server Provider* und den *EJB-Container Provider* zu. An den J2EE-Server Provider werden keine speziellen Anforderungen gestellt. An den *EJB-Container Provider* werden zu den o.g. Anforderungen, wie die Bereitstellung der Deployment Tools, weitere genannt, wie z.B. Bereitstellen einer Laufzeitumgebung für EJBs, Transaktionsmanagement, Sicherheitsmanagement, ein skalierbares Management von Ressourcen, die Verwaltung von verteilten Remote-Clients und andere in der Spezifikation definierten Dienste.

Die Servlet-Spezifikation [SERVLETS23] definiert zusätzlich den *Web- (Servlet-) Container Provider*.

Zusätzlich existiert in der J2EE-Spezifikation die Rolle *J2EE Platform Provider*. Diese Rolle wird zwar nirgends genau definiert, es ist aber anzunehmen, dass hiermit ein J2EE-Server Provider gemeint ist.

2.1.2 Application Component Provider

Die Rolle *Application Component Provider* umfasst den *HTML Document Designer*, den *Document Programmer* und den *Enterprise Beans Developer*. Angehörige dieser Rollen produzieren, mit der Hilfe von Tools, J2EE-Applikationen und Komponenten. Die EJB-Spezifikation definiert eine Untergruppe mit dem Namen *Enterprise Bean Provide*r, welche äquivalent zum *Enterprise Bean Developer* ist.

2.1.3 Application Assembler

Der *Application Assembler* stellt die vom Application Component Provider entwickelten Komponenten zu einer kompletten J2EE-Applikation, in Form einer Enterprise Archive (.ear) Datei, zusammen. Dazu nutzt der *Application Assembler* GUI Tools, die vom *Tool Provider* zur Verfügung gestellt werden. Der *Application Assembler* ist verantwortlich für die Bereitstellung von Zusammenstellungsinstruktionen, welche die externen Abhängigkeiten der Applikation beschreiben. Der *Deployer* muss die Zusammenstellungsinstruktionen während des Deployment-Prozesses befolgen.

2.1.4 Deployer

Der *Deployer* ist verantwortlich für das Deployment von Webapplikationen und Enterprise JavaBeans in die spezifische operative Umgebung. Dazu nutzt er Tools, die vom J2EE *Product Provider* zur Verfügung gestellt werden.

Das Deployment besteht normalerweise aus drei Schritten:

1. Es beginnt mit der *Installation*. Der Deployer bringt die Applikationsmedien auf den Server und generiert zusätzliche container-spezifische Klassen und Interfaces, die der Container zur Verwaltung der Applikationskomponenten benötigt. Anschließend deployed er die Applikationskomponenten sowie die zusätzliche Klassen und Interfaces in den entsprechenden Containern.

2. Während der nun folgenden *Konfiguration* werden die, vom Application Component Provider deklarierten, externen Abhängigkeiten aufgelöst und die, vom Application Assembler definierten, Applikationszusammenstellungsinstruktionen befolgt. So ist der Deployer zum Beispiel verantwortlich für Abbildung der Sicherheitsrollen auf die Benutzer und Gruppen, die auf dem Zielsystem existieren.

3. Zum Schluss startet der Deployer die *Ausführung* der neu installierten und konfigurierten Applikation. In einigen Fällen passt ein speziell qualifizierter Deployer die Businesslogik der Applikationskomponenten zur Deploymentzeit an. Beispielsweise nutzt der Deployer Tools, die mit dem J2EE-Produkt bereitgestellt wurden, um einfachen Applikationscode herzustellen, der als Wrapper (Schale) für Enterprise Bean Business Methoden dient. Oder er passt das Erscheinungsbild einer JSP-Seite an.

Der Deployer hinterlässt Webapplikationen, Enterprise Beans, Applets und Application Clients, die an die Zielumgebung angepasst wurden und in spezifischen Containern deployed sind.

2.1.5 System Administrator

Der *System Administrator* ist verantwortlich für die Konfiguration und Administration des Applikationsservers und des Netzwerkes. Er ist auch verantwortlich für die Beobachtung der korrekten Ausführung der deployten Applikationen. Dazu nutzt der Systemadministrator Runtime Monitoring und Management Tools, die vom *J2EE Product Provider* zur Verfügung gestellt werden.

2.1.6 Tool Provider

Ein *Tool Provider* liefert Tools, die für die Entwicklung und das Packaging (Zusammenstellung) der Applikationskomponenten genutzt werden. Die Anzahl an verfügbaren Tools korrespondiert mit der Anzahl an Typen von Applikationskomponenten, die von der J2EE-Plattform unterstützt werden. Plattformunabhängige Tools können in allen Phasen von der Entwicklung bis zum Deployment einer Applikation genutzt werden. Tools für das Deployment, Management und Monitoring der Applikationen können plattformabhängig sein. Zukünftige Versionen der Spezifikation definieren möglicherweise zusätzliche Interfaces, die plattformunabhängige Tools ermöglichen.

2.2 Die Plattform Verträge

Dieser Abschnitt beschreibt die J2EE-Contracts (Verträge), die vom J2EE Product Provider erfüllt werden müssen.

2.2.1 Netzwerkprotokolle

Zur Gewährleistung der Interoperabilität, ordnet die Spezifikation den verschiedenen Komponententypen verschiedene *Netzwerkprotokolle* zu, die den Industriestandards entsprechen. Mit diesen Protokollen können Systeme, die nicht der Technologie des J2EE-Produktes entsprechen, auf J2EE-Applikationskomponenten zugreifen. Zur Kommunikation mit Servlets und JSP-Seiten fordert die Spezifikation das HTTP und das HTTPS Protokoll. Für den Zugriff auf EJBs wird das IIOP Protokoll gefordert. Der J2EE Product Provider muss die Kommunikation mit den installierten Applikationskomponenten über die definierten Protokolle ermöglichen.

2.2.2 J2EE APIs

Die *J2EE APIs* definieren einen Vertrag zwischen den J2EE-Komponenten und der J2EE-Plattform. Der Vertrag spezifiziert die Runtime- und Deployment-Interfaces. Der J2EE Product Provider muss die J2EE APIs derart implementieren, dass die Semantik und Policies der Spezifikation erfüllt werden. Der Application Component Provider liefert Komponenten, die den APIs und Policies entsprechen.

2.2.3 J2EE Service Provider Interfaces (SPIs)

Die *J2EE Service Provider Interfaces (SPIs)* definieren einen Vertrag zwischen der J2EE-Plattform und den Service Providern, deren Dienste in ein J2EE-Produkt eingebunden werden können. Das *Connector API* definiert Service Provider Interfaces für die Integration von *Resource Adapter* in einen J2EE-Applikationsserver. Eine Resource Adapter Komponente, welche das Connector API implementiert, nennt man *Connector*. Der J2EE Product Provider muss die J2EE SPIs derart implementieren, dass die Semantik und Policies der Spezifikation eingehalten werden. Ein Provider von Service Provider Komponenten (z.B. ein Connector Provider) sollte Komponenten bereitstellen, die diesen SPIs und Policies entsprechen.

2.2.4 Deployment Descriptor

Ein *Deployment Descriptor* ist eine XML-Datei, die dem Deployer die Anforderungen der Applikation beschreibt. Er ist ein Vertrag zwischen dem Application Component Provider, Application Assembler und dem Deployer. In ihm müssen der Application Component Provider und der Application Assembler die Eigenschaften und Anforderungen der Applikationskomponenten bzw. der Applikation, wie externe Ressourcen, Sicherheit, Umgebungsparameter u.s.w. spezifizieren. Der J2EE Product Provider muss ein Deployment Tool bereitstellen, das J2EE Deployment Descriptoren interpretieren kann und es dem Deployer erlaubt, die Anforderungen der Applikationskomponenten auf die Fähigkeiten des spezifischen J2EE-Produktes und dessen Umgebung abzubilden.

2.3 Die J2EE Standarddienste

Die J2EE-Spezifikation fordert von jedem J2EE-Produkt, dass es verschiedene Standarddienste zur Verfügung stellt. Diese Standarddienste werden durch verschiedene APIs realisiert, die das J2EE-Produkt bereitstellen muss, um den Anforderungen, der im letzten Kapitel „Die Plattform Verträge“ angesprochenen Verträge, zu entsprechen. Einige dieser Dienste werden bereits durch die Java 2 Platform Standard Edition (J2SE) bereitgestellt. Diese J2EE Standarddienste (APIs) werden im folgenden kurz dargestellt und in den nächsten Kapiteln präzisiert.

2.3.1 HTTP

Das HTTP-Protokoll [RFC 1945] [RFC 2616] ist für die Kommunikation mit Web-Komponenten (HTML-Dokumenten, JSP-Seiten und Servlets) notwendig. Das HTTP API wird auf der Seite des Clients wird durch das *java.net* Package und auf der Seite des Servers durch die Servlet- und JSP-Interfaces definiert.

2.3.2 HTTPS

Die Verwendung des HTTP-Protokolls über SSL [SSL30] wird durch die gleichen Client und Server APIs wie beim HTTP-Protokoll definiert.

2.3.3 RMI-IIOP

Das IIOP-Protokoll[1] [CORBA231] wird für die Kommunikation mit Enterprise JavaBeans gefordert. Es wird vom *RMI-IIOP* Subsystem bereitgestellt, das aus mehreren APIs zusammengesetzt ist. Es ermöglicht eine objektorientierte Programmierung, unabhängig vom unterliegenden Netzwerkprotokoll. Die Implementierung dieser APIs erlaubt, das J2SE RMI[2]-Protokoll (JRMP) und das CORBA[3] IIOP-Protokoll zu nutzen. So können J2EE-Applikationen RMI-IIOP APIs mit dem IIOP-Protokoll für den Zugriff auf CORBA-Dienste verwenden, die jedoch mit den RMI-Programmier-Restriktionen kompatibel sein müssen. Solche CORBA-Dienste (z.B. Altsysteme) sind normalerweise außerhalb des J2EE-Produktes lokalisiert. Nur für J2EE-Applicationclients ist es erforderlich, direkt eigene CORBA-Dienste mit Hilfe der RMI-IIOP APIs zu definieren. Eine typische Anwendung von CORBA-Objekten ist der Gebrauch von s.g. Callbacks, wenn auf andere CORBA-Objekte zugegriffen wird.

J2EE-Applikationen müssen notwendiger Weise die RMI-IIOP APIs nutzen (speziell die *narrow* Methode von *java.rmi.PortableRemoteObject*), wenn sie auf Enterprise JavaBeans Komponenten, wie in der EJB-Spezifikation beschrieben, zugreifen. Dies erlaubt Enterprise JavaBeans protokollunabhängig zu sein. Darüber hinaus müssen J2EE-Produkte in der Lage sein, Enterprise Beans mit Hilfe des IIOP-Protokolls zu exportieren und auf Enterprise Beans, wie in der EJB 2.0 Spezifikation beschrieben, zuzugreifen. Das IIOP-Protokoll wird auch benötigt, um eine Interoperabilität zwischen verschiedenen J2EE-Produkten zu gewährleisten, die jedoch auch weitere Protokolle verwenden dürfen.

2.3.4 Java IDL

Mit JavaIDL[4] können Applikationskomponenten und J2EE-Applikationen, als Clients, externe CORBA-Objekte [J2IDLS][IDL2JS], die in einer beliebigen Programmiersprache geschrieben wurden, über das IIOP-Protokoll aufrufen. Nur J2EE-Applicationclients dürfen JavaIDL verwenden, um sich selbst als CORBA-Dienste zu präsentieren.

2.3.5 JDBC API

Das JDBC[5] API [JDBCS21] dient der Anbindung von relationalen Datenbanksystemen und besteht aus zwei Teilen. Einem Interface auf Applikationsebene, das von Applikationskomponenten für den Zugriff auf eine Datenbank genutzt wird, und einem Service Provider Interface für den Anschluss von Datenbanktreibern.

[1] Internet Inter-ORB Protocol
[2] RMI = Remote Method Invocation
[3] OMG´s CORBA = Common Object Request Broker Architecture
[4] IDL = Interface Definition Language
[5] JDBC = Java DataBase Connectivity

2.3.6 Java Naming and Directory Interface (JNDI)

Mit Hilfe des JNDI API [JNDIS12] kann auf Namens- und Verzeichnisdienste zugegriffen werden. Es besteht aus zwei Teilen. Einem Interface auf Applikationsebene, das von Applikationskomponenten für den Zugriff auf Namens- und Verzeichnisdienste verwendet wird, und einem Service Provider Interface für den Anschluss weiterer Namens- und Verzeichnisdienste.

2.3.7 Java Transaction API (JTA)

Das Java Transaction API [JTAS101] besteht aus zwei Teilen. Einer Abgrenzungsschnittstelle auf Applikationsebene, die vom Container und Applikationskomponenten benutzt wird, um Transaktionsgrenzen zu kennzeichnen, sowie einem Interface zwischen Transaktionsmanager und Ressourcemanager, das auf Ebene des J2EE SPI genutzt wird (in zukünftigen Releases).

2.3.8 Java Message Service (JMS)

Der Java Messaging Service [JMSS102] ist ein Standard API für Messaging, welche zuverlässiges Point-to-Point Messaging ebenso wie das Publish-Subscribe Modell unterstützt. Die Spezifikation fordert einen JMS Provider, der beides implementiert, Point-to-Point Messaging ebenso wie das Publish-Subscribe Modell.

2.3.9 JavaMail

Mit dem JavaMail API [JMAILS11] kann eine Applikationskomponente Emails versenden. Es besteht aus zwei Teilen. Einem Interface auf Applikationsebene, welches von Applikationskomponenten genutzt wird, um Mails zu versenden, und einem Service Provider Interface, welches auf J2EE SPI Ebene genutzt wird.

2.3.10 JavaBeans Activation Framework (JAF)

Das JavaMail API nutzt das JAF API [JAFS10], daher ist es ebenfalls erforderlich.

2.3.11 Java API for XML Parsing (JAXP)

JAXP [JAXPS10] unterstützt die Industriestandard APIs SAX[1] und DOM[2] zum Parsen von XML Dokumenten, und es werden auch XSLT[3] Transformations Engines unterstützt.

2.3.12 J2EE Connector Architecture

Die Connector Architecture [CONS10] ist ein J2EE SPI. Es erlaubt, Resource Adapter, die den Zugriff auf Enterprise Information Systems (EIS) unterstützen, in jedes J2EE-Produkt einzubauen. Es definiert zwischen dem J2EE-Server und einem Resource Adapter diverse Standardverträge auf Systemebene.

[1] SAX = Simple API for XML
[2] DOM = Document Object Model
[3] XSLT = Extensible Stylesheet Language Transformations

Ein Standardvertrag (Standard Contract) beinhaltet:

- Einen *Connection Management Contract*, der einem J2EE-Server das „Pooling“[1] von Verbindungen zum darunter liegenden EIS erlaubt. Des weiteren erlaubt dieser Vertrag Applikationskomponenten, sich mit dem EIS zu verbinden. Dies führt zu einer skalierbaren Applikationsumgebung, so dass eine große Anzahl von Clients auf ein EIS zugreifen können.

- Einen *Transaktion Manager Contract* zwischen dem Transaktionsmanager und einem EIS, das einen transaktionsgestützten Zugriff auf mehrere EIS Resource Manager unterstützt. Mit diesem Vertrag kann ein J2EE-Server einen Transaktionsmanager für die Verwaltung von Transaktionen über mehrere Resource Manager nutzen. Es werden auch Transaktionen unterstützt, die vom EIS intern verwaltet werden, d.h. ohne der Notwendigkeit einen externen Transaktionsmanager zu verwenden.

- Einen *Security Contract*, der den sicheren Zugriff zum EIS gewährleistet.

2.3.13 Java Authentication and Authorization Service (JAAS)

Mit JAAS [JAASS10] können Dienste Benutzer authentisieren und so eine Zugriffskontrolle durchsetzen. Es implementiert eine Javaversion des Standard PAM[2] Frameworks und erweitert die Zugriffskontrollarchitektur der Java 2 Plattform in kompatibler Art und Weise zur benutzerbasierten Autorisierung. Weitere Informationen sind im Kaptitel „J2SE-Security“ zu finden.

2.4 Resource Manager Driver

Ein *Resource Manager Driver*, kurz Driver, ist eine Softwarekomponente auf Systemebene, die eine Netzwerkschnittstelle zu einem externen Resource Manager implementiert. Ein Driver kann die Funktionalität der J2EE-Plattform erweitern, indem er entweder eines der J2EE Standard Service APIs (z.B. JDBC Treiber) implementiert, oder er definiert und implementiert eine Verbindung zu einem externen Applikationssystem. Der Driver nutzt als Schnittstelle zur J2EE-Plattform ein J2EE Service Provider Interface (J2EE SPI). Ein Driver, der ein J2EE SPI nutzt, ist auf jedem J2EE-Produkt lauffähig.

2.5 Die Datenbank

Die J2EE-Plattform benötigt eine Datenbank, auf welche durch das JDBC API zugegriffen werden kann. Web-Komponenten, Enterprise Beans und von Application Clients muss der Zugriff auf die Datenbank ermöglicht werden. Ein Zugriff von Applets auf die Datenbank wird nicht gefordert.

[1] engl. Pooling = Verwalten von Verbindungen, zur Steigerung der Performanz
[2] PAM = Plugable Authentication Module

2.6 Der J2EE-Server

Die Spezifikationen definieren keine speziellen Anforderungen an den Server, jedoch an die jeweiligen Container. Es wird nicht zwischen Server- und Container-Provider unterschieden.

Ein J2EE-konformer Server ist eine Laufzeitumgebung für verschiedene Container (s.u.). Aufgaben des Servers sind nach Denninger [Denninger 00] beispielsweise:

- Thread- und Prozessmanagement (damit auf einem Server mehrere Container parallel ihre Dienste anbieten können)
- Unterstützung von Clustering und Lastverteilung (d.h. die Fähigkeit, mehrere Server im Verbund zu betreiben und die Anfragen der Clients je nach Auslastung zu verteilen, um bestmögliche Antwortzeiten zu gewährleisten)
- Ausfallsicherheit
- Namens- und Verzeichnisdienst (um Komponenten auffinden zu können)
- Zugriffsmöglichkeit auf und das Pooling von Betriebssystemressourcen (z.B. Netzwerksockets für den Betrieb eines Web-Containers)

Üblicherweise implementiert ein J2EE Product Provider die serverseitige J2EE Funktionalität, indem er eine existierende transaktionsverarbeitende Infrastruktur nutzt und sie mit der Java 2 Platform, Standard Edition (J2SE) Technologie kombiniert. Die Schnittstelle zwischen Server und den Containern ist dabei sehr stark vom Hersteller abhängig. Die Spezifiktionen definieren hierfür kein Protokoll. Die J2EE Client Funktionalität wird üblicherweise mit Hilfe der J2SE Technologie implementiert (siehe Ressource Manager Driver).

J2EE-Server Unterstützung für Applikationskomponenten

Ein J2EE-Server unterstützt die Stationierung (Deployment), Verwaltung (Management) und Ausführung (Execution) von J2EE-konformen Applikationskomponenten. Diese können je nach Abhängigkeit vom J2EE-Server in drei Kategorien eingeteilt werden:

a) Komponenten, die auf einen J2EE-Server stationiert, verwaltet und ausgeführt werden. Dies schließt Web-Komponenten und Enterprise JavaBean Komponenten ein.

b) Komponenten, die auf einen J2EE-Server stationiert und verwaltet werden, dann aber auf eine Client Maschine geladen und ausgeführt werden. Diese sind HTML-Dokumente und Applets, die in HTML-Dokumente eingebunden sind.

c) Komponenten, deren Stationierung und Verwaltung nicht klar durch die J2EE-Spezifikation definiert ist. Application Clients fallen unter diese Kategorie. Zukünftige Version der J2EE-Spezifikation werden dies wohl näher spezifizieren.

2.7 Die J2EE-Container

Container stellen J2EE-Komponenten eine Laufzeitumgebung zur Verfügung. Sie bieten Applikationskomponenten einen vereinheitlichten Blick auf die darunter liegenden J2EE APIs. J2EE-Applikationskomponenten interagieren niemals auf direktem Weg mit anderen J2EE-Applikationskomponenten. Sie nutzen Protokolle und Methoden des Containers, um miteinander und mit anderen Plattformdiensten zu interagieren.

Das Zwischenschalten eines Containers zwischen die Applikationskomponenten und den J2EE-Diensten erlaubt es dem Container transparent Dienste einzufügen, die im Deployment Descriptor der Komponente definiert wurden. Wie zum Beispiel deklaratives Transaktionsmanagement, Sicherheitsüberprüfungen (Security Checks), Resource Pooling und Zustandsverwaltung (State Management).

Ein typisches J2EE-Produkt wird für jeden Applikationskomponententyp einen Containertyp bereitstellen: Applicationclient-Container, Applet-Container, Web-Container und EJB-Container.

Abbildung 2 zeigt die verschiedenen J2EE-Dienste und Container.

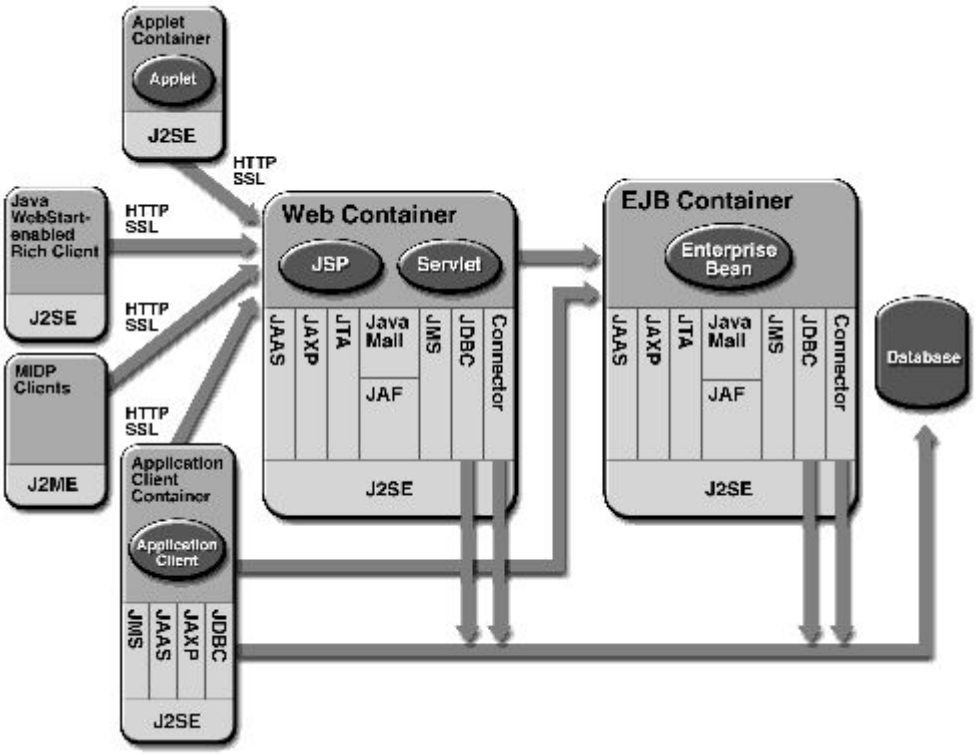

Abbildung 2: J2EE-Container und Dienste aus [DEAJ2EE]

Die J2EE 1.3 Spezifikation fordert, dass jeder Container ein „Java Compatible Runtime Environment" bietet, wie es in der J2SE 1.3 Spezifikation definiert ist. Ein Applet-Container kann das Produkt „Java Plug-in" nutzen, um diese Funktionalität sicher zu stellen, oder eine eigene native Implementierung bieten.

Die Spezifikation definiert, dass Container vom J2EE Product Provider implementiert werden und dass die Container Tools die Dateiformate verstehen müssen, die für das Packaging der

Applikationskomponenten zum Deployment verwendet werden. Sie definiert verschiedene Standarddienste, die jedes J2EE-Produkt unterstützen muss. Die Container bieten APIs, die Applikationskomponenten nutzen, um auf diese Standarddienste zuzugreifen. Des weiteren beschreibt die Spezifikation Standardwege, wie J2EE-Dienste mit so genannten *Connectoren* mit nicht-J2EE-Applikationssystemen (wie z.B. Mainframe und ERP Systeme) zu verbinden sind.

2.7.1 Applet-Container

Ein *Applet-Container* hostet[1] Applets, die von einer Website geladen wurden. Die Spezifikationen stellen keine Anforderungen an diesen Typ von Container. Es wird auch nicht gefordert, dass ein Applet direkt auf Remote-Methoden von EJBs zugreifen kann. Üblicherweise ist die Implementation eines Applet-Containers eine Kombination aus Webbrowser und Java Plug-in.

2.7.2 Applicationclient-Container

Ein *Applicationclient-Container* hostet stand-alone J2EE-Applikationen. Die EJB-Spezifikation stellt noch besondere Anforderungen an die Interoperabilität, wie die Unterstützung von RMI, JNDI, Schutz der Integrität und Vertraulichkeit von Nachrichten, Authentisierung und Propagierung von Authentisierungsdaten.

2.7.3 Web-Container

Nach der Servlet-Spezifikation ist ein Web-Container (auch Servlet-Container genannt) Teil eines Web- oder Applikationsservers. Die Server stellen die notwendigen Netzwerkdienste zur Verfügung, über welche die Requests (Anfragen) und Responses (Antworten) versandt, MIME basierte Anfragen dekodiert und MIME basierte Antworten kodiert werden. Der Container hostet Servlets und verwaltet ihren Lebenszyklus.

Ein Web-Container wird entweder direkt in einem Webserver oder webfähigen Applikationsserver implementiert, oder als zusätzliche (add-on) Komponente, über ein natives Erweiterungs-API, angebunden.

Nach der Servlet-Spezifikation muss jeder Web-Container mindestens die Version HTTP/1.0 des HTTP-Protokolls implementieren. Weitere Request/Response basierende Protokolle wie HTTPS (HTTP über SSL) können ebenfalls unterstützt werden. Die Servlet-Spezifikation empfiehlt nachdrücklich, dass Container zusätzlich auch HTTP nach der HTTP/1.1 Spezifikation implementieren. Weiter fordert die Servlet-Spezifikation, dass ein Web-Container mindestens auf J2SE 1.2 beruhen muss.

Wie beim Applicationclient-Container stellt die EJB-Spezifikation noch besondere Anforderungen an die Interoperabilität, wie die Unterstützung von RMI, JNDI, Schutz der Integrität und Vertraulichkeit von Nachrichten, Authentisierung und Propagierung von Authentisierungsdaten.
Um die Sicherheit zu erhöhen, kann ein Container in der Umgebung, in der das Servlet ausgeführt wird, Restriktionen verhängen. In der Java 2 Platform werden diese Restriktionen mit Hilfe der Permission-Architektur definiert. Zum Beispiel könnten high-level Applikationsserver die Erzeugung von *Thread* Objekten limitieren, um sicherzustellen, dass keine anderen Komponenten des Containers negativ beeinträchtigt werden.

[1] hostet = beherbergt

2.7.4 EJB-Container

Ein typischer EJB-Container bietet eine skalierbare Laufzeitumgebung für eine große Anzahl von nebenläufigen EJB-Objekten.

Nach der EJB-2.0-Spezifikation muss ein EJB-2.0-Container den EJB-Exemplaren folgende APIs zur Verfügung stellen. Ein Container-Provider darf jedoch weitere Dienste hinzufügen.

Java 2 Platform, Standard Edition v1.3 (J2SE) APIs

- EJB 2.0 APIs
- JNDI 1.2 (Java Naming and Directory Interface), zur Lokalisierung von *Home-Interfaces* der EJBs und Ressourcen
- JTA 1.0.1 (Java Transaction API), nur das *UserTransaction* Interface zur Transaktionssteuerung
- JDBC™ 2.0 Erweiterung (Java DataBase Connectivity), zur Datenbankanbindung
- JMS 1.0.2 (Java Messaging Service), als Messaging-Dienst für u.a. Message-Driven-EJBs
- JavaMail 1.1, zum Versenden von Emails
- JAXP 1.0 (Java API for XML Parsing), zum Parsen von u.a. Deployment Descriptoren

Wie bei den anderen Containern, stellt die EJB-Spezifikation noch besondere Anforderungen an die Interoperabilität, wie die Unterstützung von RMI, JNDI, Schutz der Integrität und Vertraulichkeit von Nachrichten, Authentisierung und Propagierung von Authentisierungsdaten.

Der Container ist maßgeblich für die Sicherheit der EJBs und der Umgebung des Applikationsservers verantwortlich. Aus diesem Grund gibt die EJB-Spezifikation folgende Security Policy vor. Sie beschreibt, welche Rechte ein Container einer Enterprise JavaBean zur Laufzeit geben darf. Hierbei bedeutet *grant*, dass der Container dieses Recht an eine EJB vergeben darf, und *deny*, dass er dieses Recht nicht vergeben darf. (Die Konzepte von Permissions und Policies werden im Kapitel „J2SE-Security“ erläutert.)

Permission Name	**EJB-Container Policy**
java.security.AllPermission	*deny*
java.awt.AWTPermission	*deny*
java.io.FilePermission	*deny*
java.net.NetPermission	*deny*
java.util.PropertyPermission	*grant “read”, “*”,* *deny all other*
java.lang.reflect.ReflectPermission	*deny*
java.lang.RuntimePermission	*grant “queuePrintJob”,* *deny all other*
java.lang.SecurityPermission	*deny*
java.io.SerializablePermission	*deny*
java.net.SocketPermission	*grant “connect”, “*” ,* *deny all other*

EJBs unterliegen also sehr strengen Restriktionen. Auffällig ist die *connect* Erlaubnis der *java.net.SocketPermission*, diese ist für EJBs notwendig, um Client-Funktionalitäten der JavaIDL und RMI-IIOP APIs nutzen zu können, welche Teile der Java 2 Plattform sind.

Der EJB-Container stellt den EJBs eine Laufzeitumgebung zur Verfügung und bietet ihnen verschiedene Dienste an. Die wichtigsten Funktionen des Containers seien im folgenden kurz dargestellt.

Kontrolle des Lebenszykluses einer Bean (Laufzeitumgebung)

Der Lebenszyklus von EJBs wird durch den EJB-Container verwaltet. Er erzeugt bei Clientanfragen Exemplare der Bean-Klassen und versetzt sie über Callback-Methoden in verschiedene Zustände (siehe Kapitel „Enterprise JavaBeans“). Zur Optimierung der Performanz verwaltet der Container einige Bean-Typen in Pools (Pooling). Das Löschen der Bean-Exemplare geschieht auch wieder durch den Container.

Exemplar-Pooling und Aktivierung bzw. Passivierung (Laufzeitumgebung)

Da die Exemplarerzeugung von Objekten sehr kostenaufwendig ist und um die Anzahl von Objekten möglichst klein zu halten, verwaltet der Container Bean-Exemplare in einem Pool. Befindet sich ein Exemplar im Pool, hat es den Zustand *pooled* und ist gewissermaßen deaktiviert. Wird nun ein Exemplar benötigt, so wird ein Bean-Exemplar passenden Typs dem Pool entnommen, es reaktiviert und sein Zustand auf *ready* gesetzt.

Um Systemressourcen (speziell Speicher) zu schonen, werden Bean-Exemplare, die zur Zeit nicht benötigt werden, mit Hilfe von Objektserialisierung persistent gemacht und aus dem Speicher entfernt (Passivierung). Werden die Exemplare wieder benötigt, so werden die persistenten Objekte wieder deserialisiert und im Speicher zur Verfügung gestellt (Aktivierung).

Die Fähigkeit zum Exemplar-Pooling (auch Instanzen-Pooling genannt), Aktivierung und Passivierung hängt vom Typ des EJB ab. Die EJB-Spezifikation fordert kein Exemplar-Pooling des Containers.

Verteilung (Laufzeitumgebung)

Der Container sorgt für die Ortstransparenz der Enterprise Beans. Ein Client muss daher nicht wissen, auf welchem Server die Bean läuft, für ihn unterscheidet sich die Benutzung einer Bean auf einem anderen Rechner nicht wesentlich von der Benutzung eines Objekts auf dem lokalen Rechner. Die verteilte Kommunikation wird vom Container verwaltet. Dies geschieht mittels Java RMI oder RMI-IIOP (kompatibel mit CORBA).

Namens- und Verzeichnisdienst (Dienst)

Zum Auffinden von Objekten (spez. Beans), Diensten und Ressourcen stellt der EJB-Container einen Namen- und Verzeichnisdienst zur Verfügung. Mit einem Namensdienst kann man Referenzen auf Objekte unter einem bestimmten Namen an einem bestimmten Ort hinterlegen (*Binding*). Gebundene Objekte lassen sich dann über ihren Namen finden (*Lookup*). Ein Verzeichnisdienst ist mächtiger als ein Namensdienst. Hier werden die Objekte und andere Ressourcen in einer hierarchischen Struktur hinterlegt und es lassen sich noch weitere beschreibende Informationen hinterlegen. Eine Schnittstelle zu Names- und Verzeichnisdiensten ist JNDI.

Persistenz (Dienst)

Der EJB-Container muss einen Persistenzmechanismus anbieten, über den eine Bean persistent gemacht werden kann. Dies bedeutet, dass der Container automatisch die Daten der Bean in einem beliebigen Medium (z.B. Datenbank) ablegt und später, bei einer erneuten Initialisierung, die Bean mit ihren persistenten Daten wieder initialisiert wird. Für die Bean ist es unerheblich, wo die Daten gespeichert werden. Der Container trägt die Verantwortung, dass die Daten der Beans immer in einem konsistenten Zustand gehalten werden. Dieser Mechanismus ist für Entity-Beans mit *Container Managed Persistance* von entscheidender Bedeutung. Diese EJBs sind mit jedem Persistenzmedium kompatibel. Ihre Performanz ist allerdings deutlich schlechter als bei anderen Bean-Typen. Dazu später mehr im Kapitel „Entity-Beans“. Im übrigen lässt sich der Persistenzmechanismus auch zur Integration von Altsystemen nutzen.

Transaktionen (Dienst)

Nach der EJB-Spezifikation muss jeder EJB-Container flache (nicht geschachtelte) Transaktionen unterstützen. Der Container hat hier die Funktion eines Transaktionsmonitors.

Es wird zwischen *bean-managed transaction demarcation* und *container-managed transaction demarcation* unterschieden. Bei expliziter Transaktionsunterstützung *(bean-managed transaction demarcation)* wird die Transaktion aus dem Programmcode der Bean heraus geregelt (durch die Benutzung eines speziellen Interfaces). Bei deklarativer Transaktionsunterstützung *(container-managed transaction demarcation)* wird das Verhalten im Deployment Descriptor der Bean definiert.

Der Container muss die dazu notwendigen Protokolle bereitstellen, einschließlich das *Two Phase Commit Protocol* zwischen einem Transaktionsmonitor und einem Datenbanksystem oder einem JMS Provider, die *Transactional Context Propagation* und den verteilten *Two Phase Commit*.

Die Spezifikation verlangt, dass der Container das *Java Transaction API (JTA)* und das *Connector API* unterstützt. JTA bildet eine Schnittstelle zwischen dem Transaktionsmanager und den anderen Teilen, die an dem verteilten Transaktionssteuerungssystem beteiligt sind, wie Applikationen, Ressourcemanager und Applikationsserver.

Die Spezifikation fordert keine explizite Unterstützung des *Java Transaction Services (JTS) API*, welches eine Javaversion der *CORBA Object Transaction Services (OTS) 1.1* Spezifikation implementiert. JTS unterstützt die Interoperabilität von Transaktionen über das IIOP-Protokoll zur Propagierung der Transaktionen zwischen Servern.

Sicherheit (Laufzeitumgebung)

Der Container ist für die Durchsetzung von Authentisierung, Autorisierung und einer sicheren Kommunikation, gemäß der im Deployment Descriptor definierten Security Policy, verantwortlich. Eine genauere Betrachtung sowie die Anforderungen der EJB-Spezifikation sind im Kapitel „J2EE-Security“ zu finden.

2.8 Die J2EE-Architektur

Die logischen Beziehungen der die Architektur bestimmenden Elemente der J2EE-Plattform sind in Abbildung 3 zu sehen. Dies impliziert nicht die physikalische Aufteilung der Elemente auf verschiedene Maschinen, Prozesse, Adressräume oder virtuelle Maschinen.

Die Container, dargestellt durch einzelne Rechtecke, sind J2EE-Laufzeitumgebungen (J2EE Runtime Environments), welche den Applikationskomponenten die benötigten Dienste zur Verfügung stellen. Die Applikationskomponenten sind in der oberen Hälfte der Rechtecke abgebildet und die Dienste in der unteren Hälfte durch kleine Boxen. Die APIs der Java 2 Platform Standard Edition (J2SE) werden von jeden Containertyp angeboten.

Zum Beispiel stellt der Applicationclient-Container einer Applicationclient-Komponente das Java Messaging Service (JMS) API zur Verfügung. Die einzelnen Komponenten werden im nächsten Kapitel detaillierter beschrieben.

Die Pfeile kennzeichnen die benötigten Zugriffe auf andere Teile der J2EE-Plattform. Der Applicationclient-Container bietet Application Clients einen direkten Zugriff auf eine Datenbank mit Hilfe des JDBC API. Ein ähnlicher Zugriff auf Datenbanken wird JSP-Seiten und Servlets durch den Web-Container, und Enterprise JavaBeans durch den EJB-Container ermöglicht.

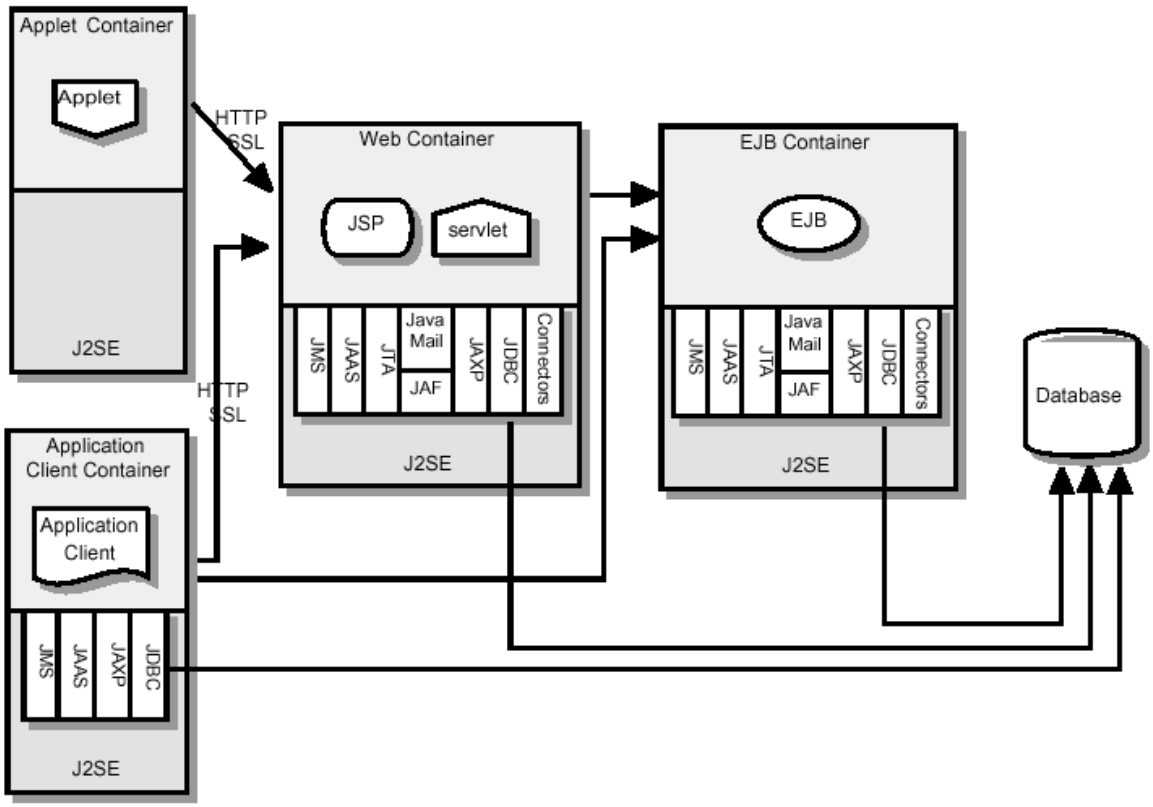

Abbildung 3: J2EE-Architektur Übersicht aus [J2EES13]

2.9 Die Applikationskomponenten

Clientseitig sind zwei Typen von „Applikationen“ zu finden:

- *Application Clients,* sind Java Programme, die üblicherweise auf einem Desktop Computer laufen und eine graphische Benutzerschnittstelle bieten. Sie verhalten sich aus der Sicht des Benutzers wie „normale” (native) Applikationen und können direkt mit Web- und EJB-Containern interagieren.

- *Applets,* sind GUI Komponenten, die normalerweise in einem Web Browser laufen. Diese können aber auch von anderen Applikationen oder Geräten, die das Programmiermodell von Applets unterstützen, ausgeführt werden. Applets können mit serverseitigen Webapplikationen nur über einen Web-Container interagieren.

Webapplikationen werden serverseitig aus unterschiedlichen Komponenten und anderen Elementen wie z.B. HTML-Dokumenten zusammengestellt. Die Spezifikationen definieren verschiedene Typen von Applikationskomponenten, die jedes J2EE-Produkt unterstützen muss.

2.9.1 Servlets und JSPs

Servlets, JSP-Pages, Filter und *Web Event Listener* sind serverseitige Komponenten, die in einem Web-Container laufen. Sie kommunizieren mit Clients über Protokolle, die auf Request/Response Nachrichten basieren.

Mit Servlets und JSP-Seiten lassen sich HTML-Dokumente generieren, die dann eine Benutzerschnittstelle einer Applikation bilden. Eine Generierung von XML-Dokumenten und anderen Formaten ist ebenfalls möglich, die dann von weiteren Komponenten genutzt werden können.

2.9.1.1 Servlet Definition

Nach der Java Servlet-Spezifikation v2.3 [SERVLETS23] von SUN ist ein Servlet eine webbasierte Komponente, die dynamische Inhalte generiert und von einem Web-Container verwaltet wird. Servlets sind plattformunabhängige Klassen, diese werden in plattformneutralen Bytecode kompiliert und anschließend dynamisch in einen javafähigen Webserver geladen und dort ausgeführt. Web-Container werden manchmal auch Servlet Engines genannt. Diese sind Servererweiterungen, die den Server um Funktionalitäten für Servlets erweitern. Servlets interagieren mit Webclients (z.B. Webbrowser) über das Request/Response Paradigma, das der Servlet-Container (Web-Container) implementiert.

Vergleich mit anderen Technologien

Die Funktionalität von Servlets liegt irgendwo zwischen CGI[1] Programmen und proprietären Servererweiterungen, wie z.B. die Netscape Server API (NSAPI) oder den Apache Modulen. Servlets haben, gegenüber anderen Mechanismen zur Servererweiterung, folgende Vorteile:

- Sie sind generell schneller als CGI Skripte, da sie ein anderes Prozessmodell nutzen.
- Sie nutzen ein Standard-API, das von vielen Webservern unterstützt wird.
- Sie haben alle Vorteile der Programmiersprache Java, einschließlich der einfachen Entwicklung und der Plattformunabhängigkeit.
- Sie können eine große Anzahl von APIs verwenden, die für die Java Plattform verfügbar sind.

Exemplare von Servlets

Ein Servlet muss im Deployment Descriptor der Webapplikation, die das Servlet enthält, deklariert werden. Diese Deklaration beschreibt auch, wie das Servlet zu erzeugen ist. Dies wird ausführlich in der Servlet-Spezifikation, im Kapitel „Deployment Descriptor“ beschrieben. Für ein Servlet, das nicht in einer verteilten Umgebung (Grundeinstellung) gehostet wird, darf der Web-Container nur ein Servlet-Exemplar pro Servlet Deklaration erzeugen. Dennoch kann der Servlet-Container mehrere Exemplare erzeugen, um eine starke Nachfragelast abwickeln zu können. Dazu muss er die Requests zu dem jeweiligen Exemplar serialisieren und das Servlet muss das so genannte *SingleThreadModel* Interface implementieren.

In dem Fall, dass das Servlet als Teil einer Applikation deployed wurde, die im Deployment Descriptor als verteilbar (*distributable*) gekennzeichnet wurde, sind mehrere Exemplare möglich. Allerdings darf der Container nur ein Exemplar pro Servlet-Deklaration pro Virtual Machine (VM) besitzen. Ist das Servlet Teil einer verteilten Applikation und implementiert es das *SingleThreadModel* Interface, kann der Container mehrere Exemplare des Servlets, in je einer Virtual Machine des Containers, erzeugen.

Die Nutzung des *SingleThreadModel* Interfaces garantiert, dass nur ein Thread zur Zeit die *service* Methode (s.u.) des Servlets ausführt**.** Es ist wichtig zu bemerken, dass diese Garantie nur für das jeweilige Servlet-Exemplar gilt, da der Container diese Objekte in einem Pool verwaltet. Objekte, auf die von mehr als einem Servlet-Exemplar zur Zeit zugegriffen wird, wie z.B. Exemplare von *HttpSession*, bleiben für mehrere Servlets jederzeit verfügbar, einschließlich der, die das *SingleThreadModel* implementieren.

2.9.1.2 Servlet Lebenszyklus

Ein Servlet besitzt einen wohl definierten Lebenszyklus, dieser definiert, wie es geladen, erzeugt und initialisiert wird, Anfragen von Clients behandelt und wie sein Dienst wieder entladen wird. Dieser Lebenszyklus wird in dem API durch die Methoden *init, service* und *destroy* des Interfaces *javax.servlet.Servlet* ausgedrückt, das alle Servlets direkt oder indirekt, durch die abstrakten Klassen *GenericServlet* oder *HttpServlet*, implementieren müssen.

[1] CGI = Common Gateway Interface

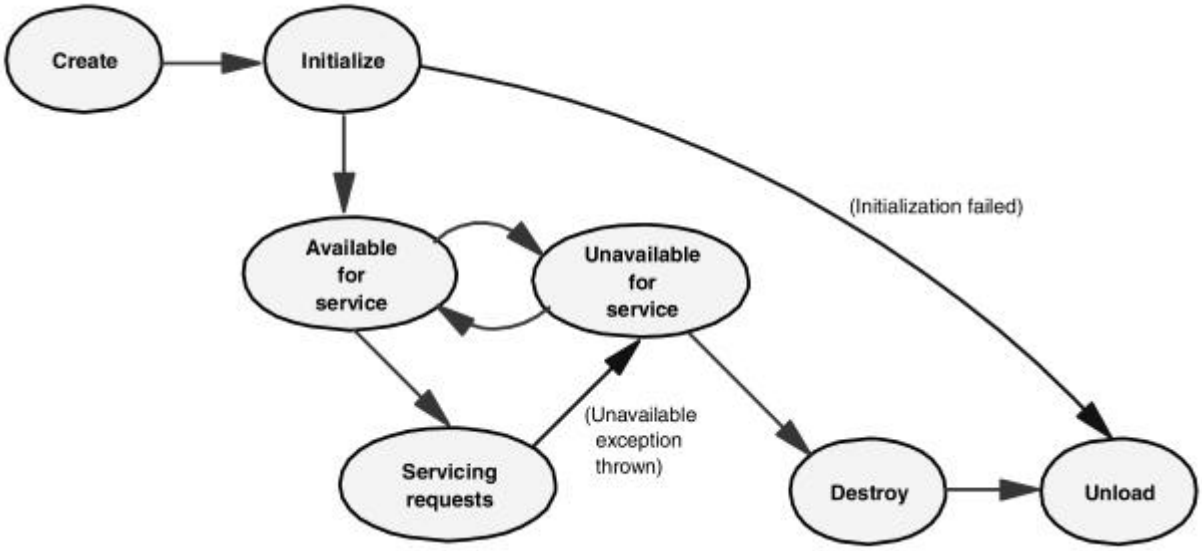

Abbildung 4: Servlet Lebenszyklus aus [SG245755]

Laden und Exemplarerzeugung

Der Servlet-Container ist für das Laden und die Exemplarerzeugung des Servlets verantwortlich. Dies kann beim Starten des Containers geschehen oder verspätet, wenn der Container feststellt, dass das Servlet zur Bedienung einer Anfrage benötigt wird. Wenn die Servlet Engine gestartet wird, muss der Servlet-Container alle vom Servlet benötigten Klassen lokalisieren. Der Servlet-Container lädt die Klassen mit Hilfe der normalen Java Classloading Einrichtungen. Das Laden kann von einem lokalen Dateisystem, entfernten Dateisystem oder anderen Netzwerkdiensten geschehen. Nach dem Laden der Servlet-Klasse erzeugt der Container ein Exemplar der Klasse.

Initialisierung

Nachdem ein Servlet-Exemplar erzeugt wurde, muss der Container das Servlet initialisieren, bevor es Anfragen von Clients behandeln kann. Während der Initialisierung liest das Servlet persistente Konfigurationsdaten ein, initialisiert kostspielige Ressourcen (wie z.B. JDBC API basierende Datenbankverbindungen) und führt andere einmalige Aktivitäten aus.

Der Container initialisiert das Servlet, indem er die *init* Methode des *Servlet* Interfaces mit einem Konfigurationsobjekt ruft. Dieses Konfigurationsobjekt ist für jede Servlet-Deklaration eindeutig und muss das *ServletConfig* Interface implementieren. Es erlaubt dem Servlet auf Name/Wert Paare mit Initialisierungsparametern, aus den Konfigurationsinformationen der Webapplikation, zuzugreifen. Das Konfigurationsobjekt gibt dem Servlet auch Zugriff auf ein Objekt, welches das *ServletContext* Interface implementiert und die Laufzeitumgebung des Servlets beschreibt. Dies wird ausführlich in der Servlet-Spezifikation, im Kapitel „Servlet Context“, beschrieben.

Ausnahmen (Exceptions) während der Initialisierung

Während der Initialisierung kann ein Servlet-Exemplar eine *UnavailableException* oder eine *ServletException* erzeugen. In diesen Fall darf das Servlet nicht in den aktiven Dienst genommen werden und muss vom Container entfernt werden, ohne die *destroy* Methode des Servlets zu rufen. Nach einer erfolglosen Initialisierung kann vom Container ein neues Exemplar erzeugt und initialisiert werden. Die Ausnahme zu dieser Regel ist, wenn die

UnavailableException nur eine minimale Zeit von Nichtverfügbarkeit anzeigt und der Container dann diese Zeitperiode warten muss, bevor er ein neues Servlet-Exemplar erzeugt und initialisiert.

Bearbeitung von Anfragen (Requests)

Nach der ordnungsgemäßen Initialisierung des Servlets, kann es vom Container zur Bearbeitung von Client-Anfragen genutzt werden. Anfragen werden durch Request-Objekte vom Typ *ServletRequest* repräsentiert. Das Servlet antwortet auf die Anfragen, indem es Methoden eines Response-Objekts vom Typ *ServletResponse* ruft und dort eine Antwort formuliert. Diese Objekte werden der *service* Methode des *Servlet* Interfaces als Parameter übergeben. Im Falle einer HTTP-Anfrage sind die vom Container gelieferten Objekte von den Typen *HttpServletRequest* und *HttpServletResponse*. Es ist zu beachten, dass ein Servlet-Exemplar während seiner Lebenszeit eventuell keine Anfragen zu bearbeiten hat.

Multithreading-Probleme

Ein Servlet-Container kann mehrere nebenläufige Anfragen in die *service* Methode des Servlets leiten. Zur korrekten Bearbeitung dieser Anfragen muss der Entwickler des Servlets adäquate Vorkehrungen treffen, die ein nebenläufiges Arbeiten mehrerer Threads in der *service* Methode gestattet. Für den Entwickler besteht die Alternative, das *SingleThreadModel* Interface zu implementieren. Dieses fordert von dem Container eine Garantie, dass sich zu jedem Zeitpunkt nur ein Anfrage-Thread in der *service* Methode befindet. Ein Container kann diese Anforderung erfüllen, indem er alle Anfragen an das Servlet serialisiert oder einen Pool mit Servlet-Exemplaren verwaltet. Ist das Servlet Teil einer Webapplikation, die als *distributable* (verteilbar) gekennzeichnet wurde, kann der Container für jede VM, über die die Applikation verteilt ist, einen Pool mit Servlet-Exemplaren verwalten.

Für Servlets, die nicht das *SingleThreadModel* Interface implementieren und deren *service* Methode (oder Methoden wie *doGet* oder *doPost*, die zur *service* Methode der abstrakten Klasse *HttpServlet* weitergeleitet werden) mit dem Schlüsselwort *synchronized* definiert wurde, kann der Servlet-Container keinen Pool verwenden, daher müssen alle Anfragen zu dieser Methode serialisiert werden. Dies hat eine negative Wirkung auf die Performanz des Systems.

Ausnahmen (Exceptions) während der Bearbeitung von Anfragen (Requests)

Ein Servlet kann während der Bearbeitung einer Anfrage eine *ServletException* oder eine *UnavailableException* erzeugen. Eine *ServletException* zeigt, dass bei der Bearbeitung der Anfrage ein Fehler aufgetreten ist und dass der Container eine angemessene Maßnahme zur Erledigung der Anfrage einzuleiten hat. Eine *UnavailableException* zeigt, dass ein Servlet temporär oder permanent nicht in der Lage ist, Anfragen zu bearbeiten.

Wenn von der *UnavailableException* eine permanente Nichtverfügbarkeit angezeigt wird, muss der Servlet-Container den Dienst des Servlets entfernen, indem er die *destroy* Methode des Servlets aufruft und das Servlet-Exemplar auflöst.

Wenn von der *UnavailableException* eine temporäre Nichtverfügbarkeit angezeigt wird, kann der Container, für die vorübergehende Zeit der Nichtverfügbarkeit, das Routing der Anfragen zum Servlet unterlassen oder fortführen. Der Container muss, als Antwort auf jede abgewiesene Anfrage, eine *SERVICE_UNAVAILABLE (503)* Statusantwort zusammen mit einem *Retry-After Header* zurückgeben, der anzeigt, wann die Nichtverfügbarkeit beendet sein wird.

Der Container kann die Unterscheidung zwischen permanenter und temporärer Nichtverfügbarkeit ignorieren und damit alle *UnavailableExceptions* als permanent betrachten. Dies hat zur Folge, dass er alle Servlets entfernt, die eine solche Exception erzeugen.

Thread-Sicherheit

Die Implementationen der Request- und Response-Objekte sind nicht garantiert threadsicher. Das bedeutet, dass sie nur im Bereich des anfragebehandelnden Threads genutzt werden sollten. Referenzen zu den Request- und Response-Objekten sollten nicht an Objekte gegeben werden, die in anderen Threads ausgeführt werden, denn das daraus resultierende Verhalten ist nicht mehr deterministisch.

Ende des Dienstes

Die Spezifikationen definieren keine Zeitperiode, die definiert, wie lang ein Container ein Servlet geladen halten muss. Soll ein Servlet entladen werden, wird vom Container die *destroy* Methode des *Servlet* Interfaces gerufen. Das Servlet sollte dann jede beanspruchte Ressource freigeben. Es erhält so auch Gelegenheit, seinen Zustand persistent zu speichern.

Bevor der Servlet-Container die *destroy* Methode ruft, muss er jeden Thread, der in der *service* Methode des Servlets läuft, erlauben, seine Ausführung zu beenden, oder der Thread überschreitet ein vom Server definiertes Zeitlimit. Ist die *destroy* Methode erst mal gerufen, darf der Container keine weiteren Anfragen auf dieses Exemplar des Servlets routen. Sollte der Container das Servlet wieder benötigen, muss er ein neues Exemplar aus der Klasse des Servlets bilden. Nachdem die *destroy* Methode beendet wurde, muss der Servlet-Container das Servlet-Exemplar freigeben, damit es für die Garbage Collection verfügbar wird.

2.9.1.3 JSP Definition

Eine Java Server Page (JSP) ist, nach der JSP-Spezifikation v1.2 von SUN [JSPS12], ein textbasiertes Dokument, das beschreibt, wie Anfragen (Request) abzuarbeiten und Antworten (Response) zu erzeugen sind. Diese Beschreibung mischt statische Daten einer Vorlage mit dynamischen Aktionen. Die JSP Technologie unterstützt viele verschiedene Paradigmen zur dynamischen Erstellung von Inhalten.

Die Haupt-Features von Java Server Pages sind:

- Standarddirektiven (standard directives)
- Standardaktionen (standard actions)
- Scriptingelemente (scripting elements)
- ein Erweiterungsmechanismus für Tags (tag extension mechanism)
- Vorlagen für Inhalte (template content)

Die JSP-Spezifikation erbt von der Servlet-Spezifikation die Konzepte von *Web Applications, ServletContexts, Sessions, Requests* und *Responses*.

JSP-Seiten und Servlet-Klassen werden beide Webkomponenten (Web Components) genannt. Sie werden an einen Container geliefert, der die Dienste bietet, die im *JSP Component Contract* gefordert werden. Die Trennung der Komponenten vom Container erlaubt die Wiederverwendung der Komponenten und Quality-of-Service Features, welche vom Container zur Verfügung gestellt werden.

JSP-Seiten sind textliche Komponenten. Sie durchlaufen zwei Phasen: eine *Translation-(Übersetzungs-)* Phase und eine *Request (Anfrage)*-Phase. Die Übersetzungsphase findet pro Seite nur einmal statt. Die Request-Phase findet hingegen bei jeder Anfrage statt. Ein erzeugtes JSP Objekt behandelt Anfragen und generiert Antworten. JSP-Seiten können auch vor der ersten Nutzung übersetzt werden. Durch die Übersetzung entsteht ein Servlet, welches die JSP-Seite repräsentiert. Die Übersetzung kann auch durch den Web-Container zur Zeit des Deployments oder On-Demand, wenn eine Anfrage eine nicht übersetzte JSP-Seite anfordert, erfolgen. Bis auf ein Paar Ausnahmen ist die Integration von JSP-Seiten in die J2EE 1.3 Plattform wie bei Servlets.

2.9.1.4 JSP Lebenszyklus

Den Lebenszyklus kann man mit dem von Servlets vergleichen, da jede JSP-Seite in ihrer Übersetzungsphase in ein Servlet übersetzt wird. Dies wird in Abbildung 5 dargestellt. Ist die JSP-Seite erst mal übersetzt, so verhält sie sich, bis auf wenige Ausnahmen, wie ein Servlet (siehe Servlet Lebenszyklus). Die Ausnahmen sind in der JSP-Spezifikation ausführlich beschrieben.

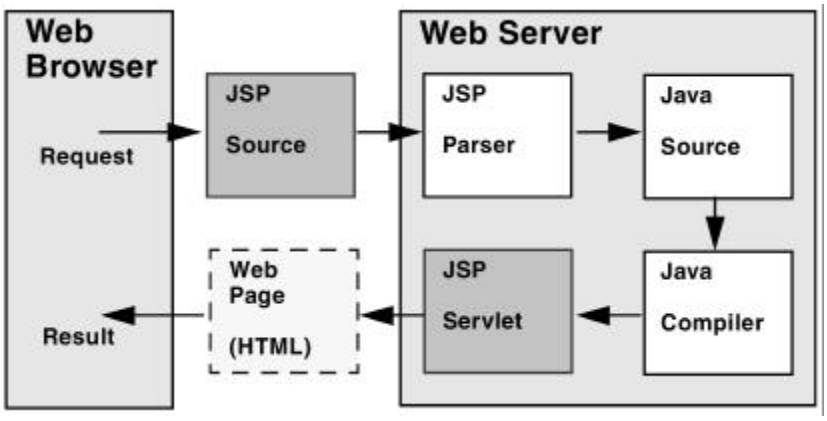

Abbildung 5: JSP Lebenszyklus aus [SG245755]

2.9.1.5 Filter Definition

Filter sind ein neues Konzept, das erstmals in der Servlet-Spezifikation v2.3 von SUN [SERVLETS23] definiert wurde. Nach der Spezifikation ist ein Filter eine wiederverwendbare Komponente, die Inhalte von HTTP-Requests (Anfragen), HTTP-Responses (Antworten) und deren Header-Informationen transformieren kann. Wobei es sich um statische oder dynamische Inhalte handeln kann. Im Allgemeinen generieren Filter keine Anfragen oder Antworten auf Anfragen, so wie es Servlets tun, vielmehr modifizieren oder adaptieren sie Anfragen für eine Ressource, bzw. modifizieren oder adaptieren Antworten einer Ressource.

Der Application Developer erzeugt einen Filter, indem er das *javax.servlet.Filter* Interface implementiert und einen öffentlichen Konstruktor ohne Argumente anbietet. Diese Klasse wird zusammen mit den statischen Inhalten und den Servlets in ein Web-Archiv verpackt. Ein

Filter wird im Deployment Descriptor, mit Hilfe des *filter* Elements, deklariert. Ein Filter oder eine Sammlung von Filtern kann für den Aufruf konfiguriert werden, indem im Deployment Descriptor *filter-mapping* Elemente definiert werden. Dies geschieht durch die Abbildung des Filters auf den logischen Namen eines speziellen Servlets oder auf ein URL-Muster, welches eine Gruppe von Servlets und Ressourcen mit statischen Inhalten kennzeichnet.

Funktionen von Filterkomponenten:

- Zugriff auf eine Ressource, bevor eine sie betreffende Anfrage bearbeitet wird.
- Abfangen (Interception) und Bearbeiten einer Anfrage für eine Ressource, bevor die Anfrage durch die Ressource bearbeitet wird.
- Modifikation des Request-Headers und der Request-Daten, durch Wrapping (Umhüllen) der Anfrage zu einer angepassten Versionen des Request-Objekts.
- Abfangen (Interception) und Bearbeiten einer Antwort einer Ressource, nachdem sie eine Anfrage bearbeitet hat.
- Modifikation des Response-Headers und der Response-Daten, durch die Erzeugung einer angepassten Versionen des Response-Objekts.
- Aktionen auf einem Servlet, einer Gruppe von Servlets oder statischen Inhalten, durch keinen, einen oder mehreren Filter in einer spezifizierbaren Reihenfolge.

Beispiele für Filterkomponenten :

- Authentisierungs-Filter
- Logging- und Auditing-Filter
- Bildumwandlungs-Filter
- Datenkompressions-Filter
- Encryption-Filter
- Tokenizing-Filter
- Filter, die beim Zugriff auf Ressourcen Events (Ereignisse) triggern (auslösen)
- XSL/T Filter, die XML-Inhalte transformieren können
- MIME-Typen Kettenfilter
- Caching-Filter

2.9.1.6 Filter Lebenszyklus

Der Lebenszyklus und die möglichen Ausnahmen einer Filterkomponente ähneln denen von Servlets. Nach dem Deployment der Webapplikation und noch bevor der Container, aufgrund einer Anfrage, auf eine Ressource zugreift, muss der Container eine Liste von Filtern lokali-

sieren, die dann der Webressource zugeordnet werden muss. Der Container muss sicherstellen, dass er für jeden Filter der Liste ein passendes Exemplar der Klasse erzeugt und dessen *init(FilterConfig config)* Methode gerufen hat. Es wird pro *filter* Deklaration im Deployment Descriptor nur ein Exemplar pro Java Virtual Machine des Containers erzeugt. Zur Konfiguration des Filters stellt der Container das Konfigurationsobjekt *config*, eine Referenz auf *ServletContext* und verschiedene Initialisierungsparameter zur Verfügung.

Ein Filter kann eine Exception erzeugen, um eine Fehlfunktion anzuzeigen. Ist diese Exception vom Typ *UnavailableException*, kann der Container ihr *isPermanent* Datenelement untersuchen und eventuell später nochmals auf den Filter zugreifen.

Bei einer eingehenden Anfrage nimmt der Container das erste Filter-Exemplar der Liste und ruft dessen *doFilter*Methode, wobei er die *ServletRequest* und *ServletResponse* Objekte und eine Referenz auf das *FilterChain* Objekt übergibt. Die *doFilter* Methode des Filters wird üblicherweise nach folgendem Muster, oder eine Untermenge von diesem, implementiert.

Schritt 1: Die Methode untersucht den Request-Header.

Schritt 2: Um den Request-Header oder die Daten zu modifizieren, kann die Methode das Request-Objekt mit einer angepassten Implementation von *ServletRequest* oder *HttpServletRequest* wrappen (umhüllen).

Schritt 3: Sie kann das Response-Objekt mit einer angepassten Implementation von *ServletResponse* oder *HttpServletResponse* wrappen (umhüllen) und den Response-Header oder die Daten modifizieren.

Schritt 4: Der Filter kann die nächste Entität in der Filter-Kette aufrufen. Die nächste Entität kann ein anderer Filter oder eine Web-Ressource sein, wenn der Filter der letzte der Kette war, die im Deployment Descriptor definiert wurde. Der Aufruf der nächsten Entität geschieht durch das Rufen der *doFilter* Methode des *FilterChain* Objekts und der Übergabe der Request- und Response-Objekte oder der gewrappten (umhüllten) Versionen dieser. Die Implementation der *doFilter* Methode des *FilterChain* Objekts, das vom Container zur Verfügung gestellt wird, muss die nächste Entität in der Filter-Kette lokalisieren, dort die *doFilter* Methode rufen und die passenden Request- und Response-Objekte übergeben. Alternativ kann die Filter-Kette die Anfrage blockieren und nicht die nächste Entität aufrufen, wobei der aktuelle Filter für das Ausfüllen des Response-Objekts verantwortlich wird.

Schritt 5: Nach dem Aufruf des nächsten Filters in der Kette, kann der Filter den Response-Header untersuchen.

Schritt 6: Alternativ kann der Filter eine Exception erzeugen, um einen Fehler bei der Verarbeitung anzuzeigen. Wenn der Filter, in der *doFilter* Methode eine *UnavailableException* erzeugt, muss der Container nicht mehr die ganze Kette von Filtern abarbeiten. Wenn die Exception nicht permanent ist, kann der Container sich dazu entscheiden, die ganze Kette zu einem späteren Zeitpunkt zu wiederholen.

Wenn der letzte Filter der Kette erreicht ist, ist die nächste Entität, auf die zugegriffen wird, das Ziel Servlet oder eine andere Ressource am Ende der Kette.

Bevor das Filter-Exemplar vom Container entfernt werden kann, muss der Container erst die *destroy* Methode des Filters rufen, um dem Filter zu ermöglichen, seine Ressourcen freizugeben und weitere Aufräumoperationen durchzuführen.

2.9.2 Enterprise JavaBeans

Die Thematik der Enterprise JavaBeans ist sehr umfangreich. Eine vollständige Darstellung würde den Rahmen dieser Arbeit sprengen, dennoch soll in diesem Kapitel ein grober Überblick gegeben werden. Dem interessierten Leser empfehle ich die Bücher: „Enterprise JavaBeans“ von S. Denninger und „Mastering EJB“ von Ed Roman.

Enterprise JavaBeans werden in der „Enterprise JavaBeans (TM) Specification v2.0“ von SUN vollständig spezifiziert. Neu in dieser Spezifikation sind u.a. die Konzepte von *Message-Driven-Beans* und die Interfaces für den lokalen Zugriff auf EJBs.

Enterprise JavaBeans (EJB) Komponenten werden in einer kontrollierten Umgebung ausgeführt, die Transaktionen unterstützt. EJBs enthalten üblicherweise die Businesslogik der Applikation und können beim Deployment durch Änderungen im Deployment Descriptor angepasst werden.

Eine weitere Definition findet sich in dem Buch „Enterprise JavaBeans“, von Richard Monson-Haefel (O'Reilly 1999):

“Enterprise JavaBeans is a standard server-side component model for component-based distributed transaction monitors.“

Alle EJBs besitzen folgende Bestandteile:

- Ein *Home-Interface*, in welchem alle Methoden definiert sind, die den Lebenszyklus der Bean betreffen. Es wird von Clients genutzt, um Bean-Exemplare zu erzeugen, zu finden und um sie zu löschen. Ab EJB 2.0 wird zwischen einer leichtgewichtigen Version des zu implementierenden Interfaces zur lokalen Kommunikation, mit dem Namen *javax.ejb.EJBLocalHome* (Local-Home-Interface genannt), und der klassischen, schwergewichtigen Version für entfernte (remote) Kommunikation, mit dem Namen *javax.ejb.EJBHome* (Remote-Home-Interface genannt), unterschieden.

- Ein *Component-Interface* (in EJB 1.1 Remote-Interface genannt) definiert die Methoden, die von der Bean nach außen hin angeboten werden. Ab EJB 2.0 wird zwischen einer leichtgewichtigen Version, des zu implementierenden Interfaces, zur lokalen Kommunikation, mit dem Namen *javax.ejb.EJBLocalObject* (Local-Interface genannt), und der klassischen, schwergewichtigen Version für entfernte (remote) Kommunikation, mit dem Namen *javax.ejb.EJBObject* (Remote-Interface genannt) unterschieden.

- Die *Bean-Klasse*, welche die eigentliche EJB ausmacht und in der sämtliche Logik codiert ist. Sie implementiert alle Methoden, die im Home- und im Component-Interface definiert wurden (mit Ausnahme der *FindByPrimaryKey* Methode), ohne diese beiden Interfaces im Sinne des Schlüsselwortes *implements* tatsächlich einzubinden. Die Signaturen der Home- und Component-Interfaces müssen mit den entsprechenden Methoden der Bean-Klasse übereinstimmen. Die Bean-Klasse muss je nach Typ ein spezielles Interface (Entity, Session oder Message-Driven) implementieren. Die Bean wird nicht veröffentlicht und ist dem Client nicht direkt zugänglich. Es findet eine Trennung zwischen Schnittstelle und Implementierung statt.

- Alle EJBs haben eine *Identität*. Für Entity-Beans wird eine Primärschlüsselklasse definiert, da ihre Identität nach außen sichtbar ist. Bei Session-Beans ist dies irrelevant, da ihre Identität nach außen nicht sichtbar ist.

- Jede EJB hat einen *Deployment Descriptor*, über den sie angepasst werden kann.
- Die Bean und ihre Interfaces werden zusammen mit ihrem Deployment Descriptor in eine *ejb-jar Archivdatei* verpackt.
- Beans, die nicht nur lokale (andere Beans) als Clients haben, müssen zusätzlich folgende Elemente besitzen:
    - Ein *Metadata Interface*, welches von Remote Clients zum dynamischen Aufruf einer EJB genutzt wird, wenn die EJB dem Client zur Zeit seiner Kompilierung unbekannt war.
    - Ein *Handle*. Die Tatsache, dass das Remote-Objekt nicht serialisierbar ist, kann in bestimmten Anwendungsfällen eine starke Einschränkung darstellen. Das Objekt kann nicht in eine Datei geschrieben oder über eine Netzwerkstrecke versandt werden. Aus diesem Grund ist in jeden Component-(Remote-) Interface eine Methode *getHandle* definiert, die ein serialisierbares Objekt der Klasse *javax.ejb.Handle* zurückgibt. In jedem Home-Interface gibt es eine entsprechende Methode *getHomeHandle*, die ein Objekt der Klasse *javax.ejb.HomeHandle* zurückgibt. Die Handle-Objekte haben ihrerseits eine Methode *getEJBObject*, deren Rückgabewert das verbundene Remote-Objekt der Bean ist. Ein Handle kann somit als serialisierbarer Zeiger auf ein Component- (Remote-) oder Remote-Home-Objekt verstanden werden.

Clientseitige Objekte in einer verteilten Umgebung

Wenn das RMI-IIOP Protokoll oder ein ähnliches verteiltes Protokoll genutzt wird, kommuniziert der Remote-Client mit der Enterprise Bean über s.g. Stubs, welche die serverseitigen Objekte repräsentieren. Die Stubs implementieren die Remote-Home- und Component-(Remote-) Interfaces. Abbildung 6 zeigt die beteiligten EJB-Objekte.

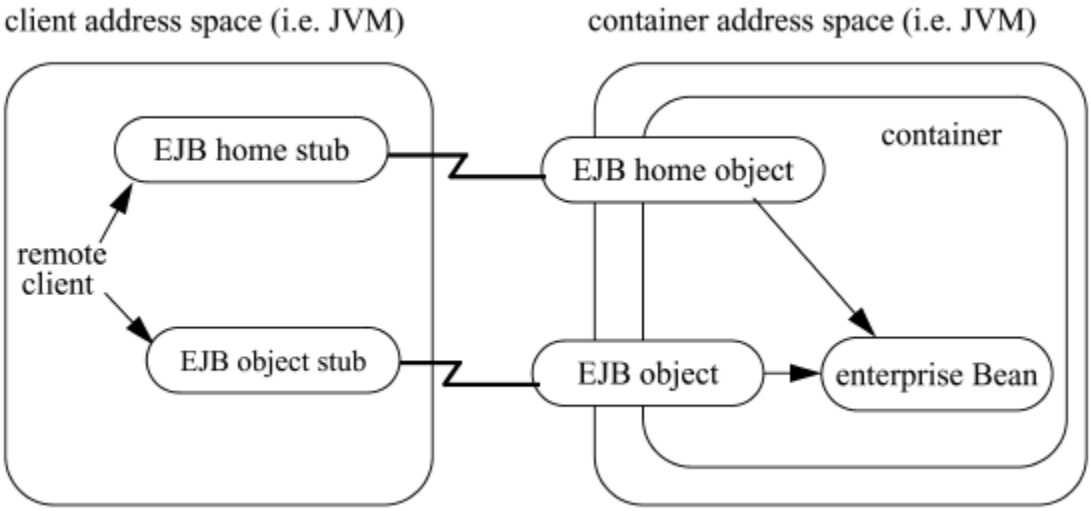

Abbildung 6: EJB-Objekte in einer verteilten Umgebung aus [EJBS13]

Restriktionen von Enterprise JavaBeans

Alle EJBs unterliegen, nach der EJB2.0-Spezifikation, folgenden Restriktionen:

- Eine Enterprise Bean darf keine *statischen Variablen* (Klassenvariablen) lesen oder schreiben. Statische Konstanten sind dagegen erlaubt. Die Spezifikation empfiehlt daher alle statischen (*static*) Variablen als *final* (= Konstante) zu deklarieren.

- Die Verwendung von *this* als Argument in einem Methodenaufruf oder als Rückgabewert für einen Methodenaufruf ist verboten. Anstatt von *this* muss die EJB die Ergebnisse von *SessionContext.getEJBObject(), SessionContext.getEJBLocalObject(), EntityContext.get-EJBObject()* oder *EntityContext.getEJBLocalObject()* verwenden.

- Eine Enterprise Bean darf keine Klassen in einem Paket definieren.

- Eine Enterprise Bean darf nicht versuchen, Zugriff auf Pakete oder Klassen zu erlangen, welche der Bean, durch die üblichen Regeln der Programmiersprache Java, nicht zur Verfügung gestellt werden.

- Eine Enterprise Bean darf keine Threads verwalten. Sie darf keinen Thread starten, stoppen, suspendieren oder fortsetzen; oder die Priorität oder den Namen des Threads ändern. Auch die Verwaltung von Thread-Gruppen ist verboten.

- Eine Enterprise Bean darf keine Primitive zur *Thread-Synchronisation* nutzen, um die Ausführung mehrerer Exemplare zu synchronisieren.

- Eine Enterprise Bean darf keine *AWT (Abstract Windowing Toolkit)* Funktionalitäten benutzen, um Ausgaben auf einem Bildschirm zu schreiben oder Eingabedaten von der Tastatur zu lesen.

- Eine Enterprise Bean darf keine Klassen des *java.io* Pakets benutzen, um auf Dateien und Verzeichnisse des Dateisystems zuzugreifen.

- Eine Enterprise Bean darf keine nativen Bibliotheken laden.

- Eine Enterprise Bean darf keinen *Class Loader* erzeugen, benutzen oder den Context des Class Loaders verändern. Eine EJB darf keinen Security Manager setzen oder erzeugen. Sie darf nicht die JVM stoppen und auch nicht die Standardeingabe, Standardausgabe oder den Standardfehlerstream verändern.

- Eine Enterprise Bean darf nicht direkt Dateideskriptoren lesen oder schreiben.

- Eine Enterprise Bean darf nicht auf *Sicherheitskonfigurationsobjekte* (Policy, Security Provider, Signer, Identity) zugreifen oder diese ändern.

- Eine Enterprise Bean darf keine Informationen über die *Security Policy* einer bestimmten Code-Quelle erhalten.

- Eine Enterprise Bean darf nicht auf *Netzwerk-Sockets* horchen, Verbindungen auf Netzwerk-Sockets akzeptieren und darf keine Sockets für Multicasts benutzen.

- Eine Enterprise Bean darf keine *Socket Factory* setzen, die von den Klassen *ServerSocket*, *Socket* oder der *Stream Handler Factory* der Klasse *URL* benutzt wird.

- Eine Enterprise Bean darf keine Unterklassen- oder Objektsubstitutions-Features des *Java Serialization Protocol* benutzen.

- Eine Enterprise Bean darf keine Klasse befragen, um an Informationen über die innere Struktur dieser Klasse zu gelangen. Die Enterprise Bean darf auch nicht das *Reflection API* benutzen, um an Informationen zu gelangen, die ihr, wegen den Sicherheitsregeln der Programmiersprache Java, nicht zur Verfügung gestellt werden.

Typen von Enterprise JavaBeans

Das nachfolgende Diagramm zeigt alle EJB-Typen:

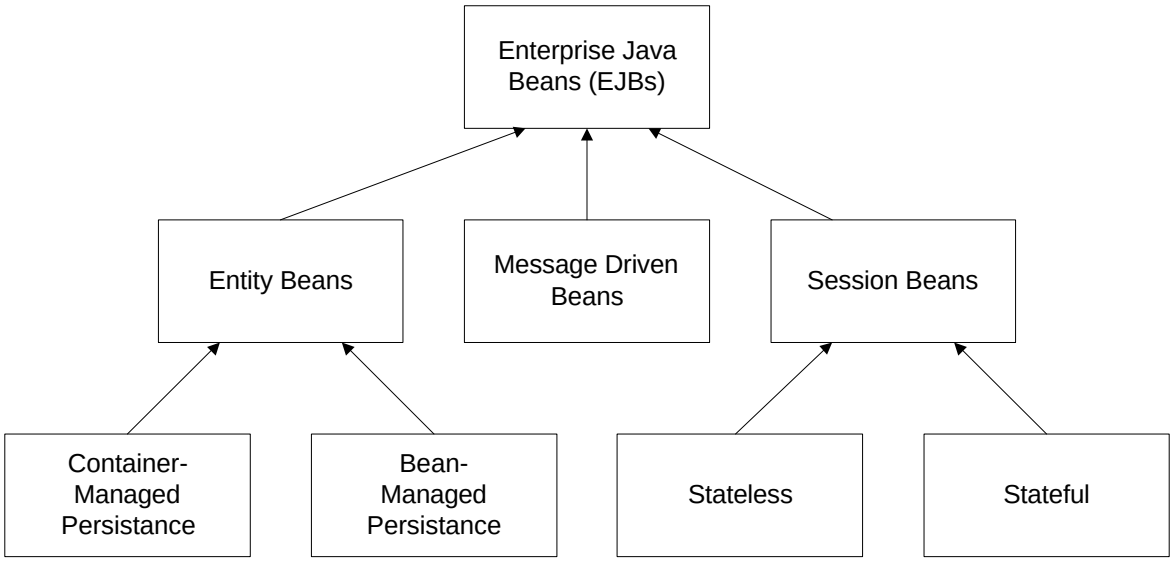

Abbildung 7: Enterprise Bean Typen

In Denninger´s „Enterprise JavaBeans“ findet sich folgende Differenzierung der Typen:

Merkmal	**Session-Bean**	**Entity-Bean**
Aufgabe der Bean	Repräsentiert einen serverseitigen Dienst, der Aufgaben für einen Client durchführt.	Repräsentiert ein Geschäftsobjekt, dessen Daten sich in einem dauerhaften Speicher befinden.
Zugriff auf die Bean	Eine Session-Bean ist für den Client eine private Ressource. Sie steht ihm exklusiv zur Verfügung.	Die Entity-Bean ist eine zentrale Ressource; das Bean-Exemplar wird von mehreren Clients gleichzeitig benutzt, und ihre Daten stehen allen Clients zur Verfügung.
Persistenz der Bean	Nicht persistent; wenn der verbundene Client oder der Server terminiert werden, ist die Bean nicht mehr verfügbar.	Persistent; wenn die verbundenen Clients oder der Server terminiert werden, befindet sich der Zustand der Entity-Bean auf einem persistenten Speichermedium; die Bean kann so zu einem späteren Zeitpunkt wiederhergestellt werden.

Die einzelnen Typen werden im folgenden detaillierter beschrieben.

2.9.2.1 Session-Bean Definition

Ein typisches Session-Objekt hat folgende Eigenschaften:

- Seine Aktionen werden zugunsten eines Clients durchgeführt.
- Es kann Transaktionen berücksichtigen.
- Es kann gemeinsam benutzte Daten in einer darunter liegenden Datenbank aktualisieren.
- Es repräsentiert nicht direkt die gemeinsam genutzten Daten in der Datenbank, obwohl es auf solche Daten zugreifen und sie aktualisieren kann.
- Es ist relativ kurzlebig.
- Es hat je nach Typ einen Zustand oder ist zustandslos.
- Wenn der Container abstürzt, wird es entfernt. Der Container muss, um seine Berechnungen fortzuführen, wieder neue Session-Objekte etablieren.

Conversational-State

Die EJB-Spezifikation definiert den *Conversational-State* als den Zustand aller Datenelemente eines Session-Bean-Exemplars einschließlich der transitiven Hülle über alle Objekte, die über Java-Referenzen von dem Session-Bean-Exemplar aus erreichbar sind. Der Conversational-State bleibt nur für die Dauer einer Sitzung eines Clients erhalten und steht exklusiv nur diesem einen Client zur Verfügung. Aus diesem Grund wird die Session-Bean auch als logische, serverseitige Erweiterung des Clients verstanden.

Aktivierung und Passivierung

Wenn ein Session-Bean-Exemplar einen Zustand für genau einen Client speichert, dann wird auch für jeden Client ein eigenes Bean-Exemplar benötigt. Damit im Server die Anzahl der Exemplare von zustandsbehafteten Session-Beans nicht beliebig wächst, verwendet der EJB-Container eine Strategie zur zeitweisen Auslagerung der Exemplare aus dem Arbeitsspeicher. Die Auslagerung wird *Passivierung* genannt, das Zurückladen *Aktivierung*. Bei der Passivierung schreibt der EJB-Container den Zustand einer zustandsbehafteten Session-Bean auf ein sekundäres Speichermedium (z.B. auf eine Festplatte). Dafür wird die *Serialisierung* von Java-Objekten verwendet. Mit Hilfe der Serialisierung (und der gegenteiligen *Deserialisierung*) kann ein EJB-Container mit einer nach oben hin begrenzten Anzahl von Session-Bean-Exemplaren arbeiten und die verfügbaren Ressourcen schonen. Dieser Mechanismus birgt jedoch ein Problem: Nicht alle Java-Objekte können einfach serialisiert werden. Die Java-Sprachdefinition verlangt von allen Java-Klassen, deren Objekte serialisierbar sein sollen, die Implementation des *java.io.Serializable* Interfaces.

Daher definiert die EJB-Spezifikation genau, welche Bedingungen die Objekte des Conversational-States einer Session-Bean erfüllen müssen:

- Alle Objekte müssen serialisierbar sein.

- Folgende Referenzen sind erlaubt:
    - Referenz auf *NULL*
    - Referenz auf ein *Component-(Remote-)Interface*
    - Referenz auf ein *Home-Interface*
    - Referenz auf den *SessionContext*
    - Referenz auf den *JNDI Namensdienst* der Bean Umgebung
    - Referenz auf das Interface *UserTransaction*

Eine Session-Bean wird vor ihrer Auslagerung auf ein sekundäres Speichermedium immer durch den Aufruf der Methode *ejbPassivate* benachrichtigt. Das Session-Bean-Exemplar muss dann die oben genannten Bedingungen herstellen. Wird das Session-Bean-Exemplar wieder im Arbeitsspeicher benötigt, so wird es durch den Aufruf ihrer Methode *ejbActivate* benachrichtigt. Wodurch es beispielsweise Datenbankverbindungen wieder aufbauen kann.

In Denninger´s „Enterprise JavaBeans“ findet sich die folgende Differenzierung der Typen:

zustandslose Session-Beans	**zustandsbehaftete Session-Beans**
hat keinen Conversational-State	hat Conversational-State
leichtgewichtig	schwergewichtig
wird nicht aktiviert und passiviert	wird aktiviert und passiviert
Exemplar steht dem Client für die Dauer eines Methodenaufrufs zur Verfügung	Exemplar steht dem Client für die Dauer einer Sitzung zur Verfügung

2.9.2.2 Zustandslose Session-Bean Definition

Zustandslose Session-Beans (Stateless Session-Beans) sind Session-Beans, deren Exemplare keinen Conversational-State besitzen. Dies bedeutet, dass alle Bean-Exemplare äquivalent sind, solange sie sich nicht in der Bedienung einer vom Client aufgerufenen Methode befinden. Der Begriff „*zustandslos*“ besagt nur, dass das Exemplar keinen Zustand für einen speziellen Client hat. Jedoch können die Exemplarvariablen ihren Zustand über mehrere Aufrufe von Clients behalten. Beispiele hierfür wären offene Datenbankverbindungen oder eine Objektreferenz auf ein EJB Objekt.

2.9.2.3 Zustandslose Session-Bean Lebenszyklus

Die folgenden Schritte beschreiben den Lebenszyklus eines zustandslosen Session-Bean-Exemplars:

- Das Leben einer Session-Bean beginnt, wenn der Container die *newInstance()* Methode der Session-Bean-Klasse ruft, um ein neues Exemplar zu erzeugen. Dann ruft der Container am Exemplar die Methoden *setSessionContext()*, gefolgt von *ejbCreate()*. Der Container kann die Erzeugung neuer Exemplare jeder Zeit durchführen. Sie steht in keinen Zusammenhang mit dem Aufruf der *create()* Methode durch einen Client.

- Das Session-Bean-Exemplar ist nun bereit, delegierte Aufrufe von Businessmethoden von jedem Client anzunehmen.

- Wenn der Container das Exemplar nicht mehr länger benötigt (gewöhnlicher Weise, wenn der Container die Anzahl der Exemplare im *Method-Ready Pool* reduzieren will), ruft der Container an dem Bean-Exemplar die Methode *ejbRemove()*. Dies beendet das Leben eines zustandslosen Session-Bean-Exemplars.

Abbildung 8 zeigt alle Zustände und Übergänge im Lebenszyklus einer zustandslosen Session-Bean.

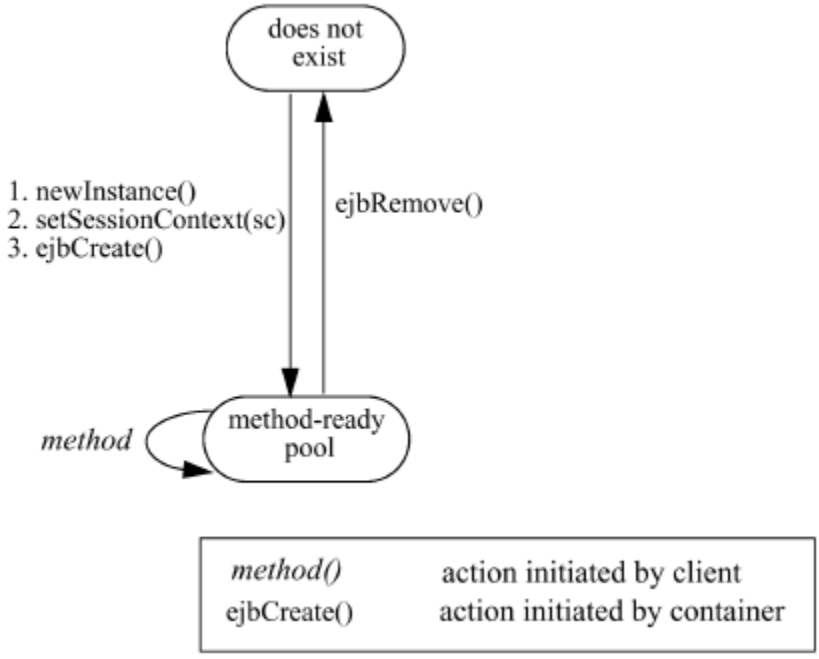

Abbildung 8: Zustandslose Session-Bean Lebenszyklus [EJBS20]

2.9.2.4 Zustandsbehaftete Session-Bean Definition

Nach der EJB-Spezifikation ist eine *Zustandsbehaftete Session-Bean (Stateful Session-Bean)* eine serverseitige Erweiterung des Clients, der sie erzeugt hat.

- Die Datenelemente des Exemplars enthalten den Conversational-State zugunsten des Clients des Session Objekts. Dieser Zustand beschreibt die Konversation durch ein spezifisches Client/Session Objektpaar.

- Üblicherweise liest und ändert das Exemplar Daten in einer Datenbank im Namen des Clients. In einer Transaktion können einige dieser Daten im Exemplar zwischengespeichert werden.

- Die Lebenszeit des Exemplars wird durch den Client kontrolliert.

Auch der Container kann das Leben eines Session-Bean-Exemplars nach einem, vom Deployer definierten, Timeout oder bei einem Fehlerereignis auf dem Server, auf dem das Bean-Exemplar läuft, beenden. Aus diesem Grund sollte der Client darauf vorbereitet sein, ein neues Session Objekt zu erzeugen, falls er sein bisheriges verliert.

2.9.2.5 Zustandsbehaftete Session-Bean Lebenszyklus

Der wesentliche Unterschied zum Lebenszyklus von zustandlosen Session-Bean-Exemplaren ist die Unterstützung von Transaktionen und die Steuerung durch den Client. Auf eine präzise Erläuterung der Übergänge wird hier, aufgrund des Umfanges, verzichtet. In der EJB-Spezifikation findet sich eine genauere Darstellung. Es werden vier Zustände unterschieden:

- *Nicht existent*: Das Exemplar existiert nicht.
- *Method Ready*: Das Exemplar existiert und wurde einem Client zugeteilt. Es steht für Methodenaufrufe zur Verfügung. Das Exemplar kann vom EJB-Container in diesem Zustand jederzeit passiviert werden.
- *Method Ready in Transaction*: Das Exemplar befindet sich in einer Transaktion und wartet auf Methodenaufrufe des Clients. Der EJB-Container darf Exemplare in Transaktionen nicht passivieren.
- *Passiviert*: Das Exemplar wurde zeitweilig aus dem Arbeitsspeicher ausgelagert, ist aber noch einem Client zugeteilt. Wenn der Client eine Methode ausführen möchte, muss der EJB-Container das Exemplar zunächst wieder aktivieren.

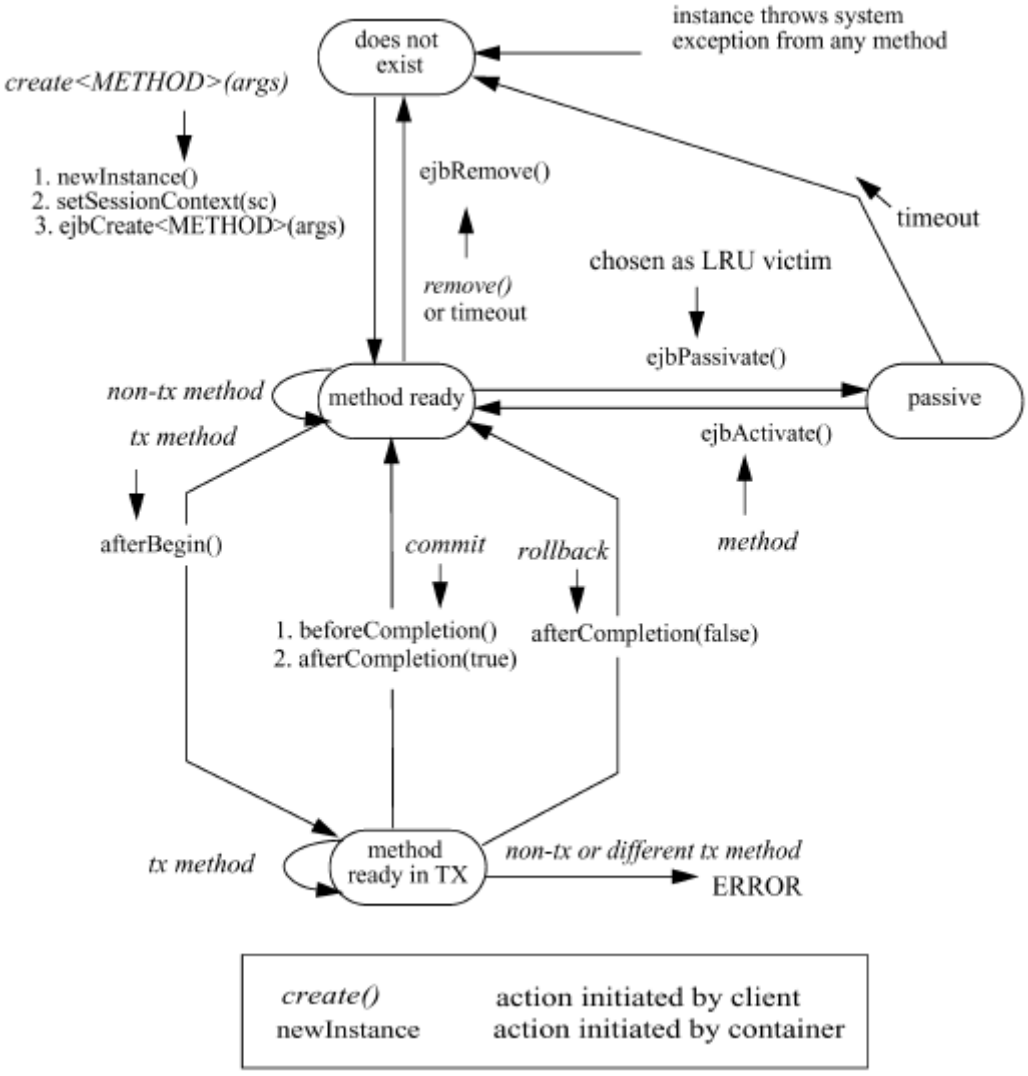

Abbildung 9: Zustandsbehaftete Session-Bean Lebenszyklus aus [EJBS20]

2.9.2.6 Entity-Bean Definition

Ein typisches Entity-Objekt hat folgende Eigenschaften:

- Es bietet eine Objektsicht der Daten in einer Datenbank.
- Es erlaubt einen gemeinsamen Zugriff mehrerer Benutzer.
- Es kann langlebig sein. (Es lebt so lange wie die Daten in der Datenbank.)
- Die Entität, der Primary-Key und seine Remote-Referenzen überleben einen möglichen Absturz des EJB-Containers. Wurde zum Zeitpunkt des Absturzes der Zustand der Entität in einer Transaktion geändert, so wird der Zustand automatisch zu dem des letzten *Commits* zurückgesetzt. Der Absturz ist für den Client nicht voll transparent, er bekommt nur eine Exception, wenn er eine Entität ruft, die den Absturz erfahren hat.

Eine Entity-Bean hat *transiente* und *persistente* Datenelemente. Transiente Datenelemente gehen beim Löschen des Bean-Exemplars verloren, persistente Datenelemente werden in einem beliebigen Medium (z.B. einer Datenbank) gespeichert.

Der Persistenzmechanismus kann *container-managed* (CMP) sein, d.h. es ist Aufgabe des Containers, die Bean persistent zu machen. Dazu können Techniken wie z.B. Serialisierung genutzt werden. Ab EJB2.0 können auch Relationen zwischen Entitäten verwaltet werden. Implementiert die Bean den Persistenzmechanismus selbst, so spricht man von *bean-managed Persistence* (BMP).

2.9.2.7 Entity-Bean Lebenszyklus aus der Sicht des Containers

Ein Entity-Exemplar kann, vom Container gesteuert, folgende Zustände annehmen:

- *Nicht existent*: Das Exemplar existiert nicht.
- *Pooled*: Das Exemplar existiert, ihm wurde aber noch keine Bean-Identität zugewiesen. Das Exemplar wird verwendet, um Zugriffe des Clients auf das Home-Interface zu ermöglichen, da sich diese nicht auf eine spezielle Bean-Identität beziehen.
- *Ready*: Das Exemplar hat eine Bean-Identität und kann vom Client verwendet werden. Der Zustand *Ready* lässt sich in drei Unterzustände aufteilen:
    - *Ready-Async*: Die Datenelementwerte sind evtl. nicht mit dem aktuellen Datenbankinhalt abgeglichen. Entweder wurden die Datenelemente noch nicht initialisiert, oder der Datenbankinhalt wurde durch einen parallelen Zugriff geändert. Durch die Methode *ejbLoad()* werden die Daten der Bean mit denen aus der Datenbank aktualisiert und der Zustand wechselt auf *Ready-Sync*.
    - *Ready-Sync*: Die Datenelementwerte haben den aktuellen Inhalt.
    - *Ready-Update*: Die Datenelemente der Bean wurden vom Client geändert. Normalerweise befindet sich die Bean in einer Transaktion. Die neuen Werte wurden noch nicht oder nur teilweise in die Datenbank geschrieben. Durch die Methode *ejbStore()* werden die Daten in die Datenbank geschrieben und der Zustand der Bean wechselt auf *Ready-Sync*.

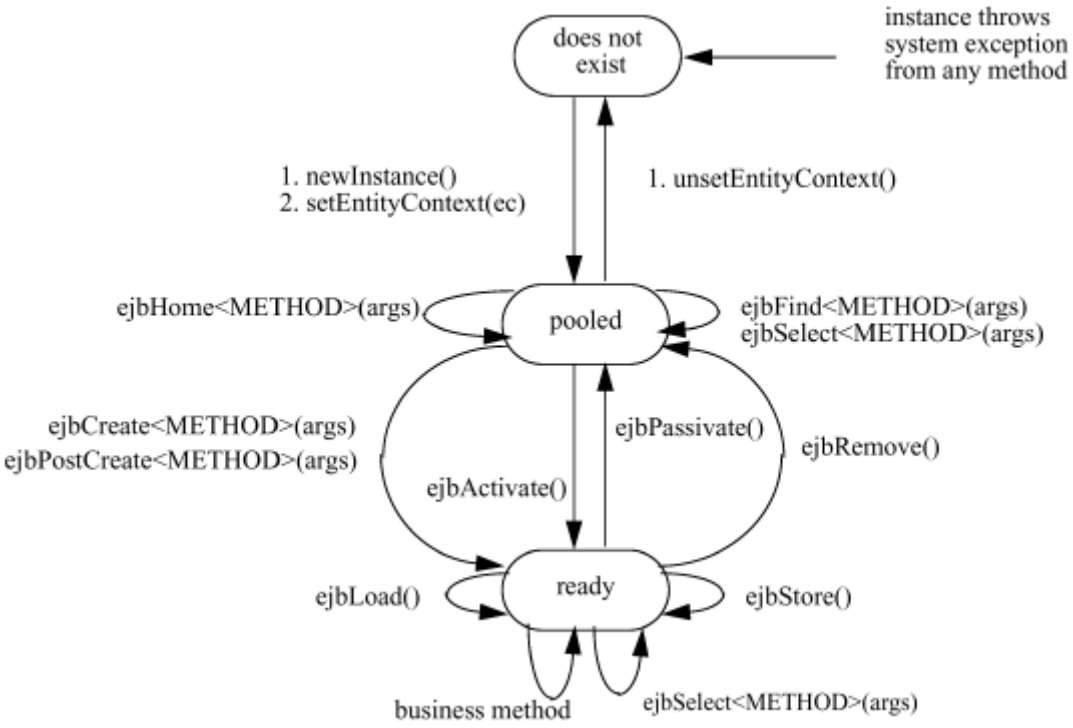

Abbildung 10: Entity-Bean Lebenszyklus aus [EJBS20]

2.9.2.8 Message-Driven-Bean Definition

Ein typisches *Message-Driven-Bean* (MDB) Objekt hat folgende Eigenschaften:

- Es wird nach dem Erhalt einer Client-Nachricht ausgeführt.
- Es wird asynchron aufgerufen.
- Es kann Transaktionen berücksichtigen.
- Es kann gemeinsam benutzte Daten in einer darunter liegenden Datenbank aktualisieren.
- Es repräsentiert nicht direkt die gemeinsam genutzten Daten in der Datenbank, obwohl es auf solche Daten zugreifen und sie aktualisieren kann.
- Es ist relativ kurzlebig.
- Es ist zustandslos.
- Wenn der Container abstürzt, wird es entfernt. Der Container muss, um seine Berechnungen fortzuführen, wieder neue MDB Objekte etablieren.

Eine MDB ist ein Konsument asynchroner Nachrichten. Bei der Ankunft einer JMS Nachricht wird sie vom Container aufgerufen. Sie besitzt weder Home- noch Component-Interface. Für den Client ist eine MDB ein Konsument von JMS Nachrichten, der irgendwo in der Businesslogik implementiert ist und auf dem Server läuft. Der Client greift auf die MDB mittels JMS zu, indem er Nachrichten zum JMS Bestimmungsort (Queue oder Topic) schickt, für den die MDB Klasse der *MessageListener* ist.

MDBs haben keinen Conversational-State, damit sind alle Bean-Exemplare äquivalent, solange sie keine Client-Nachrichten bedienen. MDBs sind anonym. Sie haben für den Client keine sichtbare Identität. Ein MDB-Exemplar wird vom Container erzeugt, um Nachrichten zu verarbeiten, für welche die Bean der Konsument ist. Die Lebenszeit wird vom Container kontrolliert. Das MDB-Exemplar hat für einen speziellen Client keinen Zustand. Jedoch können die Exemplarvariablen ihren Zustand über mehrere Aufrufe von Clients behalten. Beispiele für diese Zustände sind, wie bei den zustandslosen Session-Beans, offene Datenbankverbindungen und Objektreferenzen auf EJB-Objekte.

2.9.2.9 Message-Driven-Bean Lebenszyklus

Die folgenden Schritte beschreiben den Lebenszyklus eines MDB-Exemplars:

- Das Leben eines MDB-Exemplars beginnt, wenn der Container die Methode *newInstance()* der MDB-Klasse ruft, um ein neues Exemplar zu erzeugen. Dann ruft der Container am Exemplar die Methode *setMessageDrivenContext* gefolgt von *ejbCreate*.

- Das MDB-Exemplar ist nun bereit Nachrichten zu verarbeiten, die von einem Client an ihren Bestimmungsort geliefert werden.

- Wenn der Container das Exemplar nicht länger benötigt (was gewöhnlich der Fall ist, wenn der Container die Anzahl von Exemplaren im *Method-Ready-Pool* verringern will), ruft er am Exemplar die Methode *ejbRemove*. Dies beendet das Leben des MDB-Exemplars.

Abbildung 11 zeigt alle Zustände und Übergänge im Lebenszyklus einer MDB.

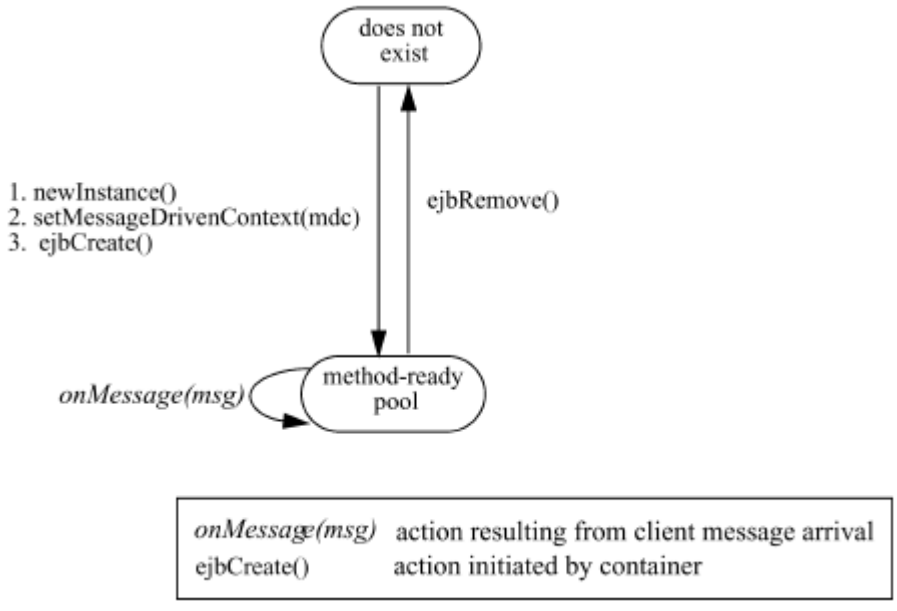

Abbildung 11: Message-Driven-Bean Lebenszyklus aus [EJBS20]

2.10 J2EE-Applikationen

J2EE-Applikationen sind aus einer oder mehreren *J2EE-Komponenten* und einem *J2EE Application Deployment Descriptor* zusammengesetzt. Der Deployment Descriptor beschreibt die Applikationskomponenten als Module. Ein J2EE Modul repräsentiert eine einfache Einheit in der Zusammensetzung einer J2EE-Applikation. J2EE Module bestehen aus einer oder mehreren Komponenten und einem Deployment Descriptor auf Modulebene. Die folgende Grafik soll diese Zusammenhänge verdeutlichen.

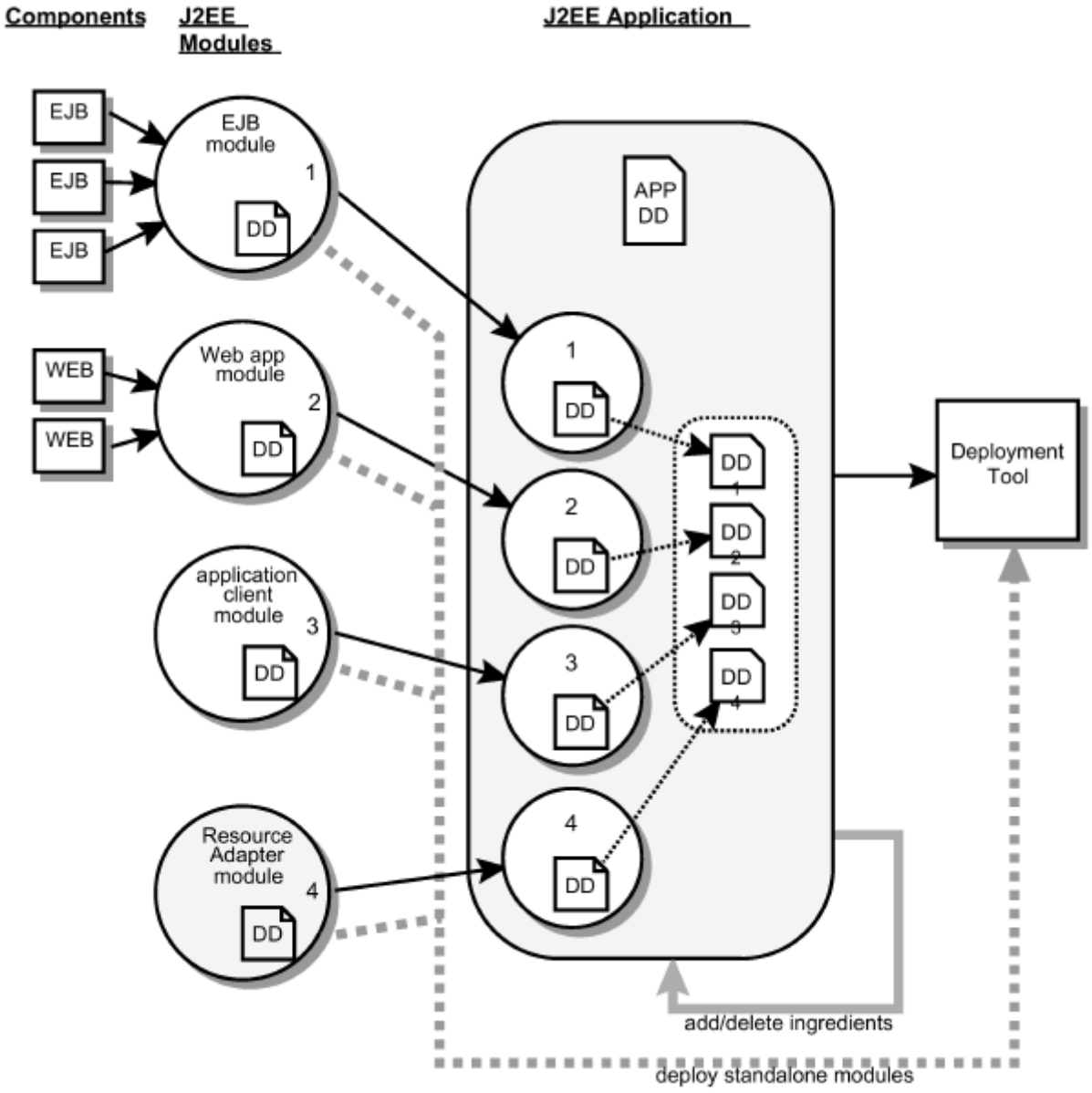

Abbildung 12: J2EE-Applikationsstruktur und Deployment aus [J2EES13]

2.11 Interoperabilität

In Unternehmen findet sich oft eine heterogene Systemlandschaft mit unterschiedlichen Hardware-Plattformen und Applikationen, die in unterschiedlichen Programmiersprachen geschrieben wurden. Mit der J2EE-Plattform ist es möglich, verschiedene Typen von Clients zu unterstützen und neue Dienste zu den existierenden Enterprise Information Systems (EISs) hinzuzufügen.

In diesen Unternehmensumgebungen kann die J2EE-Plattform zur Integration von Applikationen eingesetzt werden, die folgenden Typen entsprechen:

- Applikationen, die in Sprachen wie C++ und Visual Basic geschrieben wurden.
- Applikationen, die auf einer Personal Computer Plattform oder einer UNIX Workstation laufen.
- Stand-alone Java Technologie-basierende Applikationen, die nicht direkt von der J2EE-Plattform unterstützt werden.

Die Interoperabilitätsanforderungen der J2EE-Spezifikation erlauben:

- J2EE-Applikationen mit Altsysteme zu verbinden, wobei CORBA oder low-level Socket Schnittstellen genutzt werden.
- J2EE-Applikationen mit anderen J2EE-Applikationen, über mehrere verschiedene J2EE-Produkte, zu verbinden. Diese J2EE-Produkte können auf verschiedenen Plattformen laufen und von verschiedenen Product Providern stammen.

Die Anforderungen spezifizieren verschiedene Protokolle (Internet-Protokolle, OMG-Protokolle und Java Technologie Protokolle) sowie verschiedene Datenformate.

Internet-Protokolle

Die J2EE-Spezifikation fordert folgende Internet-Protokolle:

- TCP/IP-Protokollfamilie, sie ist die Kernkomponente zur Internet Kommunikation. TCP/IP und UDP/IP sind die Standardtransportprotokolle für das Internet. TCP/IP wird von J2SE und dem darunter liegenden Betriebssystem unterstützt.
- HTTP 1.0 ist die Kernkomponente zur Webkommunikation. Wie TCP/IP wird HTTP 1.0 von J2SE und dem darunter liegenden Betriebssystem unterstützt. Ein J2EE Web-Container muss fähig sein, seinen HTTP-Dienst auf dem Standard HTTP Port 80 anzubieten.
- SSL 3.0 und TLS[1] 1.0 (Secure Socket Layer) repräsentieren die Sicherheitsschicht der Webkommunikation. Dies schließt HTTPS (HTTP über SSL) mit ein. Ein J2EE-Web-Container muss in der Lage sein, sein HTTPS-Dienst über den HTTPS Standard Port 443 anzubieten. SSL 3.0 und TLS 1.0 werden auch für das EJB-Interoperability-Protokoll von der EJB-Spezifikation gefordert.

[1] TLS = Transport Layer Security

OMG-Protokolle

Die J2EE-Spezifikation fordert, dass die J2EE-Plattform folgende Protokolle der Object Management Group (OMG) unterstützt:

- IIOP (Internet Inter-ORB Protocol), wird durch Java IDL und RMI-IIOP von J2SE unterstützt. Java IDL bietet durch die Common Object Request Broker Architecture (CORBA) eine standardisierte Interoperabilität und Konnektivität. CORBA spezifiziert den Object Request Broker (ORB), mit dessen Hilfe Applikationen, unabhängig von ihren jeweiligen Ort, miteinander kommunizieren können. IIOP kann zusammen mit dem RMI-Protokoll, unter Verwendung der RMI-IIOP Technologie, genutzt werden. IIOP wird in den Kapiteln 13 und 15 der CORBA 2.3.1. Spezifikation [CORBA231] definiert.

- EJB Interoperability Protocol, basierend auf IIOP (GIOP 1.2) und der (draft) CSIv2 CORBA Secure Interoperability Specification, wird von der EJB-Spezifikation definiert.

- COSNaming, ist ein IIOP-basierendes Protokoll für den Zugriff auf Namensdienste. Es wird vom EJB Interoperability Protocol benötigt, um EJB-Objekte mit dem JNDI API zu finden. Zusätzlich muss es möglich sein, das JavaIDL COSNaming API für den Zugriff auf einen COSNaming Namensdienst zu verwenden. Alle J2EE-Produkte müssen einen COSNaming Namensdienst anbieten, der den Anforderungen der Interoperable Naming Service Specification [IONSS] genügt.

Java Technologie Protokolle

- JRMP (Java Remote Method Protocol); Die J2EE-Spezifikation fordert von jedem J2EE-Produkt, dass es das JRMP Protokoll unterstützt. Java RMI (Remote Method Invocation) [RMIS13] gehört zu J2SE und beinhaltet die Protokolle JRMP und IIOP. RMI ist ein verteiltes Objektmodell für die Programmiersprache Java. Verteilte Systeme laufen in verschiedenen Adressräumen und oft auch auf verschiedenen Hosts, müssen aber in der Lage sein, miteinander zu kommunizieren. JRMP erlaubt es Objekten auf Programmebene in verschiedenen Adressräumen, entfernte (Remote-) Objekte, mit der Semantik des Objektmodells der Programmiersprache Java, ähnlich wie lokale Objekte zu rufen.

Datenformate

Zusätzlich zu den Protokollen, die eine Kommunikation zwischen den Komponenten erlauben, fordert die J2EE-Spezifikation diverse Datenformate. Diese Formate bieten eine Definition der Daten, die zwischen den Komponenten ausgetauscht werden. Die folgenden Datenformate müssen unterstützt werden:

- HTML 3.2 repräsentiert den minimale Web Standard. Obwohl er nicht direkt von J2EE APIs unterstützt wird, müssen ihn J2EE Web-Clients anzeigen können.

- Bilddateiformate; Die J2EE-Plattform muss GIF und JPEG Bilder unterstützen. Unterstützung für diese Formate bieten das *java.awt.image* API[1] und die J2EE Web-Clients.

[1] http://java.SUN.com/j2se/1.3/docs/api/java/awt/image/package-summary.html

- JAR Dateien; (Java Archive) Dateien [JARS] sind das Standard Packaging Format für Java Applikationskomponenten, inklusive dem Enterprise Bean Archive (ejb-jar) spezialisierten Format, dem Web Application Archive (war) Format, dem Resource Adapter Archive (rar) und dem J2EE Enterprise Application Archive (ear) Format. JAR ist ein plattformunabhängiges Dateiformat, das es erlaubt, mehrere Java Komponenten in eine JAR-Datei zu bündeln und diese Datei in einer einzigen HTTP-Transaktion in den Browser zu laden. Die JAR Datei Formate werden von den Paketen *java.util.jar* und *java.util.zip* unterstützt.

- Class-Dateiformat; das Dateiformat von Klassen ist in der Java Virtual Machine Specification [JVMS2] spezifiziert. Jede Class-Datei enthält ein Typ, entweder Klasse oder Interface, der Programmiersprache Java und besteht aus einem Stream von 8-bit Bytes.

Viele der bisher beschriebenen APIs bieten Interoperabilität mit Komponenten, die nicht Teil der J2EE-Plattform sind, wie externe Web oder CORBA Dienste.

Abbildung 13 zeigt durch die Richtung der Pfeile die Client/Server Beziehung der Komponenten.

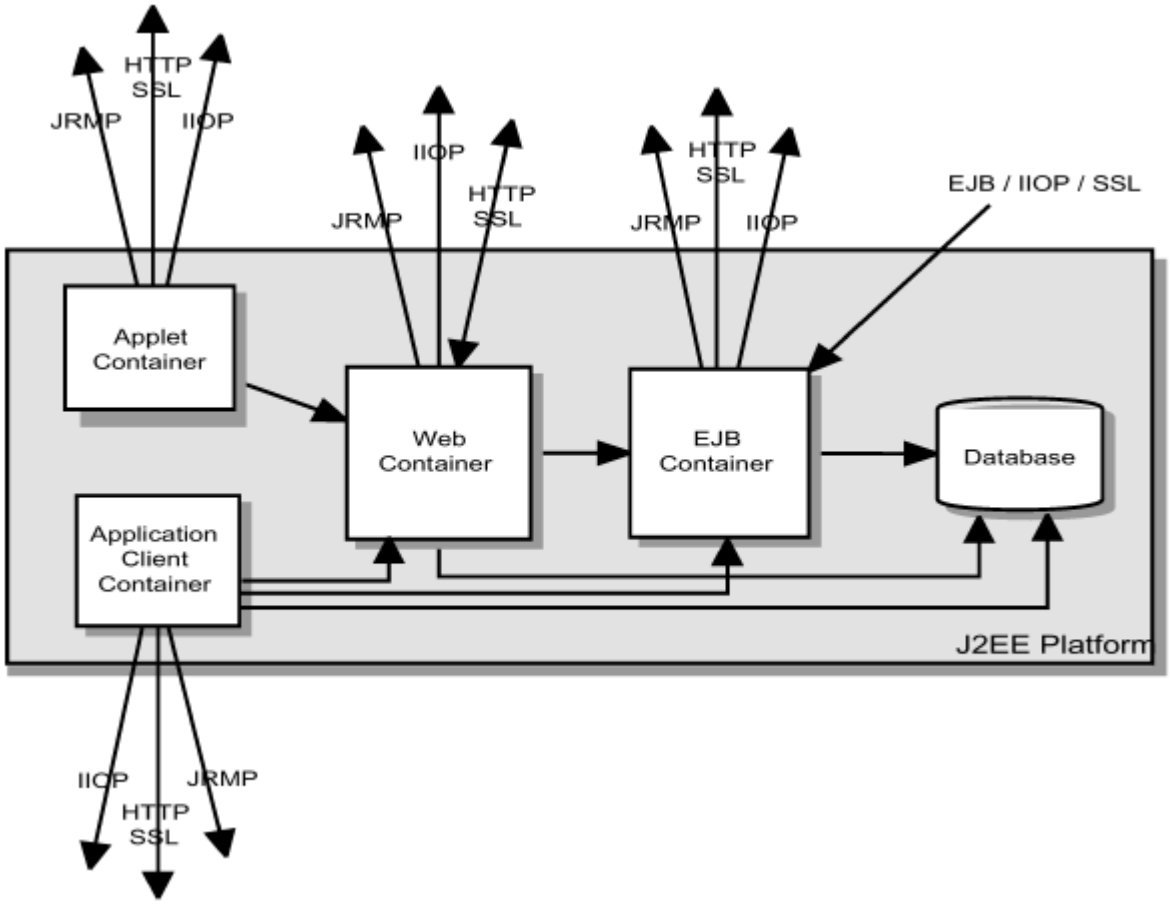

Abbildung 13: J2EE Interoperabilität aus [J2EES13]

2.12 Flexibilität der Produktanforderungen

Die Spezifikation verlangt nicht, dass jede J2EE-Produktanforderung von einem einzelnen Programm, einem einzelnen Server oder auch einer einzelnen Maschine implementiert wird. Allgemein kann man sagen, dass die Spezifikation nicht die Aufteilung der Dienste oder Funktionen zwischen verschiedenen Maschinen, Servern oder Prozessen, beschreibt. Solange die Anforderungen der Spezifikation erfüllt werden, können die J2EE Product Provider die Funktionalität aufteilen, wo immer sie wollen. Daher sind für J2EE-Produkte, welche die Anforderungen der Spezifikationen erfüllen, viele verschiedene Konfigurationen und Implementationen möglich.

Ein sehr einfaches J2EE-Produkt könnte, obwohl dies ein sehr extremes und möglicher Weise auch seltenes Beispiel ist, z.B. eine einfache Java Virtual Machine sein, die Applets, Web Komponenten und Enterprise Beans in einem Container und Applikation Clients in einem anderen Container verwaltet. Ein typisches low-end J2EE-Produkt würde einen einfachen Server bieten, der Web Komponenten und Enterprise Beans unterstützt. Ein high-end J2EE-Produkt hingegen, könnte Server Komponenten zur Lastverteilung auf mehrere Maschinen verteilen. Die Spezifikation schreibt diese Konfigurationen weder vor noch schließt sie diese aus.

2.13 J2EE-Produkt Erweiterungen

Die J2EE-Spezifikation beschreibt die Mindestmenge von Funktionalitäten, die ein J2EE-Produkt zur Verfügung stellen muss. Die meisten J2EE-Produkte bieten jedoch weit mehr Funktionalitäten als in der Spezifikation gefordert.

Die Spezifikation begrenzt nur in geringem Maße die Fähigkeit eines Produktes, Erweiterungen anzubieten, die über die geforderten Funktionalitäten hinausgehen. Insbesondere sind dies die gleichen Restriktionen, die bei J2SE für Erweiterungen der Java APIs gelten. Ein J2EE-Produkt darf keine Klassen zu Paketen hinzufügen, die zum Sprachumfang von Java gehören. Oder gar in den Klassen dieser Pakete die Signaturen der Methoden ändern oder weitere Methoden hinzufügen. Dennoch, viele andere Erweiterungen sind erlaubt. Ein J2EE-Produkt darf weitere Java APIs anbieten, seien dies weitere optionale oder andere (sachgerecht benannte) Pakete. Es kann auch weitere Protokolle oder Dienste unterstützen, die in der Spezifikation nicht erwähnt sind. Ferner kann es Applikationen unterstützen, die in anderen Programmiersprachen geschrieben sind und Anschlussmöglichkeiten für andere Plattformen oder Applikationen bieten.

Applikationen, die Funktionen nutzen, die nicht in der Spezifikation beschrieben werden, sind schlechter portierbar. Abhängig von den genutzten Funktionalitäten kann der Verlust nur gering oder aber auch signifikant sein. Das Dokument “Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition” [DEAJ2EE] zeigt zu diesem Problem mögliche Lösungswege auf.

SUN erwartet viele Varianten von J2EE-Produkten, welche sich in vielen Aspekten des Quality-of-Service unterscheiden. Diese Produkte werden verschiedene Grade von Performanz, Skalierbarkeit, Stabilität, Verfügbarkeit und Sicherheit aufweisen. Die Spezifikation fordert z.Zt. nur wenige dieser Eigenschaften. In Zukunft soll es Verfahren geben, die es Applikationen erlauben, ihre Anforderungen an diese QoS-Eigenschaften eines J2EE-Produktes zu definieren.

3 J2SE-Security

Im folgenden werden die Konzepte der Sicherheitsmechanismen von Java in der Standardversion dargestellt. Dieses Kapitel stammt ursprünglich aus der Studienarbeit [Luerssen 01] des Autors und wurde vollständig überarbeitet und erweitert.

3.1 Allgemeine Sicherheitsmerkmale von Java

3.1.1 Unberechtigter Speicherzugriff

Die Programmiersprache Java entstand aus einer Untermenge der Programmiersprachen C++ und Smalltalk (Bytecode und Garbage Collection) mit einer leicht veränderten Syntax. Um einen unberechtigten Speicherzugriff zu verhindern, wurde die Pointerarithmetik entfernt. Pointer gibt es weiterhin in Form von Referenzen, so dass Datenstrukturen, wie z.B. verkettete Listen und Bäume, generiert werden können. Die Spezifikation von Java definiert den Umgang mit nicht initialisierten Variablen sehr genau. Jeder heap-basierte Speicher wird automatisch initialisiert, stack-basierter Speicher nicht. Lokale Variablen müssen initialisiert werden. Die korrekte Initialisierung wird während Kompilierung geprüft. Daher können Klassen- und Exemplarvariablen niemals undefinierte Werte annehmen.

3.1.2 Garbage Collection

Ein weiteres Feature von Java, das zur Sicherheit beiträgt, ist die *Garbage Collection*. Nicht mehr referenzierter Speicher wird zur Laufzeit vom *Garbage Collector* freigegeben. Daher muss ein Softwareentwickler nicht mehr zur Entwicklungszeit entscheiden, wann eine Speicherfreigabe sicher ist. So werden Fehler, die durch eine fehlerhafte Freigabe entstehen, vermieden.

3.1.3 Starke Typisierung und Zugriffsmodifier

Java ist *stark typisiert*, d.h. schon während der Kompilierung werden alle Typen geprüft und illegale Casts führen zu einer Fehlermeldung. Auch hierdurch wird der illegale Zugriff auf Speicherbereiche, die nicht zum Originalobjekt gehören, verhindert.

Zugriffsmodifier (*public, protected, friendly*[1], und *private*) regeln den Zugriff bzw. die Sichtbarkeit von Datenelementen, Methoden, Klassen, Interfaces und Pakete. In der Klassendefinition verhindert die Verwendung des Modifiers *final* ein weiteres Vererben der Klasse. Auf Methoden angewendet, wird ein Überschreiben verhindert. Mit *final* deklarierte Variable sind Konstanten. Es existiert auch eine Variante von *final*, das “blank final”. Hier kann ein Wert nur einmal im Konstruktor zugewiesen werden.

3.1.4 Sicherheit durch Offenheit

Im Gegensatz zu vielen anderen Programmiersprachen, sind Sicherheitsmodell, Konzepte und Implementierungsdetails von SUN offengelegt. Es liegt kein Black-Box Verhalten und kein “Security through obscurity” Konzept vor.

[1] *friendly* ist kein Schlüsselwort. Wird kein Schlüsselwort angegeben, wird *friendly* angenommen.

3.1.5 Auditing

Ein wichtiges Merkmal von sicheren Systemen ist das Auditing. In der aktuellen J2SE-Version 1.3 ist es noch nicht implementiert.

3.2 Die Java Virtual Machine (JVM)

Wenn von der Java Virtual Machine gesprochen wird, können drei Dinge damit gemeint sein:

1. Eine abstrakte Spezifikation.
2. Eine konkrete Implementation.
3. Ein Laufzeit-Exemplar.

Die abstrakte Spezifikation ist das Konzept der JVM. Es ist in der Java Virtual Machine Specification [JVMS2] definiert. Konkrete Implementationen existieren auf vielen Plattformen und können entweder eine komplette Software Implementation der abstrakten Spezifikation oder eine Kombination von Soft- und Hardware sein. Ein Laufzeit-Exemplar der konkreten Implementation beherbergt (hostet) eine einzelne laufende Java Applikation. Die Java Virtual Machine ist der essenzielle Teil der Java-Plattform, da er Lösungen für die Features von Java enthält, welche die Vorteile von Java gegenüber anderen Programmiersprachen ausmachen. Die JVM ist nicht nur verantwortlich für die Hardware- und Software-Unabhängigkeit von Java, sie schützt den Benutzer auch vor maliziösen Code. Die Sicherheitsmechanismen beinhalten Überprüfungen des Codes auf vielen verschiedenen Ebenen.

Die JVM ist ein abstrakter Rechner, der ähnlich wie echte Rechner einen Befehlssatz besitzt und zur Laufzeit verschiedene Speicherbereiche manipulieren kann. Die JVM macht keine Vorannahmen bezüglich der Hardware oder des Betriebssystems. Auch im Bezug auf die Implementierungstechnologie gibt es keine Vorannahmen. Dies ermöglicht zum Beispiel eine Implementation der JVM auf einem einzigen Chip. Die Programmiersprache Java betrifft die JVM weniger, vielmehr das spezielle binäre Format der Class-Dateien, welche JVM-Instruktionen (Bytecode) und ergänzende Informationen enthalten. Um einen gewissen Grad an Sicherheit zu gewährleisten, hat die JVM starke Format- und Strukturbeschränkungen für den Byte- und Programmcode.

3.2.1 Der Lebenszyklus der Java Virtual Machine

Wenn eine Java-Applikation startet, wird ein Laufzeit-Exemplar der JVM erzeugt. Wenn die Applikation beendet wird, wird auch das Laufzeit-Exemplar der JVM beendet. Jede Java Applikation hat ihre eigene JVM. Die Java-Version der „Hello World“ Applikation wird im folgenden, als typisches erstes Programmbeispiel, beschrieben.

Die JVM startet die Ausführung der Applikation in der *main()* Methode, die als *public static* deklariert ist, *void* (nichts) zurückgibt und einen String als Parameter akzeptiert.

```
class HelloWorldApp{
        public static void main (String[] args)
        {
        System.out.println(“Hello World!”);
        }
}
```

Nachdem der Java Quelltext (Source Code) in einer Datei gespeichert wurde, deren korrekte Bezeichnung *ClassName.java* lautet, wird dieser kompiliert und der resultierende Bytecode in einer Datei, mit dem Namen *ClassName.class*, gespeichert. Dieser Bytecode kann dann von einer JVM ausgeführt werden. Quelltext wird mit dem Kommando *javac ClassName.java* kompiliert. Mit dem Kommando *java ClassName.class* wird die Applikation in der JVM gestartet, wobei weitere Argumente nach dem Klassennamen angegeben werden können.

Innerhalb der JVM gibt es zwei Typen von Threads: *daemon* und *non-daemon*. Der daemon Thread wird üblicherweise von der JVM selbst genutzt. Dieser könnte zum Beispiel ein Thread sein, der die Garbage Collection durchführt. Applikationen können Threads, die sie erzeugt haben, als daemon oder non-daemon kennzeichnen. Der erste Thread einer Applikation, der Thread der Methode *main(),* ist ein non-daemon Thread. Ein Laufzeit-Exemplar der JVM existiert so lange noch non-daemon Threads laufen. Wenn alle non-daemon Threads beendet sind, beendet die Java Applikation ihre Ausführung und zur gleichen Zeit terminiert das Laufzeit-Exemplar der JVM. In dem Beispiel *HelloWorldApp* generiert die *main* Methode keine weiteren Threads. Ist die main Methode durchlaufen, ist der einzige non-daemon Thread der Applikation beendet.

3.2.2 Die Architektur der Java Virtual Machine

Die Java Virtual Machine Specification [JVMS2] beschreibt die abstrakte innere Architektur einer abstrakten JVM bezüglich ihrer Subsysteme, Speicherbereiche, Datentypen und Instruktionen. Diese Komponenten beschreiben weniger die innere Architektur einer konkreten Implementation, vielmehr beschreiben sie genau das externe Verhalten einer Implementation. Die Definitionen sind also eher extensional gehalten. Abbildung 14 zeigt ein Blockdiagramm der JVM Architektur mit den wesentlichen Subsystemen und Speicherbereichen, wie sie in der Spezifikation beschrieben sind.

Jede JVM besitzt einen *Class Loader*, der für das Laden von Typen (Klassen und Interfaces) verantwortlich ist, sowie eine *Execution Engine*, welche für die Ausführung der, in den geladenen Klassen enthaltenen, Instruktionen verantwortlich ist. Wenn die JVM ein Programm ablaufen lässt, benötigt sie Speicher, um Bytecode, von dem Programm initiierte Objekte, Parameter, Rückgabewerte von Methoden, lokale Variable, Zwischenergebnisse von Berechnungen und andere aus der Klasse extrahierte Informationen zu speichern.

Die JVM organisiert den benötigten Speicher in mehrere Laufzeit-Daten Bereiche. Die Spezifikation dieser Speicherbereiche ist sehr abstrakt und gibt dem Designer die Freiheit, selbst die strukturellen Details der Implementation zu entscheiden. Einige dieser Speicherbereiche sind allen Threads zugänglich, andere Speicherbereiche stehen nur einem Thread der Applikation exklusiv zur Verfügung. Diese exklusiven Speicherbereiche werden erzeugt, wenn ein neuer Thread erzeugt wird. Sie werden zerstört, wenn der Thread beendet wird [Venners 99]. Jeder neue Thread erhält sein eigenes *PC (Program Counter) Register* und einen *Java Stack*. Ein Thread kann auf die PC Register und Java Stacks anderer Threads nicht zugreifen. Der Wert des PC Registers zeigt auf die nächste auszuführende Instruktion, wenn ein Thread eine Java Methode (aber keine native Methode) ausführt. Der Java Stack speichert den Zustand des Java Methodenaufrufs des Threads. Der Zustand eines Methodenaufrufs umfasst seine lokalen Variablen, die Parameter des Aufrufs, den möglichen Rückgabewert und Zwischenergebnisse der Berechnungen. Der Zustand nativer Methoden wird auf eine ähnliche Weise implementationsabhängig auf einem *Native Method Stack* gespeichert.

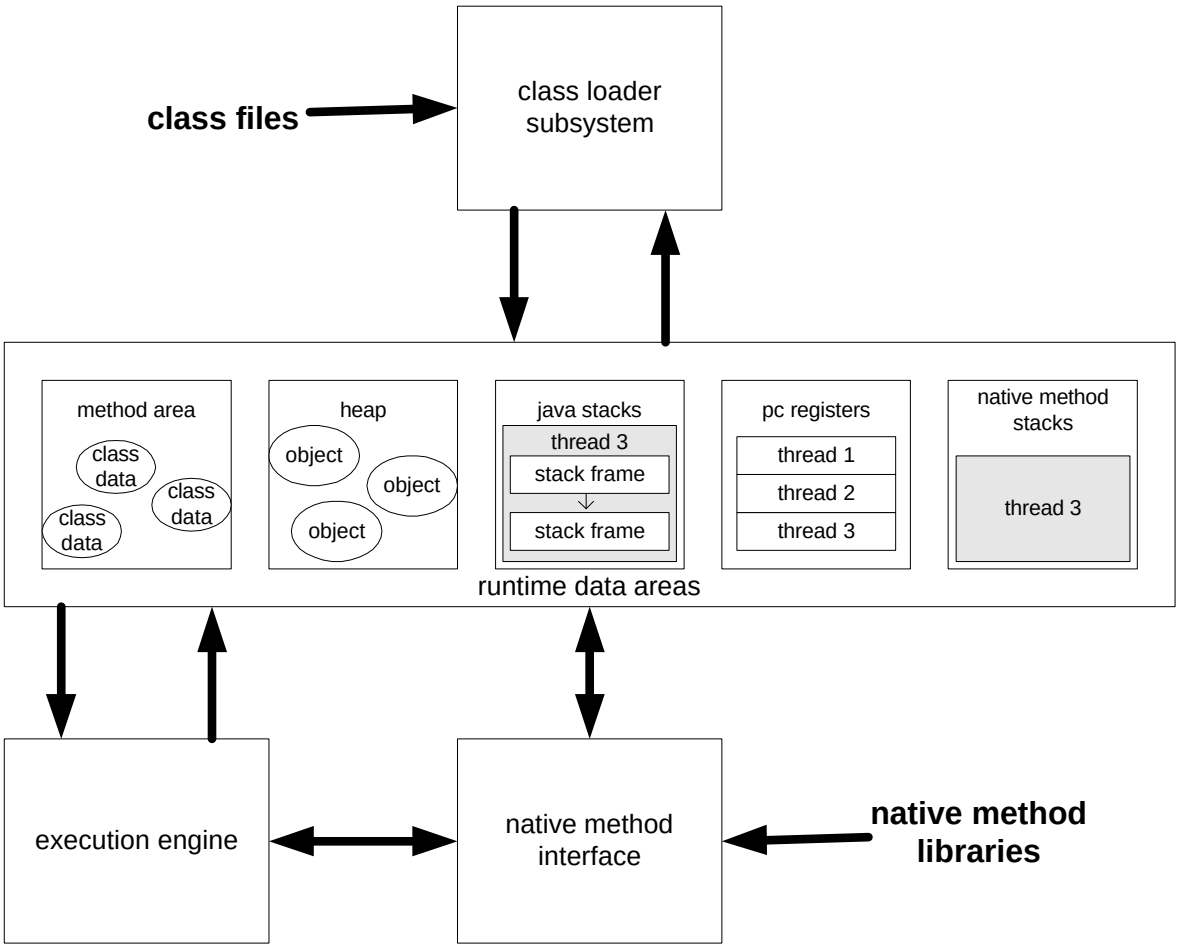

Abbildung 14: Innere Architektur der JVM

Jedes Laufzeit-Exemplar der JVM besitzt einen Methodenbereich (*Method Area*) und einen *Heap*. Alle in einer JVM laufenden Threads teilen sich diese Bereiche. Der Methodenbereich enthält Informationen, die sich aus dem Parsing der Class-Datei ergeben. Der Heap dient der Speicherung von Objekten, die vom Programm zur Laufzeit erzeugt werden. Der Java Stack umfasst *Stack Frames*, die den Zustand eines Java-Methodenaufrufs speichern. Die JVM packt (push) einen neuen Stack Frame auf den Stack, wenn sie eine Methode aufruft. Ist die Methode beendet, nimmt (pop) und verwirft die JVM den Stack Frame vom Stack. Die JVM selbst hat keine Register, jedoch nutzen ihre Instruktionen den Java Stack für Zwischenergebnisse. Der Grund für diesen Ansatz ist, dass die Menge von Instruktionen möglichst klein gehalten und auf jeder Rechnerarchitektur mit wenigen Registern einfach implementierbar sein soll. Es soll auch die Optimierung des Codes durch so genannte „just-in-time“ und dynamische Compiler erleichtert werden, die in einigen JVM Implementationen den Code zur Laufzeit optimieren.

3.3 Die Evolution des Sicherheitsmodells

Das ursprüngliche Sicherheitsmodell der Java Plattform (JDK[1] 1.0) war das “Sandbox” Modell, es sollte nicht vertrauenswürdigen Code (der z.B. aus dem Internet stammt) eine stark restriktive Umgebung bieten. Das Sandboxmodell (siehe Abbildung 15) gibt lokalen, vertrauenswürdigem Code vollen Zugriff auf die Systemressourcen und heruntergeladenen Code (z.B. ein Applet) stellt es nur die limitierte Sandbox-Umgebung zur Verfügung. Die Zugriffskontrolle wird durch den Security Manager durchgesetzt. Code, der in der Sandbox ausgeführt wird, darf:

- nicht von der lokale Festplatte lesen.
- nicht auf die lokale Festplatte schreiben.
- keine Netzwerkverbindungen aufbauen, außer zu dem Host, von dem es geladen wurde.
- keine neuen Prozesse erzeugen.
- keine neuen Bibliotheken dynamisch laden.

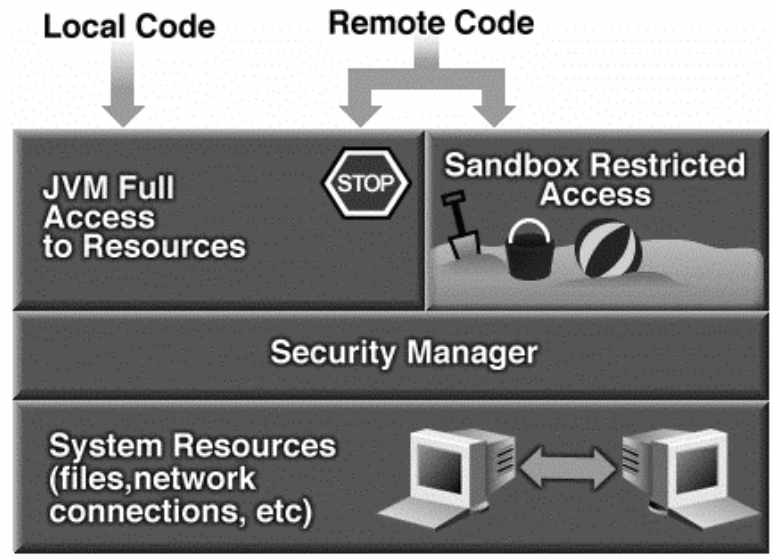

Abbildung 15: JDK 1.0 Security Model © SUN

JDK 1.1 führte das Prinzip der *signierten Applets* ein (siehe Abbildung 16). Ein digital signiertes Applet wird, wenn der Public-Key zum Verifizieren der Signatur vertrauenswürdig ist, wie lokaler Code behandelt, mit vollem Zugriff auf die Ressourcen. Applets ohne Signatur laufen weiter in der Sandbox ab. Signierte Applets werden mit ihren Signaturen in signierten JAR-Archiven ausgeliefert.

[1] JDK = Java Development Kit

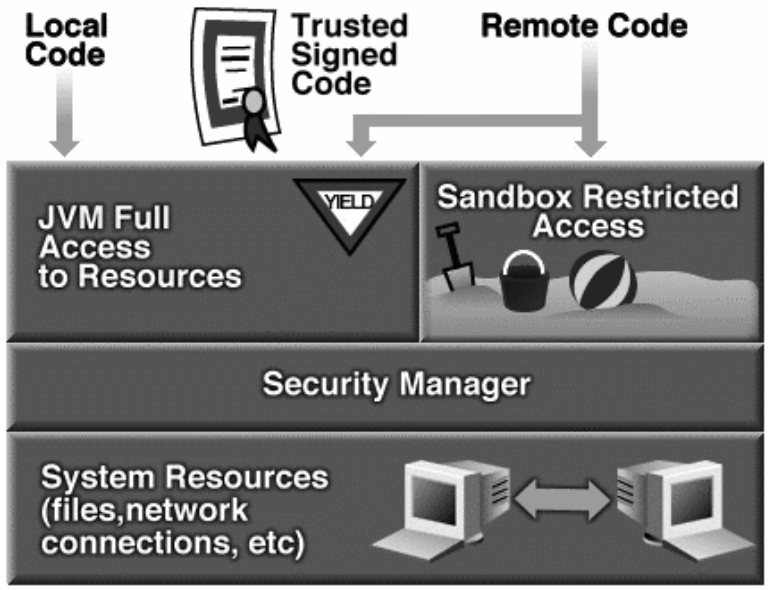

Abbildung 16: JDK 1.1 Security Model © SUN

JDK 1.2 bietet gegenüber seinen Vorgängern viele Verbesserungen. Jeder Code, egal ob lokal oder remote geladen, kann Gegenstand einer *Security Policy* sein. Eine *Security Policy* definiert Rechte (*Permissions*) für verfügbaren Code von unterschiedlichen Unterzeichnern (*Signer*s) und Herkunftsorten (*Codebases*). Unterzeichner und Herkunftsorte werden so genannten *CodeSources* zugeordnet, die eine Kombination aus einer *codeBase URL*, von welcher der Code geladen wurde, und einer Menge von Unterzeichnern, die für den Code garantieren, darstellen.

Die Konfiguration geschieht durch die *Security Policy* Datei, die eine Reihe von *Permission grant* Einträgen enthält. Sie wird von Benutzern und Administratoren konfiguriert und hat folgenden Syntax in :

```
grant [SignedBy "signer_names"] [, CodeBase "URL"]
     [, Principal [principal_class_name] "principal_name"]
     [, Principal [principal_class_name] "principal_name"] ... {
   permission permission_class_name [ "target_name" ]
            [, "action"] [, SignedBy "signer_names"];
   permission ...
};
```

Eine *Permission* repräsentiert das Zugriffsrecht auf eine Systemressource. Üblicherweise hat eine Permission ein Argument, das *Target*, welches die zu schützende Ressource beschreibt, und weitere *Action* Argumente, die spezielle erlaubte Aktionen kennzeichnen:

```
perm = new java.io.FilePermission("/tmp/abc", "read");
```

In diesem Fall wäre “/tmp/abc” das Target und “read” die Action.

Die Klasse *java.security.Permission* dient als Basisklasse für alle JDK 1.2 Klassen, die sich auf Permissions beziehen. Alle Permissions erben von dieser Basisklasse. Dies ist in Abbildung 17 dargestellt. Zusätzlich lassen sich eigene Permissions definieren.

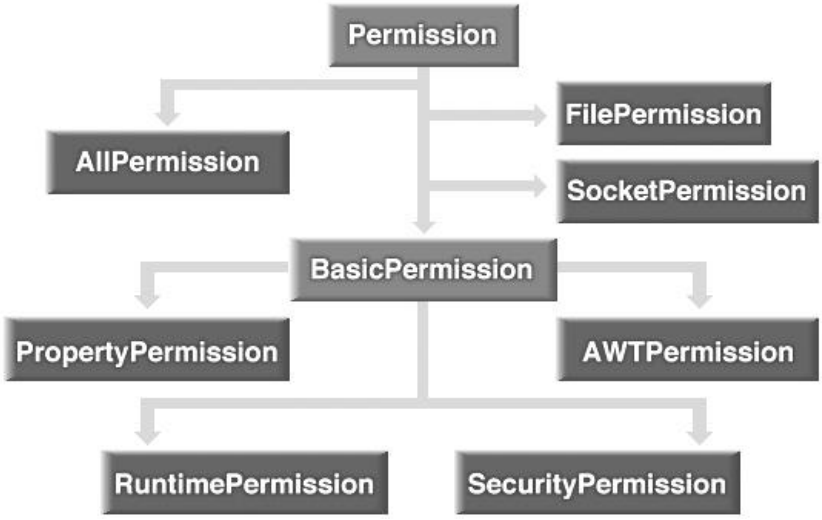

Abbildung 17: Permissions © SUN

Ein Eintrag in einer benutzerdefinierten Policy Datei könnte zum Beispiel so aussehen:

grant signedBy "Luerssen", codeBase "http://www.luerssen-consulting.de" {
permission java.io.FilePermission "c:\\temp\-", " write";
}

Hiermit würde Code, der von dem Benutzer „Luerssen" signiert wurde und von der Domäne „http://www.luerssen-consulting.de" geladen wurde, Schreibrechte auf das lokale temporäre Verzeichnis und dessen Unterverzeichnisse bekommen. Würde man anstelle des Minus (–) einen Stern (*) verwenden, würden die Rechte nur für das spezifische Verzeichnis gelten.

Wichtig zu bemerken ist, dass jede Klasse immer alle Dateien aus dem Verzeichnis ihrer *codeBase* lesen kann. Dieses Recht besitzt implizit jede Klasse!

Rechte werden zum Beispiel mit dem *Policy Tool* vergeben und durch dem *Access Controller* kontrolliert. Sind diese Rechte nicht vorhanden, wird eine *AccessControlException* ausgelöst, diese ist eine Unterklasse der mehr generischen *SecurityException*. Beide sind Ausnahmen zur Laufzeit.

Das folgende Beispiel zeigt, wie man überprüft, ob ein Benutzer Leserechte auf die Datei *test.out* im plattformspezifischen temporären Verzeichnis hat. Dazu benötigt der Benutzer keine Leserechte auf die *java.io.tmp.dir* Eigenschaft, welche per Grundeinstellung nicht verfügbar ist.

```
String tempPath = System.getProperty ("java.io.tmpdir");
File f = new File (tempPath, "test.out");
FilePermission perm =  new FilePermission(f.getAbsolutePath(), "read");
AccessController.checkPermission (perm);
```

Für Rechte, die vom System vergeben wurden, ist keine explizite manuelle Prüfung notwendig. Diese Prüfung erledigt das System. Bei selbst definierten Rechten ist eine explizite Prüfung notwendig. Um beispielsweise die Eigenschaft einer Bean zu lesen, muss der Prinzipal Leserechte für diese Eigenschaft besitzen. Zum Ändern dieser Eigenschaft benötigt er entsprechende Schreibrechte.

```
private String state;
...
public String getState() {
        MyPermission p = new MyPermission ("state", "read");
        AccessController.checkPermission (p);
        // Ohne Zugriffsrecht wird hier eine Exception erzeugt
        return state;
}

public void setState (String newValue) {
        MyPermission p = new MyPermission ("state", "write");
        AccessController.checkPermission (p);
        // Ohne Zugriffsrecht wird hier eine Exception erzeugt
        state = newValue;
}
```

Applikationen werden grundsätzlich ohne Restriktionen ausgeführt. Möchte man Applikationen mit einen Security Manager laufen lassen, kann man die Laufzeitumgebung mit der Option -D*java.security.manager* ausführen:

java -Djava.security.manager MyClass

Das Laufzeitsystem organisiert den Code in individuellen Domänen. Eine *Protection Domain* ist die Zusammenfassung der *CodeSources* und der *Permissions*, die der CodeSource in der Security Policy Datei gegeben (*grant*) wurde. Jede Klasse, die über einen *Class Loader* (siehe Kapitel 3.4 Class Loader) in die JVM geladen wird, wird einer Protection Domain entsprechend der Security Policy zugeordnet. Klassen mit der gleichen Codebase und den gleichen Unterzeichnern befinden sich in der gleichen Protection Domain. Klassen mit den gleichen Permission, aber mit verschiedenen CodeSources befinden sich in verschiedenen Protection Domains.

In der Abbildung 18 ist diese Architektur dargestellt. Der Code in den Domänen (Mitte) hat in dieser Abbildung mehr Privilegien als die Sandbox (Rechts) und weniger Privilegien als die lokalen Applikationen (Links).

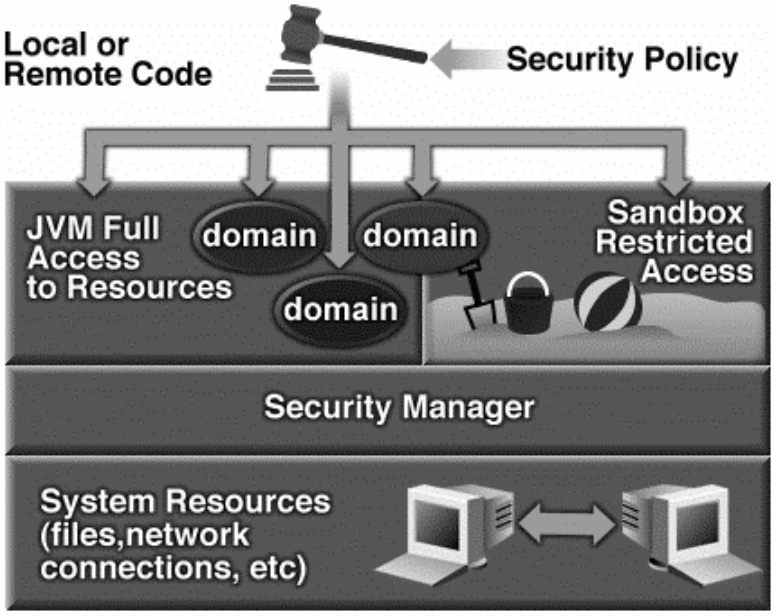

Abbildung 18: JDK 1.2 Security Model © SUN

3.4 Die Class Loader

Die Verantwortlichkeit für das Laden und die Verifizierung (mit Hilfe des *Class Verifiers*) der Klassen liegt beim *Class Loader*. Dieser findet und lädt den Bytecode zur Klassendefinition. Nach dem Laden wird der Bytecode zuerst verifiziert und anschließend die aktuelle Klasse erzeugt.

Es gibt zwei Class Loader Typen. Einen in die JVM integrierten („Bootstrap“) *Primordial Class Loader*, der die Java Basisklassen lädt (in Java 1.1 auch die Klassen im *CLASSPATH*). Von ihm existiert nur ein Exemplar und er ist in jeder JVM immer präsent. Er ist häufig in der gleichen Programmiersprache wie die JVM implementiert, die in der Regel in C implementiert wird. Klassen, die durch den Primordial Class Loader geladen wurden, werden als vertrauenswürdige („trusted“) Klassen bezeichnet und werden durch den Verifier nicht überprüft!

Des weiteren existieren Class Loader Objekte, von denen es mehrere Exemplare geben kann. Diese sind vollständig mit der Programmiersprache Java implementiert. Class Loader Objekte können andere Class Loader Objekte erzeugen, so dass eine Baumstruktur, mit der abstrakten Klasse *ClassLoader* in der Wurzel des Baumes, entsteht (siehe Abbildung 19). Neben der Vererbungshierarchie existiert eine Parent/Child-Beziehungsstruktur, die in Class Loader Objekten durch *parent* Referenzen definiert werden. Klassen, die von Class Loader Objekten geladen werden, werden als nichtvertrauenswürdig angesehen und werden daher vom Verifier überprüft.

Die Klasse *ClassLoader* ist eine abstrakte Klasse. Implementationen dieser Klasse sind z.B.: *SecureClassLoader*, *URLClassLoader*, *AppletClassLoader*, *AppClassLoader*, *ExtClassLoader* oder ein eigener benutzerdefinierter Class Loader.

Unter dem Gesichtspunkt der Sicherheit ist der Secure Class Loader der wichtigste, da er, bei jedem Ladevorgang, aus der *CodeBase* und den Unterzeichnern des Codes eine *CodeSource* bildet, und diese mit den dazugehörigen Permissions, aus der Security Policy, assoziiert. Der Secure Class Loader ordnet so jeder Klasse eine Protection Domain zu. Diese Funktionen haben natürlich auch alle Klassen, die von ihm erben. Vertrauenswürdige Klassen werden der System Domain und nichtvertrauenswürdige Klassen einer Application Domain zugeordnet.

Um zu bestimmen, wann welcher Class Loader eine Klasse lädt, lassen sich folgende Regeln anwenden:

- Der *Bootstrap (Primordial) Class Loader* lädt die Java Basisklassen, die unter dem Verzeichnis zu finden sind, das in der Systemeigenschaft *sun.boot.class.path* definiert wurde.

- Der *Extension Class Loader* (*sun.misc.Launcher$ExtClassLoader*) lädt die Klassen, die unter den Verzeichnissen zu finden sind, die in der Systemeigenschaft *sun.ext.dirs* definiert wurden.

- Der *System* (oder auch *Application*) *Class Loader* (*sun.misc.Launcher$AppClassLoader*) lädt die Klassen, die unter den Verzeichnissen zu finden sind, die in der Systemeigenschaft *java.class.path* (*CLASSPATH*) definiert wurden.

- Der *AppletClassLoader* lädt immer die erste Klasse eines Applets.

- Referenziert eine existierende Klasse eine andere Klasse, die noch zu laden ist, wird der Class Loader der existierenden Klasse verwendet.

Class Loader lassen sich in Applikationen explizit angeben, so dass man auch eigene Class Loader verwenden kann.

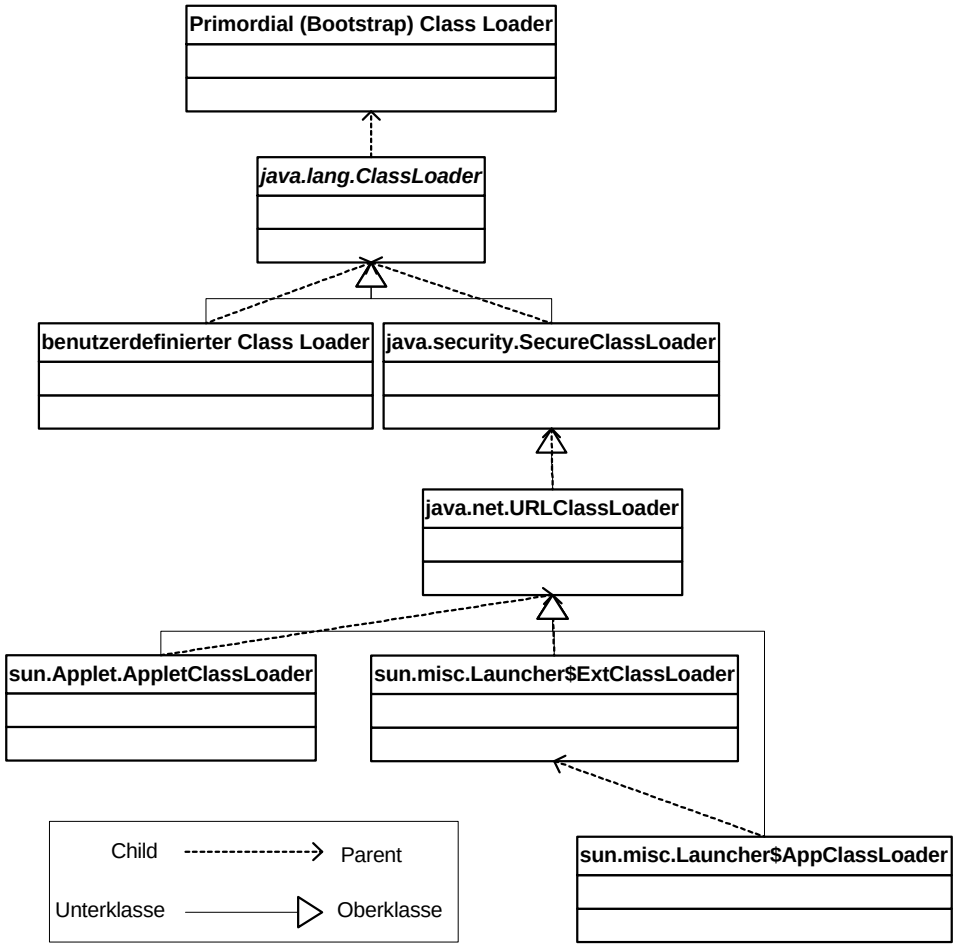

Abbildung 19: Class Loader Hierarchie

Die Class Loader Architektur unterstützt das Sicherheitsmodell von Java auf drei Ebenen:

1. Unterbindung von Interaktionen zwischen maliziösen Code und „normalen“ Code.

2. Schutz der Grenzen von vertrauenswürdigen Klassenbibliotheken.

3. Ordnen des Codes in Kategorien (*Protection Domains*), die entscheiden, welche Aktionen der Code ausführen darf.

Neben dem Finden, Laden und Verifizieren übt der Class Loader andere sicherheitsbezogene Pflichten aus:

- Erstens, der Class Loader unterbindet das Laden fremder *java.** Pakete über das Netzwerk. Dies stellt sicher, dass die JVM nicht ausgetrickst wird, indem falsche Repräsentationen von Java Standardklassen geladen werden, die das Sicherheitsmodell durchbrechen könnten. Dieser Angriff wird auch *Class Spoofing* genannt.

- Zweitens, der Class Loader bietet für jede Klasse, die von einem bestimmten Ort geladen wurde, einen extra Namensraum. So kommt es bei Klassen mit gleichen Namen, die von verschiedenen Rechnern geladen wurden, zu keinen Namenskonflikten. In der JVM können diese Klassen auch nicht miteinander kommunizieren. Die Informationen bleiben gekapselt, so dass nichtvertrauenswürdiger Code niemals Informationen von vertrauenswürdigen Code erhalten kann.

Die Class Loader Architektur nutzt verschiedene Namensräume (*Name Spaces*) für Klassen, die von verschiedenen Class Loadern geladen wurden, um vertrauenswürdigen Code vor der Interaktion mit maliziösen Code zu schützen. Ein Namensraum ist eine Menge von eindeutigen Namen, wobei jeder Name für eine Klasse steht. Hat ein Class Loader eine Klasse unter einem bestimmten Namen in einen Namensraum geladen, darf er keine weitere Klasse mit dem gleichen Namen in den Namensraum laden. Klassen mit dem gleichen Namen können in die JVM geladen werden, wenn dafür mehrere Namensräume, durch mehrere Class Loader, genutzt werden. Namensräume sind für die Sicherheit sehr wichtig. Denn sie bilden ein Schild zwischen verschiedenen Namensräumen und damit auch zwischen verschiedenen Class Loadern. Innerhalb der JVM können Klassen im gleichen Namensraum miteinander interagieren. Eine Klasse kann auf eine andere Klasse in einen anderen Namensraum nicht zugreifen. Im Methodenbereich (Method Area) der JVM ist jedem Namen eines Namensraumes ein Typ zugeordnet, der den Typ des Namens definiert. Vertrauenswürdige Pakete (Packages) können von einem anderen Class Loader geladen werden, als dem Class Loader der nichtvertrauenswürdigen Pakete, so schützt der Class Loader die Grenzen von vertrauenswürdigen Bibliotheken.

Java 1.2 nutzt zum Laden von Klassen das sogenannte *Parent Delegation Model*. In diesem Modell hat jeder Class Loader (außer dem Primordial Class Loader) eine *Parent Class Loader* Klasse, zu der ein Class Loader seine Aufgabe, eine Klasse zu laden, delegiert. Der Parent delegiert dann weiter zu seinem Parent. Dieser rekursive Prozess setzt sich fort, bis der Primordial Class Loader erreicht ist, welcher den letzten Class Loader einer Delegationskette darstellt. Kann er die Klasse laden, so gibt er die geladene Klasse zurück, anderenfalls versucht sein Child, die Klasse zu laden. Abbildung 19 zeigt das Parent/Child Delegationsmodell.

Diese Delegationskette ermöglicht der Class Loader Architektur, die vertrauenswürdigen Bibliotheken zu schützen, da der Primordial Class Loader immer vor den anderen Class Loadern versucht, die Klasse zu laden. Dies schützt vor *Class Spoofing*, bei dem maliziöser Code, mit dem Namen eines Typs des Java API, über eine Netzwerkverbindung geladen wird. Zum Beispiel ist es nicht möglich, eine Class-Datei mit dem Namen *java.lang.String* über eine Netzwerkverbindung zu laden, solange sie in der Java API existiert, da sie vom Primordial Class Loader geladen wird. Daher kann nichtvertrauenswürdiger Code niemals vertrauenswürdige Klassen durch eigene Versionen ersetzen.

Ein weiteres Angriffsszenario wäre, dass nichtvertrauenswürdiger Code eine neue Klasse zu einem vertrauenswürdigen Paket hinzufügt. Angenommen ein benutzerdefinierter Class Loader würde versuchen, eine Klasse mit dem Namen *java.lang.Attack* zu laden und zu definieren. Java erlaubt Klassen eines Pakets, sich gegenseitig gewisse Privilegien auszusprechen, die nur innerhalb des Paketes gültig sind. Da sich die neue Klasse *java.lang.Attack*, durch ihren Namen, als Teil des Java API deklariert, wäre die erste Annahme, dass sie die selben Privilegien, wie die der übrigen Klassen des *java.lang* Pakets, erhält. Das Konzept der Namensräume und unterschiedlichen Class Loader schützt auch hier. Der benutzerdefinierte Class Loader, der zum Herunterladen von *java.lang.Attack* genutzt wurde, hat seinen eigenen, vom Primordial Class Loader unabhängigen, Namensraum. Daher kann *java.lang.Attack* die

vertrauenswürdigen, vom Primordial Class Loader geladenen, *java.lang.** Klassen nicht sehen. Sie gehören nicht zum gleichen *Runtime Package,* das als die Menge von Klassen definiert ist, die von einem Class Loader geladen wurden.

3.5 Der Class Verifier

Die Aufgabe des Class Verifiers ist es sicherzustellen, dass die Class-Dateien die richtige innere Struktur besitzen und in einem konsistenten Zustand sind. Bei einer problematischen Class-Datei erzeugt der Class Verifier eine Ausnahme (Exception). Klassen werden nach dem Laden verifiziert, da die JVM nicht weiß, wie eine bestimmte Class-Datei generiert wurde. Als Konsequenz muss es eine Technik geben, mit der fehlerhafte oder maliziöse Class-Dateien erkannt werden.

Der Class Verifier hat einen großen Einfluss auf die Robustheit von Programmen, da nicht nur maliziöse Klassen, sondern auch Klassen, die von einem fehlerhaften Compiler generiert wurden, erkannt werden.

Der Class Verifier überprüft keine Klassen, die vom Primordial Class Loader geladen wurden.

3.5.1 Das Format von Class-Dateien

Eine Class-Datei ist ein einfacher Stream aus 8-bit Bytes. Alle 16-bit, 32-bit und 64-bit Werte werden in zwei, vier oder acht 8-bit Bytes umgeformt. Eine komplette Beschreibung findet sich in [JVMS2]. Folgende Elemente besitzt jede Class-Datei:

- Die ersten vier Bytes bilden die *Magicnumber: 0xCAFEBABE*
- Versionsinformationen
- Einen *Konstanten-Pool (constant pool)*
- Informationen über die Klasse (Name, Oberklasse, etc.)
- Informationen über die Datenelemente (*fields*) und Methoden der Klasse
- Debugging Informationen

Der *Konstanten-Pool* ist ein heterogener Array von Daten. Jeder Eintrag im Konstantenpool kann einer der folgenden sein:

- Ein Unicode [6] String
- Ein Klassen- oder Interfacename
- Ein Referenz auf ein Datenelement oder eine Methode
- Ein numerischer Wert
- Ein konstanter String Wert

Kein anderer Teil der Class-Datei enthält spezifische Referenzen auf Strings, Klassen, Datenelemente oder Methoden. All diese Referenzen werden im Konstantenpool gespeichert.

Die Bytes der Class-Datei bezeichnen die Namen und Typen der Datenelemente und Methoden der Klasse. Der Typ eines Datenelementes oder einer Methode wird durch einen String bezeichnet, der *Signatur (signature)* genannt wird. Datenelemente können ein weiteres Attribut mit dem Initialwert des Datenelements besitzen. Methoden können weitere s.g. *CODE* Attribute besitzen, die den Bytecode bezeichnen, der durch den Interpreter ausgeführt werden soll.

3.5.2 Der Bytecode und die Virtual Machine

Die *CODE* Attribute bieten Informationen für die Ausführung der Methode in der Maschinensprache der Virtual Machine. Diese Informationen beinhalten für jede Methode:

- Den maximalen Platzbedarf der Methode auf dem Stack.
- Die maximale Anzahl an Registern, welche von der Methode genutzt werden.
- Den Bytecode der Methode zur Ausführung in der JVM.
- Eine Tabelle mit *Exception-Handler*. Jeder Eintrag der Tabelle enthält ein Beginn und Ende Speicher-Offset des Bytecodes, einen Exception (Ausnahme) Typ und ein Speicher-Offset für den Handler der Exception. Wenn im Code innerhalb des Begin/Ende Offsets eine Exception eines bestimmten Typs auftritt, kann an dem gegebenen Handler-Offset der Handler für die Exception gefunden werden.

Jeder Bytecode besteht aus einen 1-Byte Opcode, gefolgt von keinen oder mehr Bytes mit zusätzlichen Operandeninformationen. Mit der Ausnahme von zwei „table lookup“ Instruktionen haben alle Instruktionen eine feste Länge, basierend auf dem Opcode.

Die Bytecode-Instruktionen (*Opcodes*) lassen sich in folgende Kategorien einteilen:

- Schieben (Push) von Konstanten auf den Stack.
- Zugriff und Manipulation eines Wertes in einem Register.
- Zugriff auf Arrays.
- Stack-Manipulation (*swap*, *dup*, *pop*)
- arithmetische und logische Umformungsanweisungen
- Kontrollflussanweisungen
- Funktionsrückgabe
- Manipulation von Datenelementen von Objekten
- Methodenaufruf
- Objekterzeugung
- Type Casts

3.5.3 Die Verifizierung von Class-Dateien

Nach dem Laden der Klasse wird sie vom Class Verifier in vier Phasen überprüft:

1. Strukturelle Überprüfung der Class-Datei: Diese Phase stellt sicher, dass die Class-Datei das Format einer Class-Datei besitzt. Alle erkannten Datenelemente müssen die korrekte Länge besitzen. Die Class-Datei darf nicht verkürzt sein oder an ihrem Ende zusätzliche Bytefolgen besitzen. Der Konstantenpool darf keine nichterkennbaren Informationen besitzen.

2. Semantische Überprüfung der Datentypen: In der zweiten Phase prüft der Verifier die Class-Datei etwas genauer. Er führt alle Überprüfung durch, die möglich sind, ohne in den Bytecode (in das *CODE* Datenelement) zu schauen. Geprüft wird insbesondere:

- Dass von *final* Klassen keine Unterklassen gebildet und *final* Methoden nicht überschrieben werden.
- Dass jede Klasse (außer *Object*) eine Oberklasse besitzt.
- Dass der Konstantenpool bestimmte Anforderungen erfüllt. Zum Beispiel müssen die Klassenreferenzen im Konstantenpool ein Datenelement besitzen, das auf einen Unicode String zeigt, der den Konstantenpool referenziert.
- Dass im Konstantenpool alle Datenelementreferenzen und Methodenreferenzen gültige Namen, gültige Klassen und gültige Typsignaturen besitzen.

Wenn die Datenelement- und Methodenreferenzen überprüft werden, stellt diese Phase weder sicher, dass die Datenelemente und Methoden wirklich existieren, noch dass die Typsignaturen mit denen echter Klassen übereinstimmen. Vielmehr muss die Signatur wie eine gültige Signatur „aussehen“.

3. Bytecode Verifizierung: Dies ist die umfangreichste Phase der Klassenüberprüfung. Der Bytecode jeder Methode wird verifiziert. Eine Datenflussanalyse wird für jede Methode durchgeführt. Der Verifier muss sicherstellen, dass jede Stelle eines Programmes erreicht werden kann. Darüber hinaus wird folgendes geprüft:

- Der Operanden-Stack hat immer die gleiche Größe und beinhaltet die gleichen Typen von Werten.
- Auf kein Register wird zugegriffen, außer es ist bekannt, dass es ein Wert mit eines passenden Typs enthält.
- Alle Methoden werden mit den passenden Argumenten gerufen.
- Alle Datenelemente werden nur mit Werten eines passenden Typs modifiziert.
- Alle Opcodes haben, auf dem Operanden-Stack und in den Registern, Argumente passenden Typs.

Weitere Informationen zu dieser Phase sind im Kapitel „Bytecode Verifier“ zu finden.

4. Überprüfung der symbolischen Referenzen (zur Laufzeit):

Aus Gründen der Effizienz werden einige Tests, die prinzipiell schon in Phase 3 hätten durchgeführt werden können, bis zur Ausführung des Codes verzögert. In der dritten Phase vermeidet der Verifier Class-Dateien zu laden, außer er muss es.

Wenn zum Beispiel eine Methode den Aufruf einer anderen Methode enthält, die ein Objekt vom TypA zurückgibt, und dieses Objekt sofort einem Datenelement desselben Typs zugewiesen wird, braucht der Verifier nicht zu prüfen, ob der Typ wirklich existiert. Wird dieses Objekt jedoch einem Datenelement vom TypB zugewiesen, müssen die Definitionen beider Typen geladen werden, um zu überprüfen, ob TypA eine Unterklasse von TypB ist.

Wird eine Instruktion, die eine Klasse referenziert, zum ersten Mal ausgeführt, macht der Verifier folgendes:

- Er lädt die Definition der Klasse, falls diese noch nicht geladen wurde.
- Er prüft, ob die aktuell laufende Klasse berechtigt ist, die angegebene Klasse zu referenzieren.
- Er initialisiert die Klassen, falls noch nicht geschehen.

Ruft eine Instruktion eine Methode zum ersten Mal auf oder greift sie auf ein Datenelement zu bzw. modifiziert dieses Datenelement, so macht der Verifier folgendes:

- Er stellt sicher, dass die Methode oder das Datenelement in der angegebenen Klasse existiert.
- Er prüft, ob die Methode oder das Datenelement die richtige Signatur besitzt.
- Er überprüft, dass die aktuell laufende Methode Zugriffrechte auf die angegebene Methode bzw. das Datenelement hat.

In dieser Phase braucht der Verifier nicht mehr den Typ eines Objekts auf dem Operanden-Stack zu überprüfen. Diese Überprüfung fand bereits in Phase 3 statt.

Nachdem die Überprüfung durchgeführt wurde, werden die Instruktionen im Bytecode Stream durch eine alternative Form von Instruktionen ersetzt. Der Opcode *new* wird beispielsweise durch *new_quick* ersetzt. Diese alternativen Instruktionen zeigen, dass eine Überprüfung dieser Instruktion bereits durchgeführt wurde und nicht erneut vollzogen werden muss. Die *quick* Instruktionen dürfen niemals in Class-Dateien erscheinen.

3.5.4 Der Bytecode Verifier

Wie oben schon angedeutet, ist die dritte Phase des Verifiers, der *Bytecode Verifier*, die komplexeste der vier Phasen der Klassenverifikation. Eine komplette Beschreibung findet sich in [JVMS2]. Der Code jeder Methode wird unabhängig überprüft. Zuerst werden die Bytes, welche die Instruktionen ausmachen, zu Sequenzen von Instruktionen aufgebrochen und der Offset des Beginns jeder Instruktion wird in einer Tabelle gespeichert.

Dann geht der Verifier zum zweiten Mal durch den Code und parst die Instruktionen. Während dieser Phase wird eine Datenstruktur aufgebaut, die Informationen über jede JVM-Instruktion der Methode enthält. Die Operanden jeder Instruktion werden auf ihre Gültigkeit überprüft. So werden u.a. Sprungziele, Zugriffe auf Register, Typen von Referenzen, Speicherbereiche und Exception Handler geprüft. Für jede Instruktion untersucht der Verifier den Inhalt und die Größe des Operanden-Stacks und der Register.

Als nächstes wird der *Data-Flow Analyzer* initialisiert und gestartet. Jede Instruktion mit einem s.g. „*changed“* Bit wird überprüft. Zuerst besitzt nur die erste Instruktion dieses Flag. Der Data-Flow Analyzer durchläuft nun einen Algorithmus, bei dem jede Instruktion mit ihren Typen und ihre Auswirkungen auf den Operanden-Stack und die Register untersucht wird. Wenn der Data-Flow Analyzer eine Methode ohne Verifizierungsfehler durchlaufen hat, ist Phase 3 für diese Methode beendet.

3.6 Der Security Manager

Während die Class Loader, Class Verifier und Sicherheitsfeatures von Java dazu gedacht sind, die innere Integrität der JVM und der in ihr laufenden Applikation zu erhöhen, ist der Security Manager der zentrale Punkt der Zugriffskontrolle. Der Security Manager arbeitet in der laufenden JVM und kontrolliert die Zugriffe auf Ressourcen. Eine Applikation kann, durch den Security Manager, eine anpassbare Security Policy definieren. Das Java API unterstützt die definierte Security Policy, indem es den Security Manager, vor der Ausführung einer möglicherweise unsicheren Aktion, nach dem entsprechenden Zugriffsrecht (*Permission*) fragt. Der Security Manager ist nur aktiv und überprüft die Rechte nur in dem Fall, wenn er vom Java API nach entsprechenden Rechten gefragt wird. Das Fragen nach einer Berechtigung geschieht durch den Aufruf von Überprüfungsmethoden (*check methods*) am Security Manager Objekt. Zum Beispiel bestimmt die *checkWrite()* Methode, ob ein Thread befugt ist, auf eine bestimmte Datei zu schreiben. Die Nutzung dieser Methoden definiert die Security Policy der Applikation. Vor Java 1.2 war die Implementation von Überprüfungsmethoden der einzige Weg eine Security Policy zu etablieren, da *java.lang.SecurityManager* eine abstrakte Klasse war, die es zu implementieren galt. Ein Entwickler hatte also seinen eigenen Security Manager zu schreiben, indem er von der abstrakten Klasse erbte und die Methoden implementierte. Dies brachte eine gewisse Flexibilität mit sich, aber auch viele Gefahren, da es nicht leicht ist, einen „sicheren“ Security Manager zu entwickeln.

Da nur ein Security Manager zugelassen ist, ergeben sich einige Limitierungen. Es ergeben sich z.B. Probleme, wenn mehrere Clients verschiedene Security Manager benötigen, um ordnungsgemäß zu funktionieren. Dies ist nicht realisierbar, da nur ein Security Manager installiert sein darf. Eine Lösung besteht darin, dass man den Quellcode beider Security Manager zu einem vereinigt. Die Realisierung ist sehr fehlerträchtig und auch nicht weiter skalierbar. Erst recht, wenn man sich vorstellt, man müsse 10 oder 20 Security Manager vereinigen.

Java in der Version 1.2 führte eine konkrete Implementation der *SecurityManager* Klasse ein, die durch eine ASCII Datei, die Security Policy Datei, zu konfigurieren ist. Der JDK 1.2 Security Manager erweitert das Konzept eines internen Access Controllers von JDK 1.1 um die Möglichkeit, sehr speziell Rechte (Permissions) den einzelnen Operationen zuordnen zu können.

3.7 Protection Domains und der Access Controller

Eine *Protection Domain* ist definiert als die Menge von Objekten, die einem *Prinzipal* zugänglich sind (Saltzer[1]). Also einer Entität eines Computersystems, der durch Autorisierung Rechte gewährt wurden. In diesem Sinne ist die Sandbox eine Protection Domain mit festen äußeren Grenzen. Rechte (Permissions) werden Klassen und Objekten nicht direkt vergeben, sondern indirekt durch ihre Zugehörigkeit zu einer Protection Domain, der Rechte gewährt wurden. Dies ist in Abbildung 20 dargestellt.

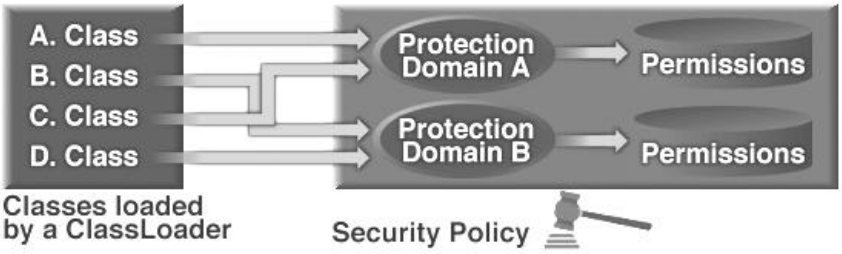

Abbildung 20: Protection Domains © SUN

Eine Protection Domain wird durch eine Security Policy Datei definiert. Der Class Loader erhält die Informationen über die Unterzeichner (*Signer*) und der Codeherkunft (*Codebase*) aus der Policy Datei und erzeugt ein *CodeSource* Objekt. CodeSource und die dazu gehörenden Permissions definieren die Protection Domain.

Der *Access Controller* ist für die Durchsetzung der Security Policy verantwortlich. Dazu untersucht der Access Controller den Stack (*Stack Introspection*), um zu bestimmen, ob eine möglicherweise unsichere Aktion verboten wird (siehe Kapitel „Zugriffskontrolle zur Laufzeit“). Die Klasse *java.security.AccessController* bietet diese Funktionalität. Sie ist kein einzelnes Objekt, sondern eine Sammlung von statischen Methoden, die zu einer Klasse zusammengefasst sind. Die Methode *checkPermission()* ist ein Mitglied der *AccessController* Klasse. Sie ist sehr wichtig, denn sie trägt die Verantwortung für die Entscheidung, ob eine Aktion erlaubt ist. Wurde ein entsprechendes Recht vergeben, gibt *checkPermission()* nichts zurück. Wurde das Recht nicht vergeben, wird eine *AccessControlerException* erzeugt. Der Standard Security Manager ruft für jede Entscheidung die *checkPermission()* Methode des Access Controllers. Daher ist der Access Controller praktisch für jede Ausführung einer bedrohlichen Aktion verantwortlich. Die *checkPermission()* Methode stellt sicher, dass jeder Stack Frame nur die Aktionen durchführt, für die er entsprechende Rechte besitzt.

Die folgende Abbildung 21 fasst die Architektur nochmals zusammen.

[1] Eine genaue Definition findet sich im Kapitel „J2EE-Security“.

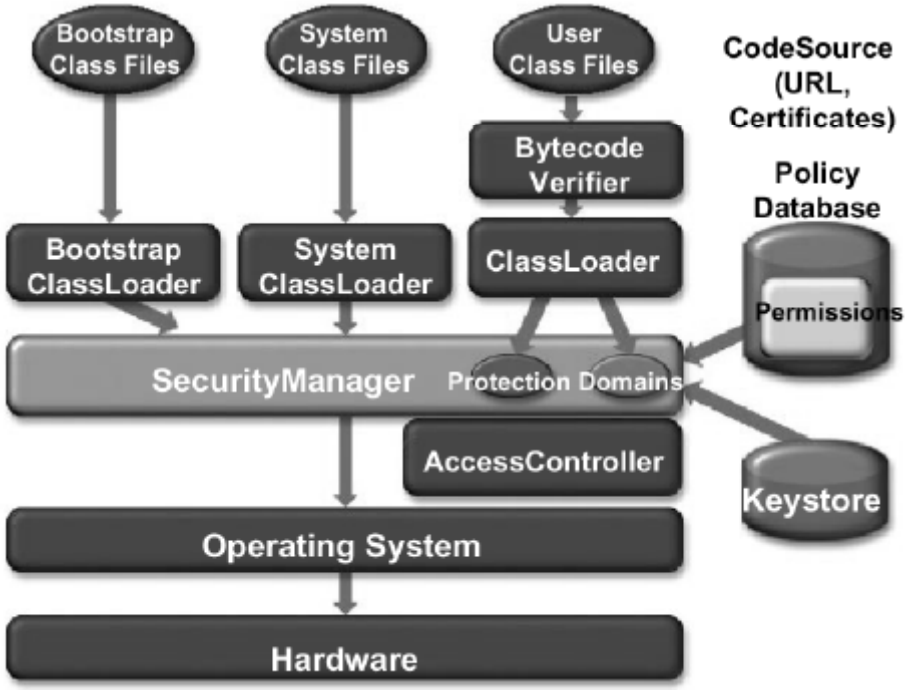

Abbildung 21: Java 1.2 Sicherheitsarchitektur © SUN

3.8 Die Zugriffskontrolle zur Laufzeit

Neben der Überprüfung der Class-Dateien durch den Class Verifier überwachen der Security Manager und der Access Controller den Code zur Laufzeit. Der Class Loader ordnet jeder Klasse einer entsprechenden Protection Domain zu, die sich aus der Security Policy ableiten lässt. So werden zum Beispiel die Basisklassen von Java einer System Domain und die Klassen einer Applikation einer Application Domain zugeordnet.

Während der Ausführung wird jedem Thread ein Java Stack in der JVM zugeordnet. Dieser Stack ist aus mehreren Stack Frames aufgebaut, wobei jeder Stack Frame den Zustand eines Methodenaufrufs des Threads repräsentiert. Jeder Thread kann eine oder mehrere Protection Domains einschließen, denn die Methodenaufrufe können von verschiedenen Klassen stammen, die verschiedenen Protection Domains angehören. Daher ist jedem Stack Frame eine Protection Domain zugeordnet.

Die Zugriffskontrolle zur Laufzeit findet nach folgendem Algorithmus statt:

0. Ein Thread versucht auf eine Ressource zuzugreifen.

1. Das Java API ruft die Methode *SecurityManager.checkPermission()*.

2. *SecurityManager.checkPermission()* ruft *AccessController.checkPermission()*.

3. *AccessController.checkPermission()* durchsucht alle Stack Frames des laufenden Threads und bestimmt dabei die Protection Domains von allen Klassen auf dem Stack.

4. *AccessController.checkPermission()* überprüft, ob alle Protection Domains die nötigen Rechte besitzen, um auf die geschützte Ressource zuzugreifen.

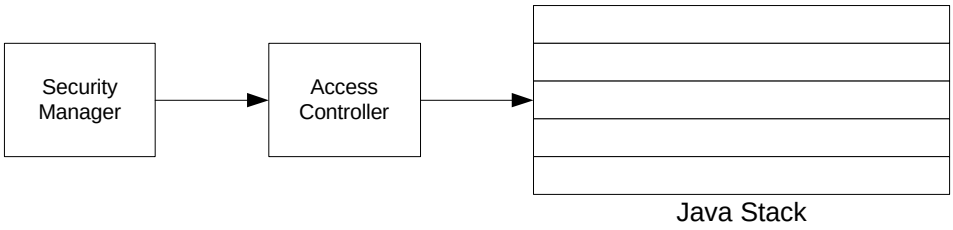

Abbildung 22: Zugriffskontrolle zur Laufzeit

In Schritt 4 bestimmt der Access Controller die Rechte des Threads, indem er die Schnittmenge der Mengen von Rechten bestimmt, die den verschiedenen Protection Domains zugeordnet sind.

Beispiel: Eine Klasse mit dem Namen „GetProperty“ enthält folgenden Code:

...
String test = System.getProperty(“user.home”);
System.out.println(“Das Benutzerverzeichnis ist: “ + test)
...

Die Auswertung (siehe Abbildung 23) sieht dann wie folgt aus:

1. *AccessController* ist in einer System Domain, die Rechte sind daher implizit gegeben. Die Auswertung wird im nächsten Stack Frame fortgesetzt.

2. *SecurityManager* ist in einer System Domain, die Rechte sind daher implizit gegeben. Die Auswertung wird im nächsten Stack Frame fortgesetzt.

3. *System* ist in einer System Domain, die Rechte sind daher implizit gegeben. Die Auswertung wird im nächsten Stack Frame fortgesetzt.

4. *GetProperty* ist in einer Application Domain. Wurden entsprechende Rechte vergeben?
   Wenn JA, dann wird die Auswertung im nächsten Stack Frame fortgesetzt.
   Wenn NEIN, dann wird eine *AccessControlException* ausgelöst.

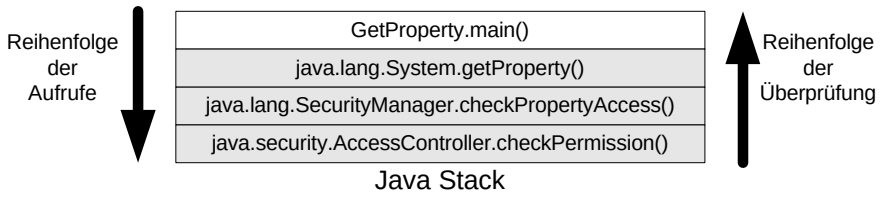

Abbildung 23: Beispiel Stack Introspection

3.8.1 Privilegierter Code

In einigen Situationen haben nicht alle Protection Domains der Stack Frames die nötigen Rechte, um eine Aktion durchzuführen. Dennoch muss es eine Möglichkeit zur Durchführung dieser Aktion geben. Dies wird durch privilegierten Code [DOPRIVILEGED] möglich.

Es kann vorkommen, dass eine Applikation nicht direkt auf eine Datei zugreifen darf, die Zeichensatzdefinitionen enthält. Diese wird jedoch von Systemklassen zur Darstellung eines Dokuments benötigt. Eine Auswertung der Protection Domains der Stack Frames würde ergeben, dass die Frames, die mit der Applikation assoziiert sind, nicht über die entsprechenden Rechte verfügen und ein Zugriff durch die Systemklassen wäre nicht erlaubt.

Um einem solchen Problem zu begegnen, führen die Systemklassen derartige Aktionen privilegiert durch. Die *AccessController.doPrivileged()* Methode gibt einem Teil von vertrauenswürdigen Code temporär Zugriff auf mehr Ressourcen, als der ihn rufenden Applikation.

Wenn eine Klasse die Methode *doPrivileged()* ruft, wird ihr Stack Frame markiert und der Access Controller muss die Überprüfung des Stacks nach diesem Frame beenden.

Privilegierter Code wird immer in der *run()* Methode des *java.security.PrivilegedAction* Interfaces implementiert. Dabei sind drei Fälle zu unterscheiden:

Fall 1: Der privilegierte Code erzeugt keine Exceptions und liefert keinen Rückgabewert. Die *AccessController.doPrivileged()* Methode erhält ein *PrivilegedAction* Objekt als Argument, ruft die *run()* Methode und gibt deren Rückgabewert (null) zurück. In diesem Fall gibt es zwei Implementierungsoptionen:

Option 1: Die Implementierung einer inneren anonymen Klasse:

```
someMethod()
{
        // normaler Code ...
        AccessController. doPrivileged( new PrivilegedAction()
                {
                        public Object run()
                        {
                                // privilegierter Code, zum Beispiel:
                                System. loadLibrary(" awt");
                                return null; // Rückgabe von null
                        }
                });
        // normaler Code ...
}
```

Die Verwendung von inneren Klassen birgt ein gewisses Risiko. Viele Javabücher behaupten, auf innere Klassen könnten nur ihre äußeren Klasse zugreifen. Dies stimmt so nicht! Der Compiler übersetzt sie in normale Klassen, da der Bytecode das Konzept der inneren Klassen nicht kennt.

Option 2: Die Implementierung einer externen Klasse:

```
class MyExternalPrivilegedAction implements PrivilegedAction
{
        public Object run()
        {
                // privilegierter Code, zum Beispiel:
                System. loadLibrary("awt");
                return null; // Rückgabe von null
        }
}


somemethod()
{
        // normaler Code...
        AccessController. doPrivileged( new MyExternalPrivilegedAction());
        // normaler Code...
}
```

Option 2 sollte aus Gründen der Sicherheit und Lesbarkeit bevorzugt werden. Das Java API verwendet primär Option 1!

Fall 2: Der privilegierte Code erzeugt keine Ausnahmen (Exceptions) und liefert einen Rückgabewert. Die *AccessController.doPrivileged()* Methode erhält ein *PrivilegedAction* Objekt als Argument, ruft die *run()* Methode und gibt deren Rückgabewert zurück.

Fall 3: Der privilegierte Code erzeugt eventuell eine Exception und liefert einen Rückgabewert. Die *AccessController.doPrivileged()* Methode erhält ein *PrivilegedAction* Objekt als Argument, ruft die *run()* Methode und gibt deren Rückgabewert zurück. Mit einem Try-Catch Block muss die *PrivilegedActionException* aufgefangen werden.

Beispiel:

- Es seien vier Klassen gegeben: *CountFileCaller1*, *CountFile1*, *CountFileCaller2* und *CountFile2*.
- *CountFileCaller1* ruft *CountFile1* und *CountFileCaller2* ruft *CountFile2*.
- Die Klassen *CountFile1* und *CountFile2* versuchen auf eine Datei lesend zuzugreifen, wobei *CountFile1* den Zugriff auf die Datei privilegiert durchführt.

Die Klassen besitzen folgende Rechte:

Klassenname	CodeBase	FilePermission	doPrivileged()
CountFileCaller1	C:\VerzeichnisA\		
CountFileCaller2	C:\VerzeichnisA\		
CountFile1	C:\VerzeichnisB\	grant	grant
CountFile2	C:\VerzeichnisB\	grant	

CountFile1 für seine privilegierte Aktion wie folgt (Fall 3, implementiert als innere anonyme Klasse) durch:

```
import java. io.*;
import java. security.*;

class MyPrivilegedExceptionAction implements PrivilegedExceptionAction {
        public Object run() throws FileNotFoundException{
                FileInputStream fis = new FileInputStream(" C:\\PASSWORDS. TXT");
                        try {
                                int count = 0;
                                while (fis. read() != -1)
                                        count++;
                                System. out. println("Die Datei enthält " + count + " Zeichen.");
                        }
                        catch (Exception e) {
                        System. out. println(" Exception " + e);
                        }
                return null;
        }
}
```

Ergebnis:

CountFile2 besitzt zwar die nötigen Rechte, *CountFileCaller2* aber nicht. Bei der Auswertung wird der gesamte Stack untersucht. Bei der Überprüfung des Stack Frames von *CountFile-Caller2* wird eine *java.security.AccessControlException* erzeugt.

CountFile1 besitzt die nötigen Rechte, *CountFileCaller1* aber nicht. Bei der Auswertung wird der Stack nur bis inklusive *CountFile1* untersucht. *CountFile1* darf auf die Datei zugreifen.

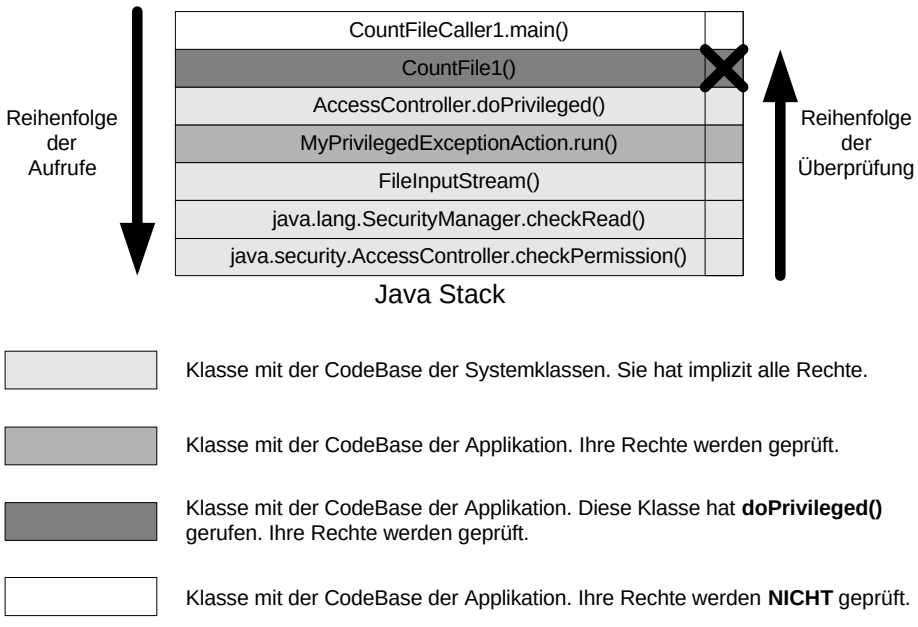

Abbildung 24: Beispiel doPrivileged

3.9 Die Java Security APIs

3.9.1 Java Cryptography Architecture (JCA)

Das JDK Security API ist eines der APIs, die zum Sprachkern der Programmiersprache Java gehören. Es umfasst das Paket *java.security* sowie seine Unterpakete. Es ermöglicht Entwicklern, low-level und high-level Sicherheitsfunktionalitäten in ihre Programme einzubauen. Das Security API wurde in JDK 1.1 mit der *Java Cryptography Architecture* (JCA) eingeführt, welche der Java Plattform einen Framework für kryptographische Funktionalitäten bietet. In JDK 1.1 umfasste JCA verschiedene APIs für digitale Signaturen und Einweg-Hashfunktionen. Java 2 SDK (JDK 1.2) führte eine neue Sicherheitsarchitektur ein und erweiterte JCA um diverse Funktionalitäten, wie zum Beispiel einer neuen Zertifikatsverwaltung, die X.509 v3 Zertifikate unterstützt. JCA umfasst die kryptographischen Teile des Java 2 Security API sowie verschiedene Konventionen und Spezifikationen, wie der *Provider Architektur*, die mehrere interoperable Kryptographieimplementationen ermöglicht.

Folgende Designprinzipen bilden für JCA die Grundlage:

- Unabhängigkeit von Algorithmen
- Implementationsunabhängigkeit
- Implementationsinteroperabilität
- Erweiterbarkeit von Algorithmen

Die Unabhängigkeit von Algorithmen wird durch die Definition der Typen der kryptographischen *Engines* (Dienste) und der Definition der Klassen, welche die Funktionalitäten dieser Dienste enthalten, erreicht. Diese Klassen werden *Engine Classes* genannt. Sie definieren einen kryptographischen Dienst in abstrakter Weise (also ohne konkrete Implementation). Die im API definierten Methoden ermöglichen Applikationen, auf einen speziellen Typ von kryptographischen Dienst, wie z.B. einen Signaturalgorithmus, zuzugreifen. Die Applikationsschnittstelle, die von den Engine Klassen unterstützt wird, wird Service Provider Interface (SPI) genannt. Beispiele hierfür sind die Klassen *MessageDigest*, *Signature* und *KeyFactory*.

Implementationsunabhängigkeit wird durch eine *provider*-basierte Architektur erreicht, die mehrere interoperable Kryptographieimplementationen erlaubt. Ein *Cryptographic Service Provider* (CSP) oder einfach Provider liefert kryptographische Erweiterungen bzw. konkrete Implementationen in Form von Paketen, die auf Service Provider Interfaces (SPIs) aufsetzen. Diese Pakete implementieren einen oder mehrere kryptographische Dienste, wie zum Beispiel *Digital Signature Algorithms* (DSAs), *Message Digest Algorithms* (MDAs) oder Dienste zur Schlüsselerzeugung. Ein Programm fordert einfach einen bestimmten Typ eines Objekts (z.B. ein *Signature* Objekt) an, das einen bestimmten Dienst (z.B. den DSA Signaturalgorithmus) implementiert, und bekommt dann eine Implementation von einem bestimmten Provider geliefert. Dabei kann das Programm auch nach einer Implementation von einem bestimmten Provider fragen. Die Provider können, für die Applikation transparent, aktualisiert werden, wenn zum Beispiel eine neue schnellere oder sicherere Version verfügbar ist.

Implementationsinteroperabilität bedeutet, dass verschiedene Implementationen miteinander interagieren können. Wenn zum Beispiel ein Provider eine Signatur generiert, dann ist die Signatur auch von einem anderen Provider verifizierbar.

Die Erweiterbarkeit von Algorithmen bedeutet, dass jeder neue Algorithmus, der in eine der unterstützten Engine Classes „passt“, einfach hinzugefügt werden kann.

Im JDK 1.1 konnte ein Provider die Implementation einer oder mehrerer Signaturalgorithmen, Einweg-Hashfunktionen oder Schlüsselerzeugungsalgorithmen enthalten. JDK 1.2 fügte sechs weitere Typen von Diensten hinzu:

- *Keystore*[1] Generierung und Verwaltung.
- Verwaltung von Parametern für Kryptoalgorithmen.
- Generierung von Parametern für Kryptoalgorithmen.
- Unterstützung von *Key Factories*, zur Konvertierung zwischen verschiedenen Schlüsselrepräsentationen.
- Unterstützung von *Certificate Factories*, zur Generierung von Zertifikaten und CRLs (Certificate Revocation Lists).
- Unterstützung von Zufallszahlengeneratoralgorithmen (RNG[2]).

Abbildung 25 zeigt die verschiedenen JCA-Module.

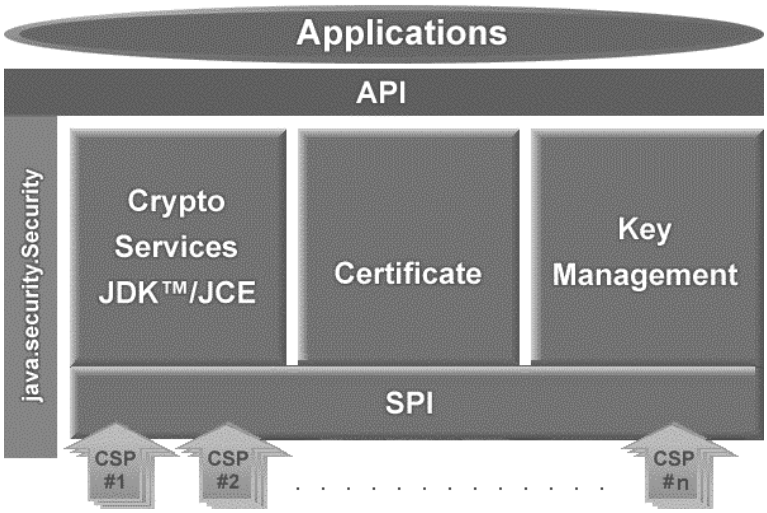

Abbildung 25: JCA-Module © SUN

java.security

Das *java.secur*ity Paket beinhaltet größtenteils abstrakte Klassen und Interfaces, die Sicherheitskonzepte kapseln, wie z.B. Zertifikate, Schlüssel und Signaturen. Konkrete Implementationen werden, wie oben schon angedeutet, hier nicht gefunden. Diese können selbst programmiert oder von einem CSP bezogen werden. Das Paket besitzt einen Anwender- und einen Herstellerteil. Der Hersteller (CSP) implementiert die Kryptoalgorithmen und der Anwender implementiert Applikationen mit den Kryptoalgorithmen eines Herstellers. Um den installierten Kryptoalgorithmus eines Herstellers zu aktivieren, gibt es zwei Wege.

[1] Speicherplatz (Datenbank) für Schlüssel
[2] RNG = Random Number Generation

Zur Spezifikation eines neuen Providers muss die Datei */lib/security/java.security* unter dem JRE Installationsverzeichnis aktualisiert werden. Es muss mindestens ein Provider installiert sein. Per Voreinstellung ist der Provider SUN *sun.security.provider.SUN* eingetragen.

Oder man fügt einen neuen Provider mit der Methode *Security.addProvider (Provider)* hinzu. Dieser Weg ist jedoch problematisch, da eine spätere Änderung des Providers eine erneute Kompilierung und Verteilung des Programmcodes notwendig macht.

java.security.acl

Das Paket *java.security.acl* definiert die Unterstützung von *Access Control Lists*. Zugriffskontrolllisten können für Autorisierungsentscheidungen eingesetzt werden.

java.security.interfaces

Das JCA 1.1 Sicherheitspaket *java.security.interfaces* beinhaltet Interfaces zur Nutzung des *Digital Signature Algorithm (DSA)*. JCA 1.2 beinhaltet zusätzlich noch Interfaces für das RSA[1] Verfahren und das Chinese Remainder Theorem (CRT).

java.security.cert

Das JCA 1.2 Sicherheitspaket *java.security.cert* unterstützt die Generierung und Verwendung von Zertifikaten (z.B. X.509 Zertifikate) sowie Zugriffsmechanismen auf JAR Dateien über *JarURLConnection* und *JarEntry*.

java.security.spec

Das Paket *java.security.spec* beinhaltet Interfaces, die Spezifikationsformate für Schlüssel beschreiben.

3.9.2 Java Cryptography Extension (JCE) API

Das JCA API wird durch das *Java Cryptography Extension* (JCE) API (*javax.crypto.**) erweitert. Es bietet einen Framework und Implementationen mit Algorithmen zur Verschlüsselung, Schlüsselerzeugung, Schlüsselaustausch und Einweg-Hashfunktionen. Es wird asymmetrische, symmetrische, Block- und Stromverschlüsselung unterstützt. Die Software unterstützt auch *secure Streams* und *sealed Objects*.

JCE 1.2.2 [JCES122] wurde so designed, dass andere qualifizierte Kryptographiebibliotheken als Service Provider sowie neue Algorithmen problemlos hinzugefügt werden können. Mit „qualifiziert“ meint die Spezifikation, dass diese Bibliotheken bzw. Provider bezüglich der U.S. Exportbestimmungen bewertet und entsprechend, von einer vertrauenswürdigen Entität, signiert wurden. Daher wird JCE getrennt von JCA ausgeliefert. In Java 2 SDK 1.4 (Codename: MERLIN) ist JCE mittlerweile integriert.

[1] RSA wurde benannt nach den Erfindern: Ron Rivest, Adi Shamir und Leonard Adleman

JCE 1.2.2 ist die erste Version, die außerhalb der USA und Kanada genutzt werden darf, da sie einen Mechanismus für „qualifizierte“ Provider enthält, der es unmöglich macht, nicht-qualifizierte Provider in den Framework zu integrieren. Der JCE 1.2.2 Framework besitzt die Fähigkeit Restriktionen, bezüglich der Kryptoalgorithmen und der maximalen Stärke der Kryptographie, zu verhängen. Diese Restriktionen werden in so genannten *Jurisdiction Policy Dateien*, nach verschiedenen juristischen Zuständigkeitskontexten[1] (jurisdiction contexts), spezifiziert, die mit der JCE 1.2.2 Software verteilt werden.

SUN bietet die JCE 1.2.2 Software weltweit in einem einzelnen Bundle an. Die enthaltenen Jurisdiction Policy Dateien besitzen keine Restriktionen bezüglich der Stärke der Kryptographie. Dies ist für die meisten Länder angemessen. Andere Hersteller können Bundles anbieten, die Jurisdiction Policy Dateien enthalten, die an die Restriktionen der Regierungen der Zielländer angepasst sind. Benutzer aus diesen Ländern können so ein passendes Bundle beziehen und der JCE-Framework setzt die spezifizierten Restriktionen durch. Der mitgelieferte Provider *SunJCE* liefert folgende Implementationen:

- DES
- DESede
- Blowfish
- PBEWithMD5AndDES
- PBEWithMD5AndTripleDES
- Diffie-Hellman Schlüsselaustausch
- HMAC-MD5
- HMAC-SHA1

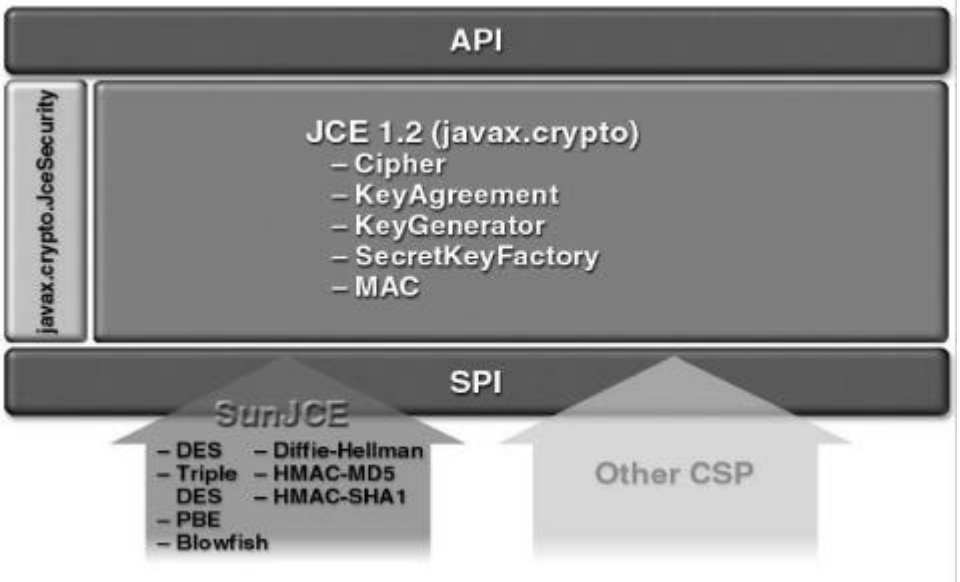

Abbildung 26: JCE API © SUN

[1] zum Beispiel: USA, Deutschland, etc.

3.9.3 Java Authentication and Authorization Service (JAAS) API

Der *Java Authentication and Authorization Service* (JAAS) 1.0 [JAASS10] ist eine Sammlung von Paketen, die es Diensten ermöglichen, Authentisierungs- und Autorisierungsentscheidungen auf der Basis von Benutzern durchzuführen.

JAAS implementiert eine Javaversion des Standards *Pluggable Authentication Module (PAM) Framework* [SS95] und erweitert damit die Zugriffskontrollarchitektur der Java 2 Plattform, auf einer kompatiblen Art und Weise, zur benutzerbasierten Autorisierung. Abbildung 27 zeigt die Architektur des JAAS API, das mittlerweile in Java 2 SDK 1.4 integriert ist.

PAM definiert ein generisches API für Authentisierungsmechanismen, wobei es von den darunter liegenden Mechanismen abstrahiert. Dies erleichtert den Austausch von Authentisierungskomponenten und bietet eine gute Lösung für das „Single Sign-On“ Problem. PAM ist in Linux, Solaris und dem Common Desktop Environment (CDE) implementiert. Mit PAM können mehrere Authentisierungstechnologien hinzugefügt werden, ohne die Login Dienste zu ändern. Die existierende Systemumgebung wird dabei nicht verändert. PAM kann genutzt werden, um verschiedene Authentisierungstechnologien, wie RSA, DCE, Kerberos, S/Key und Smart Card basierende Authentisierungssysteme, und bestehende Login Dienste zu integrieren.

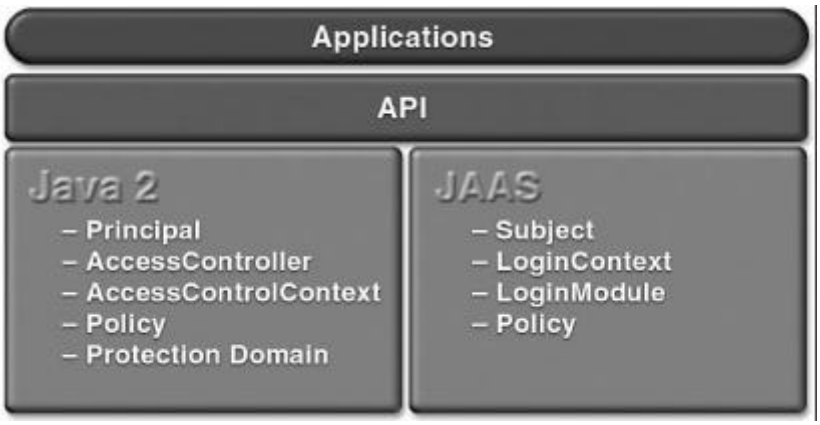

Abbildung 27: JAAS API © SUN

3.9.3.1 Authentisierung mit JAAS

Applikationen starten den Authentisierungsprozess, indem sie ein *LoginContext* Objekt erzeugen, das ein *Configuration* Objekt liefert, welches die Authentisierungstechnologie bzw. das *LoginModule* bestimmt, das für die Authentisierung genutzt werden soll. Dies hat den Vorteil, dass die Applikationen und der Authentisierungsmechanismus nicht fest verbunden sind. Wird der Authentisierungsmechanismus später geändert, braucht die Applikation nicht neu kompiliert und verteilt zu werden. Diese Architektur ist in Abbildung 28 dargestellt.

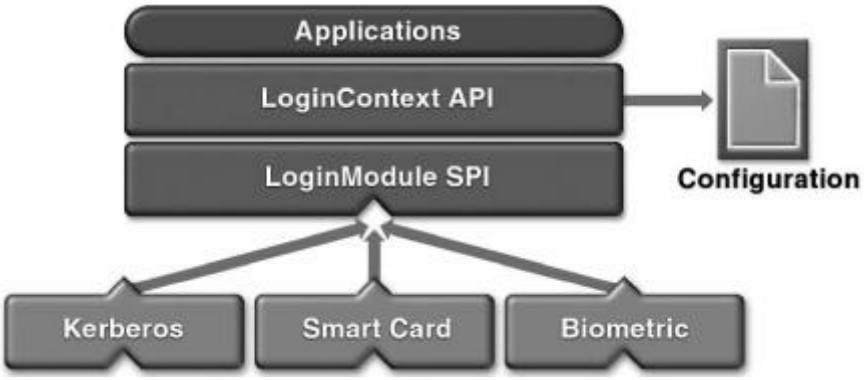

Abbildung 28: JAAS Login Module © SUN

3.9.3.2 Autorisierung mit JAAS

Nach der Authentisierung des Benutzers beginnt die Arbeit der JAAS Autorisierungskomponente, die mit den vorhandenen Java 2 Zugriffskontrollmechanismen zusammenarbeitet. Im Gegensatz zu Java 2, wo die Zugriffskontrollentscheidungen auf der *CodeSource* (Herkunftsort des Codes und Unterzeichner) beruhen, basieren die JAAS Zugriffskontrollentscheidungen auf der *CodeSource* und dem *Subject*, das den Code ausführt. JAAS erweitert lediglich die Java 2 Policy um die relevanten Subjekt bzw. Prinzipal Informationen. Die schon diskutierten *Permissions* (Rechte) können auch von JAAS interpretiert werden. JAAS besitzt seine eigene JAAS Security Policy Datei. Die beiden physikalisch getrennten Policy Dateien (J2SE und JAAS) bilden zusammen eine logische Policy. Abbildung 29 zeigt das benutzerbasierte Java 2 Sicherheitsmodell.

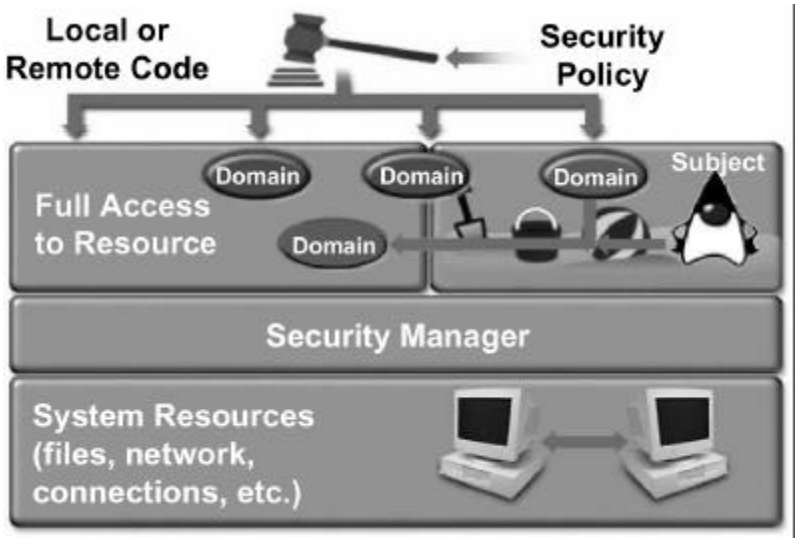

Abbildung 29: benutzerbasierte Autorisierung mit JAAS © SUN

3.9.4 Java Secure Socket Extension (JSSE) API

Die Java Secure Socket Extension (JSSE) [JSSE103] bietet einen Framework und eine Implementation einer Javaversion der SSL und TLS Protokolle. JSSE unterstützt die Verschlüsselung von Daten, Serverauthentisierung, Nachrichtenintegrität und eine optionale Authentisierung von Clients. Mit JSSE lassen sich TCP/IP Verbindungen wie HTTP, Telnet und FTP absichern.

JSSE war ein optionales Paket, das mittlerweile in das Java 2 SDK 1.4 integriert ist. Das JSSE API ergänzt die kryptographischen Dienste (*java.security*) der JCA und bietet: erweiterte Netzwerksocketklassen (*java.net*), *Trust Manager*, *Key Manager*, *SSLContext* und einen *Socket Factory Framework*, zur Kapselung der Netzwerksocketerzeugung.

Das JSSE API ist in der Lage, SSL in den Versionen 2.0 und 3.0 sowie Transport Layer Security (TLS) 1.0 zu unterstützen. Diese Sicherheitsprotokolle kapseln den normalen bidirektionalen Streamsocket und die JSSE API fügt Funktionen zur Authentisierung, Verschlüsselung und Integritätsschutz hinzu. Das JSSE aus dem Java 2 SDK 1.4 implementiert SSL 3.0 und TLS 1.0. Es implementiert nicht SSL 2.0. Die einzelnen kryptographischen Algorithmen sind unter [JSSE103] zu finden. Abbildung 30 zeigt die Architektur des JSSE API.

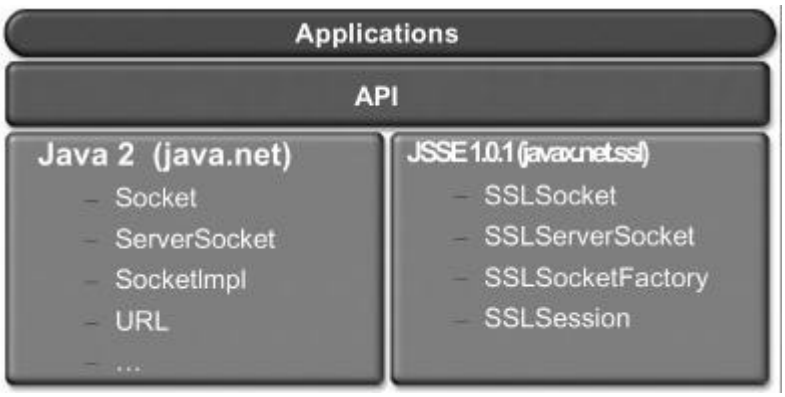

Abbildung 30: JSSE API © SUN

3.9.5 Java Generic Security Service (Java GSS) API

Das *Java GSS-API* ist ein Mechanismus für den sicheren Nachrichtenaustausch von Applikationen. Seine Funktionen umfassen: Authentisierung, Delegation und Sicherung der Vertraulichkeit und Integrität von Nachrichten. Das GSS-API ist definiert durch [RFC2743] und [RFC2853]. RFC 2853 definiert zwei Mechanismen: Einen einfachen Public-Key GSS-API Mechanismus und einen Kerberos Version 5 [RFC1510] GSS-API Mechanismus. Im Java 2 SDK 1.4 ist es mittlerweile integriert.

Das GSS-API und JSSE bieten sehr ähnliche Sicherheitsfunktionalitäten. JSSE eignet sich gut zur Sicherung von socketbasierter Netzwerkkommunikation. GSS bietet andere Funktionen, wie sichere Kommunikation über UDP, Authentisierung mit Kerberos Version 5, Delegation[1] und Personifizierung. Daher eignet sich das GSS-API, in Verbindung mit JAAS und Kerberos, sehr gut für Single Sign-On Lösungen. Dies ist in Abbildung 31 dargestellt.

Eine Sicherung der Kommunikation zwischen zwei Applikationen wird mit dem GSS-API in vier Schritten aufgebaut. Im ersten Schritt bezieht die Applikation einen Identitätsnachweis (*Credential*), der die Applikation gegenüber der anderen Applikation identifiziert. Dann bilden beide Applikationen, mit *GSSContext*, einen Sicherheitskontext (*Security Context*). Dieser Kontext enthält gemeinsame Informationen, wie kryptographische Schlüssel und Sequenznummern von Nachrichten. Der dritten Schritt nutzt Methoden von *GSSContext*, um für jede Nachricht Dienste zur Authentisierung, Integrität und Vertraulichkeit aufzurufen. Der vierte und letzte Schritt führt abschließend verschiedene Aufräumoperationen durch.

[1] Delegation und Personifizierung wird im Kaptitel „J2EE-Security“ ausführlich erklärt.

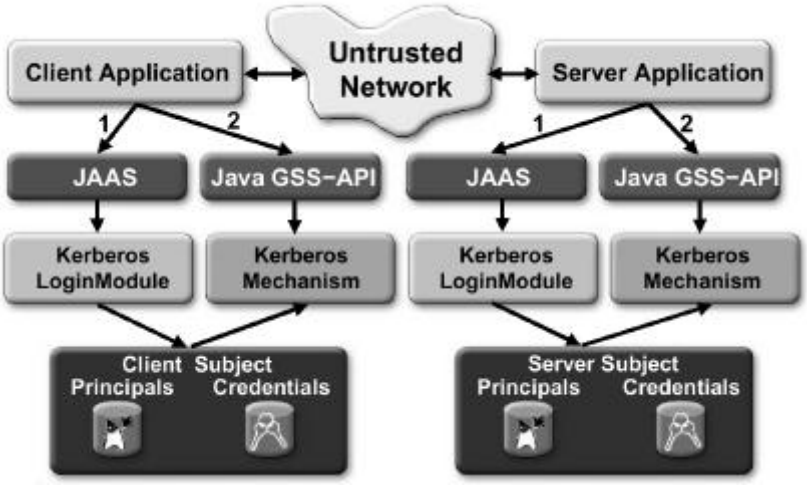

Abbildung 31: Single Sign-On mit dem Java GSS API © SUN

3.9.6 Java Certification Path (CertPath) API

Das *CertPath API* erweitert das *java.security.cert* Paket um neue Klassen und Methoden für die Erstellung, Validierung und Speicherung von *Certification Paths* (Zertifikatsketten) und Zertifikaten. Es unterstützt den PKIX Standard, der einen Algorithmus für die Validierung von X.509 Zertifikatsketten definiert, und RFC2587, das ein LDAP-Schema zur Suche in Zertifikatsketten definiert. CertPath besitzt, wie JCE, eine modulare Providerarchitektur, die in Abbildung 32 dargestellt ist.

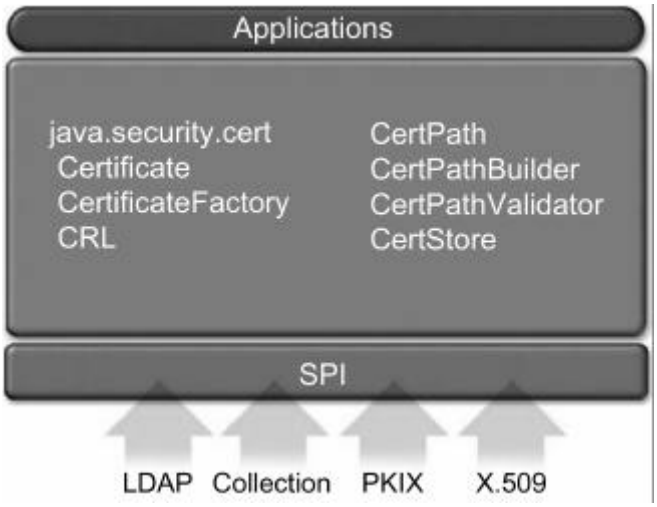

Abbildung 32: CertPath API © SUN

4 J2EE-Security

Fast jede Unternehmung hat spezielle Anforderungen an ihre Sicherheit und spezifische Mechanismen und Infrastrukturen, um diese zu erfüllen. Sensitive Ressourcen, auf die von vielen Benutzern zugegriffen kann oder welche oft über ungeschützte offene Netzwerke (wie das Internet) kommunizieren, müssen geschützt werden. Obwohl die Qualitätszusicherungen und Implementierungsdetails variieren können, teilen sie alle die gleichen Merkmale:

Authentisierung: Die Überprüfung der Identität der kommunizierenden Entitäten (z.B. Client und Server).

Zugriffskontrolle (Autorisierung): Prüfung der Aktionen von authentisierten Entitäten, um Limitierungen der Zugriffsrechte auf Ressourcen durchzusetzen.

Datenintegrität: Es ist zu prüfen, ob Informationen von Dritten modifiziert wurden.

Vertraulichkeit und Datenschutz: Sicherstellung, dass Informationen nur solchen Benutzern zur Verfügung gestellt werden, die auch autorisiert sind, auf diese zuzugreifen.

Nicht-Abstreitbarkeit: Es muss überprüfbar sein, ob ein Benutzer eine spezielle Handlung vorgenommen hat, so dass der Benutzer seine Handlung nicht abstreiten kann.

Auditing: Die manipulationssichere Aufzeichnung von sicherheitsrelevanten Geschehnissen mit dem Zweck, jederzeit die Effektivität der Sicherheitsmechanismen und Policies überprüfen zu können.

Die J2EE-Plattform verfolgt das Ziel eines sicheren, interoperablen und verteilten komponenten-basierenden Systems. Eines der primären Ziele der J2EE-Plattform ist es, den Applikationsentwickler von den Details der Sicherheitsmechanismen zu entlasten und ein sicheres Deployment von Applikationen in verschiedenen Umgebungen zu erleichtern. Die J2EE-Plattform definiert eine klare Trennung von Verantwortlichkeiten zwischen denen, die Applikationskomponenten entwickeln, denen, die Komponenten zu Applikationen zusammenfügen, und denen, die Applikationen für eine spezifische Umgebung konfigurieren. Der *Component Provider* und *Application Assembler* spezifizieren die zu schützenden Teile der Applikation in *Deployment Descriptoren*, die als Kommunikationsmittel, außerhalb des Codes des Entwicklers, verwendet werden. Anhand der Elemente der Deployment Descriptoren kann der *Deployer* eine entsprechende Konfigurationen, zum Schutz der Applikation, vornehmen. Die Deployment Descriptoren machen container-spezifische Tools möglich. Diese erleichtern es dem *Deployer,* die vom Entwickler geforderten Sicherheitsanforderungen umzusetzen.

Der *Application Component Provider* identifiziert alle sicherheitsrelevanten Abhängigkeiten, die in einer Komponente existieren, einschließlich:

- Die Namen aller Sicherheitsrollen (role names), die von der Komponente für Aufrufe der Methoden *isCallerInRole* oder *isUserInRole* genutzt werden.
- Referenzen auf alle externen Ressourcen, auf welche die Komponente zugreift.
- Referenzen auf alle Aufrufe zwischen Komponenten, die von der Komponente getätigt werden.

Der *Application Component Provider* kann auch ein Method-Permission-Modell zusammen mit Informationen liefern, welche die Sensitivität der in den jeweiligen Aufruf auszutauschenden Informationen, im Bezug auf den Datenschutz (Privacy), kennzeichnet.

Der *Application Assembler* kombiniert eine oder mehr Komponenten zu einem Applikationspaket. Anschießend produziert er aus den Sicherheitsanforderungen, der einzelnen Komponenten, eine konsistente Sicherheitsanforderung der gesamten Applikation. Ziel ist es, dem *Deployer* die relevanten Anforderungen mitzuteilen, nach denen er sich bei seinen weiteren Aktionen richten kann.

Der *Deployer* ist verantwortlich für die Umsetzung der Sicherheitsanforderungen der Applikation, die ihm vom *Application Assembler* geliefert wurden. Für die Abbildung der Anforderungen auf die Policies und Mechanismen der operativen Umgebung nutzt er plattform-spezifische *Deployment Tools*. Er konfiguriert, die vom Container implementierten, Sicherheitsmechanismen, auf dem die Komponenten deployed sind.

Die J2EE-Sicherheitsmechanismen kombinieren die Konzepte von Container-Hosting und deklarativer Sicherheit (*Declarative Security*), bei welcher die Sicherheitsanforderungen der Applikation extern deklariert werden. Optional können in Applikationen eingebundene Sicherheitsmechanismen (*Programmatic Security*) verwendet werden.

Auf die einzelnen Konzepte und Mechanismen wird im folgenden genauer eingegangen. Es sollen insbesondere die Anforderungen der Spezifikationen dargestellt werden, um später die Implementation (am Beispiel Websphere) konzeptionell zu überprüfen.

4.1 Terminologie

Principal (Prinzipal)

Ein *Principal* ist eine Entität, die mit einem Authentisierungsprotokoll durch einen Dienst authentisiert werden kann. Ein Prinzipal wird identifiziert durch seinen *Principal Name* und authentisiert durch seine *Authentication Data.* Der Inhalt und das Format des *Principal Name* und der *Authentication Data* kann abhängig vom Authentisierungsprotokoll variieren. Saltzer [SALTZER75] definiert Prinzipal als: *“Principal: The entity in a computer system to which authorizations are granted; thus the unit of accountability in a computer system.”*

Protection Domain (Schutzdomäne)

Ein grundlegendes Konzept und wichtiger Baustein der Systemsicherheit ist die *Protection Domain*. Eine Protection Domain kann als die Menge von Objekten abgegrenzt werden, die einem Prinzipal, zu einem gegebenen Zeitpunkt, zugänglich sind. Die *Sandbox* in JDK 1.0 ist ein Beispiel für eine Protection Domain mit festen Grenzen. Konzeptuell umfasst eine *Protection Domain* eine Menge von Klassen, deren Exemplare die gleiche Menge von Rechten gewährt wurde. Protection Domains werden durch die zu einem gegebenen Zeitpunkt gültigen Policy determiniert. Saltzer definiert Domain als: *”Domain : The set of objects that currently may be directly accessed by a principal.“*

Security Attributes (Sicherheitsattribute)

Mit jedem Prinzipal ist eine Menge von *Security Attributes* assoziiert. Die Sicherheitsattribute können verschiedenes ausdrücken (z.B. Zugriffsrechte auf eine geschützte Ressource, Auditing von Benutzern u.s.w.). Sicherheitsattribute können vom Authentisierungsprotokoll und/oder vom J2EE-Produkt Provider mit dem Prinzipal assoziiert werden. Die J2EE-Plattform spezifiziert nicht, wie die Security Attribute mit dem Prinzipal assoziiert werden.

Credential (Identitätsnachweis)

Ein *Credential* enthält oder referenziert Informationen (Sicherheitsattribute), die gebraucht werden, um den Prinzipal gegenüber einem Dienst des J2EE-Produkts zu authentisieren. Ein Principal erhält einen Credential nach der Authentisierung oder von einem anderen Principal, der ihm erlaubt, den Credential zu nutzen. Die Spezifikation spezifiziert nicht den Inhalt oder das Format des Credential. Inhalt und Format können sehr variieren. Es handelt sich um einen ticketbasierten Authentisierungsnachweis.

Capability (Berechtigungsnachweis)

Saltzer definiert *Capability* als: *"Capability: In a computer system, an unforgeable ticket, which when presented can be taken as incontestable proof that the presenter is authorized to have access to the object named in the ticket."* Es handelt sich um einen ticketbasierten Autorisierungsnachweis.

Security Role (Sicherheitsrolle)

Das J2EE-Sicherheitsmodell ist ein rollenbasiertes Sicherheitsmodell. Eine *Security Role* ist eine semantische Gruppierung von Permissions, die ein gegebener Typ von Benutzern haben muss, um eine Applikation benutzen zu können.

Sicherheitsrollen werden vom *Application Component Provider* oder *Assembler* definiert. Der *Deployer* bildet die Security Roles auf die *Security Identities* (Identitäten) (z.B. Benutzer und Gruppen) der operativen Umgebung ab. Security Roles werden bei *Declarative Security* und *Programmatic Security* genutzt.

Security Policy Domain

Eine S*ecurity Policy Domain*, auch *Security Domain* genannt, ist der Wirkungsbereich einer Security Policy. Eine *Security Policy Domain* wird manchmal auch *Realm* genannt. Die Spezifikation benutzt die Begriffe *Security Policy Domain* und S*ecurity Domain.*

Security Technology Domain

Eine *Security Technology Domain* ist der Wirkungsbereich, in dem ein spezieller Sicherheitsmechanismus (z.B. Kerberos) genutzt wird, um die Security Policy durchzusetzen. Eine einzelne Security Technology Domain kann mehrere Security Policy Domains enthalten.

4.2 Ziele der J2EE-Sicherheitsarchitektur

Dieser Abschnitt gibt die Ziele der J2EE-Sicherheitsarchitektur wieder und entspricht den Zielen der J2EE-Spezifikation von SUN.

1. Portabilität: Die J2EE-Sicherheitsarchitektur muss die Eigenschaft „Write Once, Run Anywhere” für Applikationen erfüllen.

2. Transparenz: Provider von Applikationskomponenten sollten kein umfangreiches Sicherheitswissen benötigen, um eine „sichere“ Applikationskomponente zu entwickeln.

3. Isolation: Die J2EE-Plattform soll die Authentisierung und Zugriffskontrolle nach den Anforderungen des Deployers, unter Verwendung der Deployment-Attribute, durchführen und deren Verwaltung durch einen Systemadministrator unterstützen.

Anmerkung: Die Trennung der Entwicklung einer Applikation von der Verantwortung für die Sicherheit der Applikation gewährleistet eine größere Portabilität von J2EE-Applikationen.

4. Erweiterbarkeit: Der Gebrauch von Plattformdiensten in sicherheitskritischen Applikationen darf nicht die Portabilität der Applikation kompromittieren. Die Spezifikation bietet APIs nach dem komponentenbasierten Paradigma zur Interaktion mit Container/Server Sicherheitsinformationen. Applikationen, die ihre Interaktionen auf diese APIs einschränken, behalten ihre Portabilität.

5. Flexibilität: Die Sicherheitsmechanismen und Deklarationen, die in der Spezifikation für Applikationen genutzt werden, sollen keine spezielle Security Policy implizieren. Produkt Provider können die Technologie anbieten, die benötigt wird, um gewünschte Security Policies unter Beachtung der Spezifikation zu implementieren und zu administrieren.

6. Abstraktion: Die Sicherheitsanforderungen einer Applikationskomponente werden im Deployment Descriptor logisch spezifiziert. Diese Informationen im Deployment Descriptor spezifizieren, wie die Sicherheitsrollen und Zugriffsanforderungen auf die spezifischen Sicherheitsrollen, Benutzer und Policies abzubilden sind. Der Deployer kann die Sicherheitseigenschaften passend zur operativen Umgebung modifizieren. Der Deployment Descriptor sollte dokumentieren, welche Sicherheitseigenschaften modifiziert werden können und welche nicht.

7. Unabhängigkeit: Die geforderten Sicherheitskonzepte sollten in einer Vielzahl von populären Sicherheitstechnologien implementierbar sein. Die Spezifikation schreibt keine spezielle Sicherheitstechnologie[1] vor. Die Spezifikation fordert nicht, dass das die geforderten Sicherheitskonzepte universell, unter der Benutzung einer oder aller Sicherheitstechnologien, implementierbar sein müssen.

8. Testbarkeit der Kompatibilität: Die Sicherheitsanforderungen der J2EE-Spezifikation müssen in einer Weise formuliert werden, die eine eindeutige Bestimmung der Kompatibilität einer Implementation ermöglicht.

Anmerkung: Ein Kompatibilitätstest bzw. eine Zertifizierung eines J2EE-Produkts kostet zur Zeit mehrere Millionen Dollar!

[1] wie zum Beispiel: Kerberos, PK, NIS+, oder NTLM

9. Sichere Interoperabilität: Applikationskomponenten, die in einem J2EE-Produkt ausgeführt werden, müssen mit Applikationskomponenten, in einem J2EE-Produkt eines anderen Herstellers, kommunizieren können. Außer die Security Policy verbietet dies.

Anmerkung: Sicherheit wird nicht explizit als Ziel definiert! Viele Anforderungen implizieren zwar Sicherheitsanforderungen (wie z.B. Skalierbarkeit, die eine gewisse Verfügbarkeit impliziert), jedoch scheint Sicherheit kein primäres Designziel zu sein.

Diese Annahme wird durch Aussagen der J2EE-Spezifikation bestätigt. Wie zum Beispiel:
"This specification does not provide any warranty or assurance of the effective security of a J2EE product."

Da Sicherheit eine Systemeigenschaft darstellt, muss sie in der Spezifikation des Systems definiert wurden. Dies bedeutet, dass ein solches System nie die Stufe B2 des Orange Books (TCSEC) erreichen kann.

4.3 Authentisierung

In der Literatur sind die Begriffe Authentifizierung, Authentisierung, Authentifikation und Authentication zu finden. Folgende Definitionen findet sind im Internet unter [KESINFO]:

> „... Die ursprünglich in Nuancen verschiedenen Begriffe "*Authentifizierung*" (eigentlich: Bezeugen der Echtheit), "*Authentisierung*" (eigentlich: Beglaubigung, Rechtsgültigmachung), "*Authentication*" (Eindeutschung des englischen authentication - Beglaubigung, Legalisierung) und "*Authentifikation*" (neue Wortschöpfung analog Authentifizierung) werden heute meist bedeutungsgleich nebeneinander gebraucht..."

Daher wird in dieser Arbeit der Begriff *Authentisierung* verwendet.

Authentisierung ist der Vorgang, bei dem die *Identität* einer *Entität* überprüft wird. In einer Umgebung mit verteilten Komponenten ist Authentisierung der Mechanismus, durch den Aufrufer (Caller) und Dienstanbieter (Service Provider) gegenseitig (oder auch nur einseitig) die Identität ihres Kommunikationspartners überprüfen, d.h. ob sie zu Gunsten eines speziellen Benutzers oder Systems handeln.

Ist die Überprüfung bidirektional, so spricht man von *beidseitiger Authentisierung (mutual Authentication).*

Eine Entität, die an einem Aufruf beteiligt ist und deren Identität nicht überprüft wurde, also anonym ist, wird mit *nicht authentisiert* bezeichnet.

Wenn ein *Client-Programm*, das durch einen Benutzer ausgeführt wird, einen Aufruf (Call) tätigt, so ist die Identität des Aufrufers i.d.R. die des Benutzers.

Ist der Aufrufer eine *Applikationskomponente*, die als Mittler (*Intermediär*) in einer Kette von Aufrufen handelt, welche von einem Benutzer initiiert wurde, so kann die Identität der Applikationskomponente die des Benutzers sein. In diesem Fall spricht man von einer *Personifikation* des ursprünglichen Aufrufers durch die Applikationskomponente. Alternativ kann die Komponente auch eine weitere Komponente mit ihrer eigenen Identität rufen, welche in keiner Beziehung zu der Identität ihres Aufrufers steht.

Eine Authentisierung wird häufig in zwei Phasen durchgeführt. Zuerst wird ein *Authentisierungskontext* etabliert, indem eine vom Dienst unabhängige Authentisierung durchgeführt wird, welche das Wissen eines *Geheimnisses (Secret)* erfordert. Dieser *Authentisierungskontext* enthält die Identität und ist in der Lage, *Authentikatoren* (Beweise der Authentisierung) zu produzieren. Dann wird der Authentisierungskontext zur Authentisierung gegenüber anderen (aufrufender oder gerufener) Entitäten genutzt.

Eine Grundlage der Authentisierung ist die Kontrolle über den Zugriff auf den Authentisierungskontext, wodurch eine Authentisierung mit der assoziierten Identität möglich wird.

Unter den möglichen Policies und Mechanismen zur Kontrolle des Zugriffs auf einen Authentisierungskontext sind:

- Ist ein Benutzer authentisiert, so erbt jeder Prozess, den der Benutzer startet, das Zugriffsrecht auf den Authentisierungskontext des Benutzers.

- Ist eine Komponente authentisiert, so kann der Zugriff auf ihren Authentisierungskontext für verwandte oder vertraute (trusted) Komponenten, solche die Teil der gleichen Applikation sind, möglich sein.

- Wenn eine Komponente einen Aufrufer personifizieren soll, so kann der Aufrufer seinen Authentisierungskontext an die gerufene Komponente *delegieren*.

Zeitpunkt der Authentisierung

Jede Authentisierung verursacht Kosten (Netzwerkverkehr, Datenbankzugriffe, etc.) und ist mit einem Sicherheitsrisiko verbunden, da ein Geheimnis (z.B. Passwort des Benutzers) über ein Medium eingegeben und/oder über das Netzwerk übertragen wird, wobei es abgehört werden kann.

Deshalb verlässt bei Kerberos das Passwort niemals den Rechner des Clients und ist auch dort nur einmal während der Eingabe (z.B. durch Trojaner, etc.) angreifbar.

Neben den Kosten und den Sicherheitsrisiken einer Authentisierung spielt auch noch die Benutzerfreundlichkeit eine Rolle, denn ein einmaliger Login ist angenehmer als mehrere Logins.

Mehrere Konzepte und Mechanismen der J2EE-Plattform können die Anzahl der Authentisierungsvorgänge extrem reduzieren. Dies führt aus Sicht des Benutzers zu einem einzigen *Single Sign-On.* Die J2EE-Plattform unterstützt Single Sign-On, so dass eine Authentisierung nur beim Überschreiten von Security Policy Domains nötig wird.

Dies wird primär ermöglicht durch:

- Lazy Authentication (verzögerte Authentisierung) bei Web-Ressourcen

- Single Sign-On (Web- oder Applicationclient)

- Login Sessions (Session Management)

- Propagierung der Identität

- Protection Domains (Trust-Beziehungen)
- Security Policy Domains (Security Policy-Beziehungen)

Die einzelnen Begriffe werden im folgenden Kapitel genauer erklärt.

4.3.1 Protection Domains

Eine *Protection Domain* ist eine Menge von Entitäten, die sich gegenseitig kennen und vertrauen. Entitäten in einer solchen Protection Domain brauchen sich daher nicht gegenseitig zu authentisieren. Interagiert eine Komponente mit anderen, in der gleichen Protection Domain, unterliegt die Identität des Aufrufers keinen weiteren Beschränkungen. Eine Authentisierung ist nur erforderlich, wenn die Domaingrenzen überschritten werden.

Der Aufrufer kann die Identität seines Aufrufers *propagieren* oder eine Identität wählen, die auf dem Wissen über von gerufenen Komponenten verhängten Beschränkungen basiert, da die Fähigkeit des Aufrufers eine Identität zu beanspruchen auf Vertrauen basiert und nicht auf Authentisierung.

Um die Grenzen einer Protection Domain zu etablieren, ist es notwendig, dass keinen nicht überprüften Identitäten über die Domaingrenzen hinweg vertraut wird. Entitäten, die grundsätzlich allen anderen Entitäten vertrauen, sollte nicht vertraut werden.

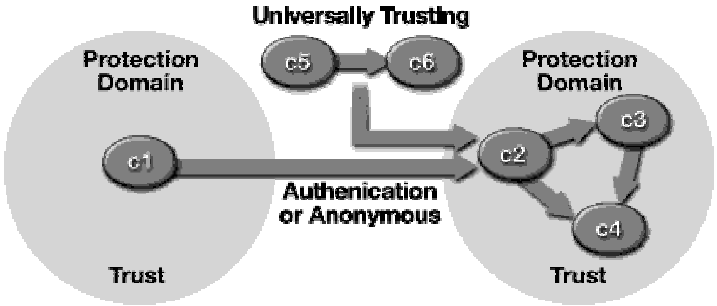

Abbildung 33: Protection-Domains aus [DEA2EE]

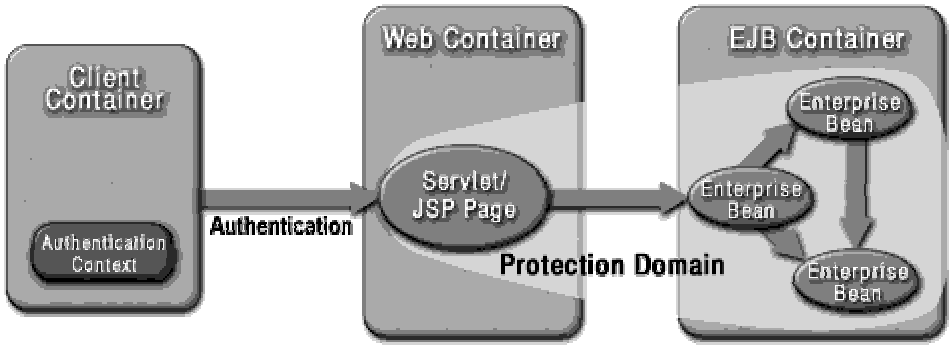

Abbildung 34: Typische J2EE-Applikationskonfiguration aus [DEAJ2EE]

In der J2EE-Architektur bildet der Container eine Authentisierungsgrenze zwischen externen Aufrufern und den Komponenten, die er hostet. Dies muss allerdings nicht so sein und kann je nach Implementation variieren. Eine Protection Domain kann mehrere Container überspannen oder ein Container kann mehrere Protection Domains beherbergen.

Der Container ist verpflichtet, für eingehende Aufrufe eine authentische Repräsentation der Identität einer Komponente in Form eines *Credentials* (z.B. ein X.509 Zertifikat oder ein Kerberosticket) zu generieren. Der Container ist bei ausgehenden Aufrufen verantwortlich, die Identität der rufenden Komponente festzustellen. Im allgemeinen ist es die Aufgabe des Containers, eine bidirektionale Authentisierungsfunktionalität zu bieten und so die Grenzen der Protection Domains zu überwachen.

Ohne eine eigene Überprüfung der Identität einer Komponente müssen die interagierenden Container entscheiden, ob die Vertrauensstellung zwischen den Container ausreichend ist, um die vom Container angebotene Repräsentation der Identität der Komponente zu akzeptieren.

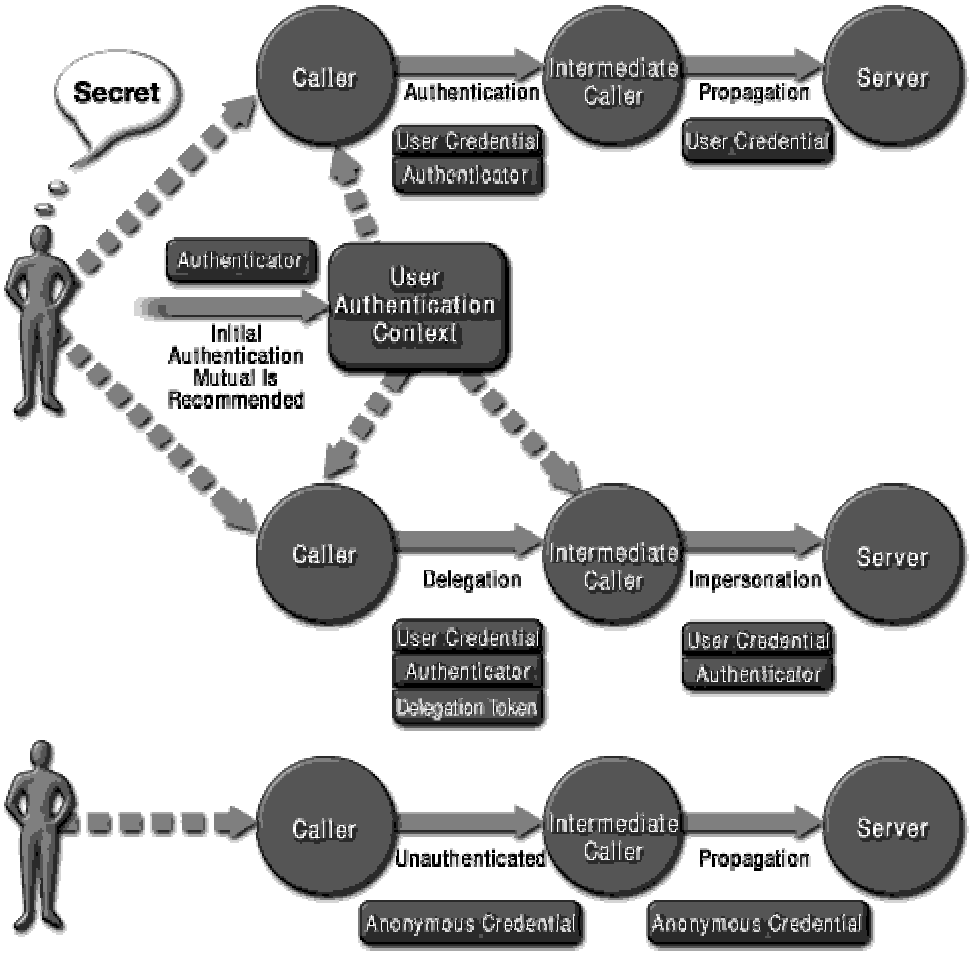

Abbildung 35: Authentisierungsszenarien aus [DEAJ2EE]

Abbildung 35 zeigt die Authentisierungskonzepte in zwei Szenarien: Ein Szenario mit einem authentisierten Benutzer und ein Szenario mit einem nicht authentisierten Benutzer.

Der authentisierte Benutzer ruft eine Komponente, die den Authentisierungskontext (Authentication Context) des Benutzers für den Beweis ihrer Identität gegenüber einem Intermediär (Intermediate Caller) nutzt. Wenn die gerufene Komponente (der Intermediär) einen Aufruf tätigt, so propagiert sie die Identität ihres Aufrufers. Die propagierte Identität wird nicht überprüft, sie wird von der Zielkomponente (Server) nur akzeptiert, wenn sich ihr Aufrufer (in diesem Fall der Intermediär) in der gleichen Protection Domain befindet.

Die Abbildung differenziert die Propagierung (Propagation) von der Delegation und der Personifizierung (Impersonation). Bei der Propagierung trägt der Dienstanbieter die Last, ob er die propagierte Identität als authentisch akzeptieren soll. Bei der Delegation gibt der Aufrufer der gerufenen Komponente Zugriff auf seinen Authentisierungskontext, was es der gerufenen Komponente ermöglicht, weitere Aufrufe im Namen des Aufrufers zu tätigen (Personifizierung). Die Personifizierung erfordert, dass der Benutzer dem Imitator (Impersonator) traut.

Der untere Teil der Abbildung zeigt die Propagierung einer nicht authentisierten Benutzeridentität in Form eines *Anonymous Credential*. Ein Anonymous Credential ist die Repräsentation einer nicht geprüften Identität, die unabhängig von Vertrauensstellungen propagiert wird.

Elsa Einstein: *"No, I don't understand my husband's theory of relativity, but I know my husband, and I know he can be trusted."*

4.3.2 Authentisierungsmechanismen

In einer typischen J2EE-Applikation verwendet der Benutzer einen Client-Container, um mit den Enterprise-Ressourcen in der Web- oder EJB-Schicht zu kommunizieren. Diese Ressourcen können geschützt oder ungeschützt sein. Geschützte Ressourcen erkennt man an der Präsenz von *Authentication Rules* (siehe Abschnitt Autorisierung), welche die Zugriffrechte auf eine Teilmenge von nicht-anonymen Identitäten beschränken. Um auf eine geschützte Ressource zu zugreifen, muss der Benutzer ein nicht anonymes *Credential* vorweisen, so dass die Identität bezüglich der Autorisierungspolicy ausgewertet werden kann. Ist eine Vertrauensbeziehung zwischen Client- und Ressourcen-Container nicht gegeben, so muss neben dem *Credential* ein *Authenticator*, welcher das Recht (einen Beweis) zur Beanspruchung der Identität des Benutzer darstellt, übergeben werden.

4.3.2.1 Lazy Authentication (Verzögerte Authentisierung)

Mit jeder Authentisierung sind Kosten verbunden. Denn bei jedem Authentisierungsprozess werden mehrere Nachrichten über eine Netzwerkstrecke verschickt und Serverressourcen beansprucht. Neben den Kosten ist mit jeder Authentisierung auch ein Sicherheitsrisiko verbunden, da vertrauliche Authentisierungsdaten über eine Netzwerkstrecke versandt werden.

Ein Mechanismus, bei dem eine Authentisierung nur im Bedarfsfall durchgeführt wird, ist die *Lazy Authentication* (Verzögerte Authentisierung). Hier wird ein Benutzer erst authentisiert, wenn er auf eine geschützte Ressource zugreift. Die verzögerte Authentisierung kann bei Clients (Applets und Applicationclients) der ersten Schicht (First-tier) verwendet werden und muss von Webservern für geschützte Webressourcen unterstützt werden.

4.3.2.2 Secure Association (Sichere Assoziation)

Eine *Secure Association* (SA) ist die Beziehung zwischen zwei oder mehreren Entitäten, die beschreibt, wie die Entitäten ihre sicherheitsrelevanten Dienste nutzen, um sicher zu kommunizieren. Diese Beziehung wird durch eine Menge von Informationen repräsentiert, die als Vertrag zwischen den Entitäten betrachtet werden kann. Die Informationen müssen zwischen allen Entitäten verteilt werden und alle Entitäten müssen diesen Informationen zustimmen.

Eine SA wird in der Regel in mehreren Schritten aufgebaut. Zunächst findet die Authentisierung der Entitäten statt. Anschließend wird die Qualität des Schutzes, wie Vertraulichkeit und Integrität, verhandelt. Abschließend wird ein *Security Context* etabliert, der einen Vertrag zwischen den authentisierten Entitäten darstellt.

Der Begriff SA wird häufig in Verbindung mit IP6 und IPSec verwendet. Eine SA lässt sich auch mit SSL realisieren. Die Spezifikation fordert für SAs keinen speziellen Mechanismus.

4.3.3 Authentisierung auf Ebene der Benutzer

Benutzerauthentisierung ist der Prozess, bei dem der Benutzer seine Identität gegenüber dem System beweist. Diese authentisierte Identität wird dann dazu genutzt, Autorisierungsentscheidungen für den Zugriff auf J2EE-Applikationskomponenten zu fällen. Der Endbenutzer kann unter Benutzung von Webclients oder Applicationclients authentisiert werden.

4.3.3.1 Benutzerauthentisierung mit Webclients

Webclients sind i.d.R. Webbrowser, die mit einem Web-Container interagieren. Der Webbrowser wird oft mit einem Java Plug-In kombiniert und enthält zusätzlich ein Applet-Container, in dem Applets laufen. Es wird gefordert, dass ein Webclient in der Lage sein muss, den Benutzer gegenüber dem Server zu authentisieren. Der Deployer oder Systemadministrator entscheidet, welche Methode bei einer Applikation oder einer Gruppe von Applikationen anzuwenden ist.

4.3.3.2 Benutzerauthentisierung mit Applicationclients

Applicationclients sind Clientprogramme, die in einem Applicationclient-Container ausgeführt werden. Sie können mit Enterprise Beans direkt interagieren, d.h. ohne die Hilfe eines Webbrowsers und ohne einen Webserver zu durchqueren. Sie können aber auch auf Web-Ressourcen zugreifen. Üblicherweise besitzen sie eine GUI und sind für die Kommunikation mit menschlichen Benutzern gedacht.

Applicationclients werden als Benutzerschnittstelle genutzt, um Endbenutzer gegenüber der J2EE-Plattform zu authentisieren, wenn die Benutzer auf geschützte Web-Ressourcen oder Enterprise Beans zugreift. Der Applicationclient-Container muss die Authentisierung von Benutzern der Applikation unterstützen, um die Authentisierungs- und Autorisierungsbeschränkungen, die von den EJB-Container und Web-Container durchgesetzt werden, zu erfüllen. Die Techniken, die dazu genutzt werden, variieren mit den Implementationen der Applicationclient-Container und befinden sich nicht unter der Kontrolle der Applikation. Der Applicationclient-Container kann in das Authentisierungssystem des J2EE-Produktes integriert sein, um Single Sign-On zu unterstützen. Die Authentisierung kann beim Start der Applikation oder verzögert, beim Aufruf einer geschützten Ressource, durchgeführt werden.

Applicationclients werden in einer, vom J2SE-Security Manager kontrollierten, Umgebung ausgeführt und sind Subjekt von Permissions. Eine minimale Menge dieser Permissions, die eine Applikation erwarten kann, sind in der J2EE-Spezifikation im Kapitel *„Java 2 Platform, Standard Edition (J2SE) Requirements“* näher beschrieben.

Die J2EE-Spezifikation definiert keine Beziehung zwischen der OS-Identität, die mit der laufenden Applikation assoziiert ist, und der Identität des authentisierten Benutzers. Die Unterstützung von Single Sign-On erfordert jedoch, dass das J2EE-Produkt in der Lage sein muss, eine Beziehung zwischen den beiden Identitäten herzustellen.

Der Container muss eine passende Benutzerschnittstelle zur Verfügung stellen, um Authentisierungsdaten zu erfassen. Zusätzlich kann ein Applicationclient die Implementation einer *Callback Handler* Klasse anbieten und im Deployment Descriptor spezifizieren, die z.B. ein angepasstes Login Prompt zur Verfügung stellt. Der Callback Handler der Applikation muss alle Callbackobjekte unterstützen, die im *javax.security.auth.callback* Package spezifiziert sind.

Der Deployer kann den, von der Applikation spezifizierten, Callback Handler überschreiben und statt dessen das vorgegebene Authentisierungsbenutzerinterface des Containers verwenden. Wenn der Deployer die Nutzung des Callback Handler konfiguriert hat, muss der Applicationclient-Container ein Objekt der Klasse erzeugen und es für alle Authentisierungsinteraktionen mit dem Benutzer verwenden.

Weitere Anforderungen an den Applicationclient werden in der Spezifikation beschrieben.

4.3.4 Authentisierung auf Ebene der Ressourcen

Ressourcen, wie zum Beispiel ein Enterprise Information System, werden oft in einer anderen Security Policy Domain deployed, als die der Applikationskomponenten. Die vielfältigen Authentisierungsmechanismen, die genutzt werden, um den Aufrufer einer Ressource zu authentisieren, führen zu Anforderungen, die beschreiben, welche Mittel ein J2EE-Produkt bieten muss, um die Authentisierung in der Security Policy Domain der Ressource durchzuführen.

Die J2EE-Spezifikation fordert von jedem J2EE-Produkt die folgenden Mechanismen:

1. Configured Identity (Run As): Beim Zugriff auf eine Ressource muss ein J2EE-Container einen Prinzipal und Authentisierungsdaten zur Authentisierung verwenden können, die ein Deployer, zur Deploymentzeit, festgelegt hat. Die Authentisierung darf nicht von den Daten abhängen, die Applikationskomponenten bereitstellten. Die sichere Speicherung von vertraulichen Authentisierungsinformationen liegt in der Verantwortlichkeit des Product Providers.

2. Programmatic Authentication: Jedes J2EE-Produkt muss den Applikationskomponenten ein API zur Verfügung stellen, mit dem sie zur Laufzeit einen Prinzipal und Authentisierungsdaten für Zugriffe auf Ressourcen spezifizieren können. Die Komponenten können den Prinzipal und die Authentisierungsdaten durch vielfältigste Mechanismen erhalten, zum Beispiel durch Parameterübergabe oder aus der Umgebung der Komponenten.

Zusätzlich empfiehlt die J2EE-Spezifikation die folgenden Mechanismen, die aber keine expliziten Anforderungen darstellen:

3. Principal Mapping: Der Prinzipal und die Sicherheitsattribute einer Ressource kann durch eine Abbildung (Mapping) der Daten des aufrufenden Prinzipals festgelegt werden. Der Prinzipal und die Sicherheitsattribute der Ressource basieren in diesem Fall nicht auf einer Vererbung, sondern auf einer Abbildung der Identität des rufenden Prinzipals.

4. Caller Impersonation: Der Prinzipal einer Ressource handelt im Namen des rufenden Prinzipals. Dies erfordert eine Delegation der Identität und Credentials des Aufrufers zu einem darunter liegenden Ressource Manager. In einigen Szenarien kann der rufende Prinzipal ein Delegierter eines anderen initiierenden Prinzipals sein und der Prinzipal der Ressource verkörpert so transitiv den initiierenden Principal. Die Unterstützung von Delegation

ist üblicherweise spezifisch für einen Sicherheitsmechanismus. Zum Beispiel unterstützt Kerberos einen Mechanismus zur Delegation der Authentisierung.

5. Credentials Mapping: Der Applikationsserver bildet Credentials eines Typs auf einen anderen Typ ab. Diese Technik wird genutzt, wenn der Applikationsserver und das EIS verschiedenen Security Technology Domains angehören. Zum Beispiel besitzt der authentisierte initialisierende Principal ein Public-Key zertifikat-basiertes Credential. Die Sicherheitsumgebung des Ressource-Managers ist mit einem Kerberos-Authentisierungsdienst konfiguriert. In diesem Fall bildet der Applikationsserver die Public-Key zertifikat-basierten Credentials auf Kerberos-Credentials ab.

Zusätzliche Informationen zu den Anforderungen an die Ressourcenauthentisierung finden sich in der Connector Spezifikation.

4.3.5 Authentisierung auf Ebene der Web-Container

Ein Application Component Provider kann eine Sammlung von Web-Ressourcen (Webkomponenten, HTML-Dokumenten, Bilddateien, gepackte Archive, u.s.w.) als geschützt deklarieren, indem er im Deployment Descriptor ein *Authorization Contraint* für diese Sammlung spezifiziert. Wenn nun ein nicht authentisierter Benutzer versucht, auf eine geschützte Web-Ressource zuzugreifen, wird der Web-Container den Benutzer auffordern, sich ihm gegenüber zu authentisieren. Der Web-Container wird die Anfrage des Benutzers erst beantworten, wenn die Identität und Zugriffrechte des Benutzers überprüft wurden.

4.3.5.1 Authentisierungsproxy

Ein Webclient kann einen Webserver als *Authentisierungsproxy* einsetzen. In diesem Fall wird ein Credential des Clients im Webserver etabliert (Delegation). Dort kann es vom Server für viele verschiedene Zwecke eingesetzt werden, wie zum Beispiel für Autorisierungsentscheidungen, Personifizierung des Clients gegenüber EJBs oder um Secure Associations mit Ressourcen zu verhandeln. Derzeitige Webbrowser beruhen im allgemeinen auf der Proxy Authentisierung.

4.3.5.2 Login Sessions und Web Single Sign-On

Die J2EE-Spezifiktion fordert von Web-Containern eine Unterstützung von *Login Sessions* und *Web Single Sign-On*. Wenn ein Benutzer erfolgreich von einem Webserver authentisiert wurde, etabliert der Container einen *Login Session Context* für den Benutzer. Dieser Login Session Context beinhaltet Credentials, die mit dem Benutzer assoziiert sind. Der Container nutzt die Credentials, um einen *Security Context* für die Session zu etablieren. Der Security Context wird vom Container für Autorisierungsentscheidungen genutzt und um Secure Associations mit anderen Komponenten (inklusive Enterprise Beans) zu etablieren.

Da der Client bezüglich der Authentisierung zustandslos ist, ist es nötig, dass der Server als sein Authentisierungsproxy handelt und seinen *Login Session Context* verwaltet. Eine Referenz auf den Zustand der Login Session wird dem Client, durch Cookies oder URL-Rewriting, zur Verfügung gestellt.

Wird SSL mit gegenseitiger Authentisierung (SSL mutual authentication) als Authentisierungsprotokoll eingesetzt, kann der Client seinen eigenen Login Context verwalten und ist damit nicht mehr abhängig von Referenzen auf den Zustand seiner Login Session.

Jeder J2EE-Webserver muss für jeden Web-Benutzer eine Login Session bereitstellen. Einer Login Session muss es möglich sein, sich über mehrere Applikationen zu erstrecken, dies erlaubt dem Benutzer, nach einem Login auf mehrere Applikationen zuzugreifen. Login-Sessions unterstützen *Web Single Sign-On*, so dass eine Authentisierung nur beim Überschreiten von Security Policy Domains nötig wird.

Bei der Verwendung von Login Sessions bleiben Applikationen unabhängig von den Implementationsdetails und der Verwaltung der Login-Informationen. Der J2EE Product Provider hat die Flexibilität, einen Authentisierungsmechanismus unabhängig von den Applikationen, die durch diesen Mechanismus geschützt werden, zu wählen.

Die Unterstützung von Login Sessions wird, laut J2EE-Spezifikation, in der Servlet-Spezifikation detailliert beschrieben. Allerdings taucht der Begriff „Login Session“ in der Servlet-Spezifikation nicht auf!

4.3.5.3 Anonymer Zugriff auf Web-Komponenten

Im Internet wird üblicherweise „anonym“ auf Web-Ressourcen zugegriffen. Daher ist eine Authentisierung bei ungeschützten Web-Ressourcen nicht erforderlich. Zur Kennzeichnung vergibt der Container einen *Anonymous Credential*. Eine Authentisierung wird erst notwendig, wenn auf Ressourcen zugegriffen wird, die im Deployment Descriptor entsprechend konfiguriert wurden. EJB-Ressourcen fordern hingegen immer ein nicht anonymes Credential.

Ein Web-Container signalisiert, dass kein Benutzer authentisiert wurde, indem er *null* zurückgibt, wenn die *HttpServletRequest* Methode *getUserPrincipal* gerufen wird.

Bei EJB-Containern ist dies nicht der Fall**.** Die EJB-Spezifikation fordert, dass die E*JBContext* Methode *getCallerPrincipal* immer ein gültiges Prinzipalobjekt zurückgibt.

Web-Komponenten im Web-Container müssen jedoch in der Lage sein, Enterprise Beans zu rufen, auch wenn kein Benutzer vom Web-Container authentisiert wurde. Deshalb fordert die Spezifikation von jedem Web-Container, dass er für diese Fälle ein gültiges Prinzipalobjekt liefert. Dies ermöglicht die *run-as* Funktion des Web- und EJB-Containers. Mit ihr lassen sich im Deployment Descriptor entsprechende Prinzipalobjekte definieren. Es ist zu beachten, dass die EJB-Spezifikation besondere Anforderungen enthält, die das EJB Interoperability Protokoll betreffen.

4.3.5.4 Geforderte Login-Mechanismen

Von allen J2EE-Produkten wird gefordert, dass sie vier Login-Mechanismen unterstützen:

- Einfache HTTP Authentisierung (*HTTP basic authentication*)
- HTTPS Client Authentisierung (*HTTPS client authentication*)
- SSL mit beidseitiger Authentisierung (*SSL mutual authentication*)
- Formularbasierter Login (*form-based login*)

Von den J2EE-Applikationen wird nicht gefordert, einen dieser Mechanismen zu nutzen, aber es wird gefordert, dass diese Mechanismen für jede Applikation verfügbar sind.

4.3.5.4.1 Anforderung: Einfache HTTP Authentisierung

Die *einfache HTTP Authentisierung* (RFC2068) ist ein Authentisierungsmechanismus, der vom HTTP-Protokoll unterstützt wird. Als Teil einer Anfrage übergibt der Webserver einen *Realm* (Bereichsbezeichnung), in dem der Benutzer zu authentisieren ist. Der Webclient erhält dann vom Benutzer den Benutzernamen und Passwort und überträgt es zum Webserver. Der Webserver authentisiert anschließend den Benutzer in dem spezifizierten Realm (in der J2EE-Spezifikation HTTP-Realm genannt). Die Servlet-Spezifikation weißt darauf hin, dass der Realm String keine spezielle Security Policy Domain reflektieren muss, die irreführenderweise auch Realm genannt wird.

Die einfache HTTP Authentisierung ist nicht sicher! Passwörter werden in einfacher Base64 Codierung gesendet und der Zielserver wird nicht authentisiert. Zusätzliche Schutzmechanismen können angewendet werden, um diese Schwäche zu umgehen. Das Passwort kann geschützt werden, indem man Sicherheitsmechanismen auf der Transportschicht (wie z.B. HTTPS) oder Netzwerkschicht (z.B. IPSEC oder VPN) anwendet.

Von jedem J2EE-Produkt wird gefordert, dass es die einfache HTTP Authentisierung nach RFC 2068 unterstützt. Von J2EE-Plattform Providern wird zusätzlich gefordert, dass die einfache HTTP Authentisierung auch über SSL unterstützt wird.

Trotz der Limitierungen, wird in der J2EE-Spezifikation auf die einfache HTTP Authentisierung eingegangen, da sie in vielen weit verbreiteten Applikationen genutzt wird. Konfiguriert wird dieser Authentisierungsmechanismus im Deployment Descriptor:

```
<web-app>
        <login-config>
                <auth-method>BASIC</auth-method>
                <realm-name>default</realm-name>
        </login-config>
</web-app>
```

4.3.5.4.2 HTTP Digest Authentication

Auch bei der HTTP Digest Authentication wird der Benutzer auf der Basis eines Benutzernamens und Passwortes authentisiert. Das Passwort wird bei diesem Verfahren jedoch verschlüsselt übertragen. Dazu wird der Hashwert der Verkettung von HTTP Request Nachricht und Passwort berechnet, der dann zusammen mit der Request Nachricht an den Webserver übertragen wird. Dieses Verfahren ist um einiges sicherer als die einfache HTTP Authentisierung. Reply-Attacken und Geburtstagsangriffe auf den Hashwert sind dennoch möglich. HTTP Digest Authentication ist nicht weit verbreitet und wird daher von der J2EE-Spezifikation für nicht erforderlich gehalten. Die Servlet-Spezifikation definiert für dieses Verfahren ein entsprechendes Element zur Deklaration im Deployment Descriptor:

```
<web-app>
        <login-config>
                <auth-method>DIGEST</auth-method>
        </login-config>
</web-app>
```

4.3.5.4.3 HTTPS Client Authentisierung (HTTPS client authentication)

Die *HTTPS Client Authentisierung* ist ein zertifikatbasiertes Authentisierungsverfahren. SSL sichert das HTTP Protokoll und unterstützt den Zertifikatsaustausch. Der Benutzer muss ein *Public Key Certificate* (PKC) präsentieren, anhand dessen er authentisiert wird. Dieses Verfahren wird zur Zeit noch selten von Endbenutzern im Internet verwendet, doch aufgrund seiner Nützlichkeit für e-Commerce Applikationen und Single Sign-On wird es von der J2EE-Applikation für jedes J2EE-Produkt gefordert. Das Authentisierungsverfahren wird im Deployment Descriptor definiert:

```
<web-app>
        <login-config>
                <auth-method>CLIENT-CERT</auth-method>
        </login-config>
</web-app>
```

4.3.5.4.4 SSL mit beidseitiger Authentisierung

Ähnlich der HTTPS Client Authentisierung handelt es sich bei SSL mit beidseitiger Authentisierung um einen zertifikatsbasierten Authentisierungsmechanismus (X.509), der Funktionalitäten von SSL nutzt. Mit dem Unterschied, dass beide Kommunikationspartner (Server und Client) sich gegenseitig authentisieren. Die J2EE-Spezifikation fordert jedem J2EE-Produkt eine Unterstützung von SSL 3.0 nach der Spezifikation [SSL30] und eine Unterstützung der beidseitigen zertifikatbasierten Authentisierung. Ein entsprechendes Element, zur Kennzeichnung im Deployment Descriptor, wird nicht definiert. Es ist anzunehmen, dass es sich hier um eine Inkonsistenz der Spezifikation handelt. SSL bietet die Möglichkeit zur Client-, Server und beidseitigen Authentisierung. Der Webserver lässt sich mit einem entsprechenden Zertifikat zur einseitigen Authentisierung des Servers konfigurieren. Mit einer zusätzlichen Definition der HTTPS Client Authentisierung im Deployment Descriptor, hat man insgesamt eine beidseitige Authentisierung konfiguriert.

Alle J2EE-Produkte müssen folgende Algorithmen (cipher suites) unterstützen, um die Interoperabilität der Authentisierung mit allen Clients sicherzustellen:

- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

Diese Algorithmen werden von den meisten Webbrowsern unterstützt und entsprechen den Exportvorschriften der U.S. Regierung. Für den professionellen Einsatz in einem Unternehmen sind die geforderten Algorithmen allerdings zu schwach.

Die EJB-Spezifikation fordert ebenfalls SSL3.0 und TLS1.0 sowie die Algorithmen:

- TLS_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_RC4_128_MD5
- TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA [38]
- SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
- TLS_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

4.3.5.4.5 Formular-basierter Login

Die Authentisierungsmechanismen können entweder den vom Webbrowser implementierten Login-Screen verwenden, was eventuell das Look-and-Feel der Applikation stören könnte, oder einen eigen formular-basierten Login-Screen bieten. Dazu führt die J2EE-Spezifikation die Möglichkeit ein, HTML oder Servlet/JSP basierte Formulare für den Login-Screen zu verwenden, wodurch eine Anpassung des Benutzerinterfaces möglich wird. Der formular-basierte Authentisierungsmechanismus wird ausführlich in der Servlet-Spezifikation beschrieben. Die Konfiguration findet über den Deployment Descriptor der Webapplikation statt. Hier wird das zu verwendende Formular vom Deployer spezifiziert:

```
<web-app>
        <login-config>
                <auth-method>FORM</auth-method>
                        <form-login-config>
                                <form-login-page>login.jsp</form-login-page>
                                <form-error-page>error.jsp</form-error-page>
                        </form-login-config>
        </login-config>
</web-app>
```

Die formularbasierte Authentisierung hat, bezüglich der Sicherheit, die gleichen Schwächen wie die einfache HTTP Authentisierung, da das Passwort des Benutzers im Klartext übertragen und der Zielserver nicht authentisiert wird. Einen zusätzlichen Schutz könnte ein sicherer Transportmechanismus (HTTPS) oder Sicherheitsmechanismen auf Netzwerkebene (wie das IPSEC Protokoll oder VPN Strategien) bieten.

Die Servlet-Spezifikation weist noch darauf hin, dass ein Session-Management notwendig ist. Dies sollte durch Cookies oder SSL-Sitzungsinformationen geschehen, da ein URL-basiertes Session-Tracking schwer zu implementieren sei.

4.3.5.5 Hybride Formen der Authentisierungsmechanismen

Wie bereits erwähnt, haben einige Authentisierungsmechanismen ihre Schwächen, da das Passwort im Klartext übertragen wird. Besser ist eine Kombination dieser Mechanismen mit SSL. Man spricht dann von einem Hybrid. Die Deklaration findet sich im Deployment Descriptor:

```
web-app>
<security-constraint>
...
        <user-data-constraint>
                <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
</security-constraint>
</web-app>
```

4.3.6 Authentisierung auf Ebene der EJB-Container

4.3.6.1 Sichere Interoperabilität (Secure Interoperability)

Die EJB-Spezifikation beschreibt Interoperabilitätsmechanismen, die sichere Aufrufe von Enterprise JavaBeans im Intranet unterstützen. Alle diese Mechanismen basieren auf dem *CORBA/IIOP Protokoll*. Mechanismen, die eine sichere Interoperabilität gewährleisten, sollten auch weit verbreitete Technologien unterstützen wie, zum Beispiel den kerberos-basierte Secret-Key Mechanismus und den X.509 zertifikatbasierte Public-Key Mechanismus.

In früheren Versionen forderte die J2EE-Spezifikation vom EJB-Container keine spezielle Implementierung der Authentisierungsmechanismen. Dies hat sich mit den Versionen J2EE 1.3 und EJB 2.0 geändert. Man vertraute auf Firewalltechnologien, die eine direkte Kommunikation (über RMI) von Client-Containern und Enterprise Beans verhindern sollten. Die Authentisierung überließ man dem Web-Container.

Die J2EE 1.3 Spezifikation fordert von EJB-Containern und EJB-Client-Containern, dass sie Version 2 des *Common Secure Interoperability* (CSIv2) Protokolls unterstützen. CSIv2 ist ein Standard der *Object Management Group* (OMG). Es handelt sich um ein Protokoll für sichere Aufrufe über RMI-IIOP. CSIv2 wurde für Umgebungen entworfen, in denen die Sicherung der Integrität und Vertraulichkeit von Nachrichten sowie die Authentisierung von Clients und Servern auf der Transport-Schicht (zum Beispiel durch SSL/TLS) durchgesetzt wird.

Zusätzlich wird noch das *Security Attribute Service* (SAS) Protokoll definiert. SAS arbeitet über der Transportschicht und dient u.a. zur Clientauthentisierung oder zur Personifizierung. Der Mechanismus der Personifizierung (Impersonation), *Identity Assertion* genannt, macht es einem Intermediär möglich, eine andere Identität anzunehmen. Damit ermöglicht Identity Assertion, einem intermediären Container die Identitäten seiner Aufrufer für seine Rufe zu verwenden. J2EE-Container nutzen den CSIv2 Identity Assertion Mechanismus zur Etablierung der Identitäten, die von Komponenten genutzt werden, um andere Komponenten, entsprechend der Definitionen des Application Deployers, zu rufen. Abbildung 36 zeigt diese Architektur.

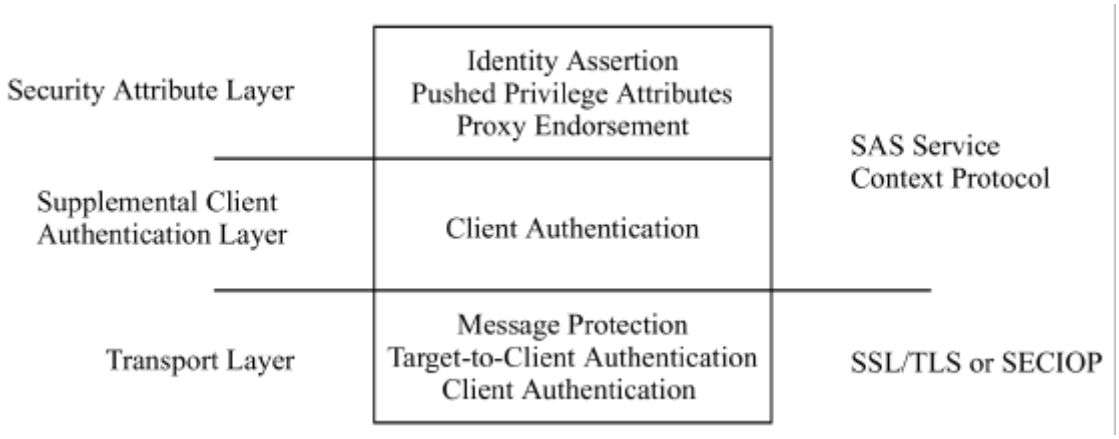

Abbildung 36: CSIv2 Protokoll Architektur aus [CORBA30]

CSIv2 definiert noch eine Sprache für Applikationsserver, die zur Kommunikation von Sicherheitsanforderungen gegenüber Clients dient. Applikationsserver nutzen diese Sprache in ihren *Interoperable Object References (IORs)*, so dass die Sicherheitsanforderungen den Clients zur Verfügung stehen.

Diese Anforderungen werden für jede CSIc2-Schicht definiert. Wobei jede Schicht eine Kombination aus unterstützter und geforderter Sicherheitsfunktionalitäten definiert, die von Clients erfüllt werden müssen. Diese Anforderungen muss der Deployer während des Deployments in einer CSIv2 Security Policy definieren. Die Hauptpunkte in dieser Policy sind Definitionen von Anforderungen an Integrität, Vertraulichkeit und Authentisierungsverfahren.

4.3.6.2 HTTP Login Gateways

Eine Alternative zu CSIv2 stellen so genannte *HTTP Login Gateways* dar. Die Kommunikation findet in diesem Fall über eine, mit SSL gesicherte, HTTP-Verbindung statt und zur Authentisierung wird SSL mit beidseitiger Authentisierung eingesetzt. Zum Informationsaustausch kann HTML, XML oder ein anderes strukturiertes Format verwendet werden.

4.4 Autorisierung

Autorisierungsmechanismen limitieren die Interaktionen mit Ressourcen auf eine Teilmenge von Benutzern oder Systemen, um Beschränkungen zur Sicherung von Integrität, Vertraulichkeit oder Verfügbarkeit durchzusetzen. Solche Mechanismen erlauben nur authentisierten und autorisierten Identitäten auf Komponenten zuzugreifen.

Die Zugriffskontrollmechanismen der J2EE-Plattform autorisieren den rufenden Programmcode nach seinem Herkunftsort, seinen Unterzeichnern und der Identität des Benutzers, unter dessen Identität der Programmcode ausgeführt wird.

Wie schon im Abschnitt Authentisierung beschrieben, kann die Identität eines Aufrufers etabliert werden, indem der Code eine Identität, aus den ihm verfügbaren Authentisierungskontexten, auswählt. Alternativ kann der Aufrufer die Identität seines Aufrufers propagieren, eine freiwählbare Identität wählen oder den Aufruf anonym durchführen. In jedem dieser Fälle übergibt der Aufrufer der gerufenen Komponente ein *Credential.*

Dieses Credential beschreibt den Aufrufer durch *Identitätsattribute*. Im Falle des anonymen Aufrufers wird ein besonderes Credential genutzt. Die Attribute identifizieren den Aufrufer eindeutig im Kontext der Autorität, die dieses Credential ausgestellt hat. Abhängig vom Typ des Credentials kann es weitere Attribute besitzen, welche gemeinsam genutzte Autorisierungseigenschaften darstellen, wie z.B. einer Gruppenzugehörigkeit, die ähnliche Credentials kennzeichnet. Die *Identität* und die *gemeinsam benutzten Autorisierungsattribute* in einem Credential werden die *Sicherheitsattribute* (*Security Attributes*) des Aufrufers genannt.

In der J2SE Plattform zählen die Identitätsattribute des vom Aufrufer genutzten Codes ebenfalls zu seinen Sicherheitsattributen. Der Zugriff auf eine gerufene Komponente wird durch den Vergleich der Sicherheitsattribute des Aufrufers mit denen, die erforderlich sind, um auf die Komponente zuzugreifen, geregelt.

In der J2EE-Architektur dient der Container als *Autorisierungsgrenze* zwischen den Komponenten, die er beherbergt, und deren Aufrufern. Die *Authentisierungsgrenze* befindet sich innerhalb der *Autorisierungsgrenze* des Containers, so dass Autorisierung im Kontext einer erfolgreichen Authentisierung betrachtet wird. Für eingehende Rufe vergleicht der Container die Sicherheitsattribute aus dem Credential des Aufrufers mit den *Zugriffskontrollregeln (Access Control Rules)* der Zielkomponente. Sind die Regeln erfüllt, so wird der Aufruf gestattet.

Es gibt zwei grundlegende Konzepte, um Zugriffskontrollregeln zu definieren: *Capabilities* und *Permissions*. Capabilities beschreiben, wozu jemand berechtigt ist. Permissions beschreiben, wer zugreifen darf.

In dem J2EE-Applikationsprogrammiermodell ist es die Aufgabe des Deployers, das Permission-Modell der Applikation auf die Capabilities der Benutzer in der operativen Umgebung abzubilden.

4.4.1 Deklarative Sicherheit

Deklarative Sicherheit bezieht sich auf den, außerhalb der Applikation lokalisierten, Ausdruck der Sicherheitsstruktur (Security Policy) einer Applikation, einschließlich der Sicherheitsrollen (Security Roles), Zugriffskontroll- und Authentisierungsanforderungen. Diese Anforderungen werden *klassen-* bzw. *typbasiert* im Deployment Descriptor definiert. Er ist ein Vertrag zwischen dem Application Component Provider, dem Application Assembler und dem Deployer.

Der Application Component Provider und Application Assembler können Gruppen von Komponenten mit einem Deployment Descriptor assoziieren. In ihm definieren sie die Sicherheitsanforderungen der Applikation. Die Zugriffsrechte sind logische Privilegien, die durch Sicherheitsrollen (*Security Roles*) definiert werden:

```
<security-role>
        <role-name>manager</role-name>
</security-role>
```

Ein Deployer bildet, mit einem Deployment Tool, die aus dem Deployment Descriptor hervorgehende Repräsentation der Security Policy der Applikation auf die Sicherheitsstruktur der jeweiligen Umgebung ab, indem er die logischen Privilegien (Security Roles) auf die Benutzer der operativen Umgebung abbildet. Zum Beispiel könnte ein Deployer eine Sicherheitsrolle auf eine Benutzergruppe oder Liste, mit einem oder mehreren Prinzipalen, der operativen Umgebung abbilden. Als Ergebnis würde jedem Aufrufer, dessen Sicherheitsattribute seine Mitgliedschaft in der Gruppe oder Liste kennzeichnen, das Privileg, das durch die Rolle repräsentiert wird, zugeordnet werden.

Aus dem vom Deployer konfigurierten Deployment Descriptor lässt sich die jeweilige Sicherheitsstruktur (Security Policy) ableiten. Diese nutzt der Container, um Authentisierungs- und Autorisierungsanforderungen durchzusetzen. Er ordnet Aufrufern logische Privilegien (Rollen) zu, die auf den Werten der Sicherheitsattribute des Aufrufers beruhen. Der EJB-Container vergibt das Recht für den Zugriff auf eine Methode nur an den Aufrufer, der mindestens eines der Privilegien, welche der Methode zugeordnet sind, besitzt.

Sicherheitsrollen schützen auch Sammlungen von Web-Ressourcen, die durch ein URL-Muster und eine assoziierte HTTP-Methode, wie zum Beispiel *GET*, definiert werden. Der Web-Container setzt die Autorisierungsanforderungen ähnlich wie der EJB-Container durch.

4.4.2 Programmatic Security

Das entscheidende Merkmal der *Programmatic Security*[1] ist, dass im Programmcode eines Exemplars einer Komponente zusätzliche sicherheitsrelevante Logik codiert ist. In der J2EE-Plattform ist dies die einzige Möglichkeit *exemplarbasierte* Zugriffskontrollentscheidungen durchzusetzen.

Ein J2EE-Container fällt seine Zugriffskontrollentscheidungen vor der Weiterleitung der Methodenaufrufe an die Komponenten. Die Programmlogik und der Zustand der Komponente spielen bei diesen Zugriffskontrollentscheidungen keine Rolle. Programmatic Security bezieht sich auf sicherheitsrelevante Entscheidungen, die von den Applikationen selbst getroffen werden.

Programmatic Security ist nützlich, wenn deklarative Sicherheit alleine nicht ausreichend ist, um das Sicherheitsmodell einer Applikation auszudrücken. Die J2EE-Spezifikation fordert von den Containern:

- die *EJBContext* Methoden *isCallerInRole* und *getCallerPrincipal*, die Enterprise Beans vom EJB-Container zur Verfügung gestellt werden, sowie

- den *HttpServletRequest* Methoden *isUserInRole* und *getUserPrincipal*, die Servlets und JSP-Seiten durch den Web-Container zur Verfügung gestellt werden.

Des weiteren fordert die Servlet-Spezifikation im *HttpServletRequest* Interface die Methode *getRemoteUser*, sie liefert den Benutzernamen, den der Benutzer im Client zur Authentisierung verwendete.

Diese Methoden erlauben es Komponenten, Entscheidungen auf der Basis der Sicherheitsrolle des Aufrufers oder entfernten Benutzers (Remote Users) in der Businesslogik zu treffen. Zum Beispiel erlauben sie einer Komponente, den Namen des Prinzipals des Aufrufers oder entfernten Benutzers als Datenbankschlüssel zu verwenden. Die Repräsentation des Namens des Prinzipals kann mit den unterschiedlichen J2EE-Produkten jedoch sehr variieren. Hier zeigt sich auch die Problematik der Programmatic Security, der Vorteil einer exemplarbasierten Zugriffskontrolle wird mit einer größeren Produktabhängigkeit erkauft.

Wenn eine Enterprise Bean die Methode *isCallerInRole* aufruft, dann überprüft der EJB-Container, ob der Aufrufer (wie im Security Context repräsentiert) der spezifizierten Rolle entspricht. Wenn ein Enterprise Bean die Methode *getCallerPrincipal* aufruft, so gibt der EJB-Container den Prinzipal zurück, der mit dem Security Context assoziiert ist. Die EJB-Spezifikation fordert, dass die Methode *getCallerPrincipal* niemals den Wert *null* liefert. Mit dem Namen des Prinzipals kann die Enterprise Bean überprüfen, ob sich ein Aufrufer in einer bestimmten Rolle befindet.

Die Funktionsweise der Methoden im *HttpServletRequest* Interface ist ähnlich. Allerdings dürfen hier alle Methoden auch den Wert *null* für einen nicht authentisierten Benutzer liefern.

Der Application Component Provider einer Komponente, die eine dieser Funktionen nutzt, muss alle in seinen Methodenaufrufen genutzten *role-name* Werte (Sicherheitsrollennamen) im Deployment Descriptor mit *security-role-ref* Elementen deklarieren.

[1] Der in der Spezifikation verwendete Begriff „Programmatic Security“ wird in dieser Arbeit übernommen, da die direkte Übersetzung in „programmatische Sicherheit“ den Sachverhalt nicht richtig wiedergibt.

Durch die Deklaration der im Programmcode verwendeten Rollenbezeichnungen, wird eine Konfiguration durch den Deployment Descriptor möglich. In ihm sollte das *security-role-ref* Element ein *role-name* Unterelement enthalten, das den Rollennamen enthält, der im Programmcode verwendet wurde. Dieses wird auf ein weiteres *security-role-ref* Unterelement mit der Bezeichnung *role-link* abgebildet, das als Wert den Namen einer im Deployment Descriptor definierten Sicherheitsrolle (*security-role*) enthält. So muss bei einer späteren Änderung der Sicherheitsrollen nur der Deployment Descriptor angepasst werden.

Der Container nutzt die Abbildung von *security-role-ref* auf *security-role*, wenn er das Ergebnis eines *isUserInRole* Methodenaufrufs bestimmt. Um zum Beispiel eine im Programmcode verwendete Sicherheitsrollenreferenz „MGR“ auf eine Sicherheitsrolle „manager“ abzubilden, wäre der Syntax wie folgt:

```
<security-role-ref>
        <role-name>MGR</role-name>
        <role-link>manager</manager>
</security-role-ref>
```

Die *isUserInRole* Methode referenziert eine Liste, um zu bestimmen, ob ein Aufrufer auf eine Sicherheitsrolle abgebildet ist. Wird in diesem Fall ein Servlet von einem Benutzer gerufen, welcher der Sicherheitsrolle *„manager“* angehört, und im Code des Servlets würde die Methode *isUserInRole("MGR")* gerufen, so wäre das Ergebnis *true*.

Lässt sich kein *security-role-ref* Element auf ein deklariertes *security-role* Element abbilden, muss der Container standardmäßig das *role-name* Element gegenüber einer Liste mit *security-role* Elementen der Web-Applikation prüfen.

4.4.3 Propagierung von Identitäten

Die J2EE-Spezifikation fordert von jedem J2EE-Produkt, dass es die *Propagierung von Identitäten* unterstützt. Es muss möglich sein, ein J2EE-Produkt so zu konfigurieren, dass in einer Kette von Aufrufen, für alle Autorisierungsentscheidung die propagierte Identität genutzt werden kann.

Bei einer entsprechenden Konfiguration des J2EE-Produkts, muss der Name des Prinzipals, der durch die *EJBContext* Methode *getCallerPrincipal* zurückgegeben wird, in einer Kette von Aufrufen von Enterprise JavaBeans in einer einzelnen Applikation, bei allen EJBs gleich sein. Wenn die erste EJB der Kette von einem Servlet oder einer JSP gerufen wurde, muss der Name des Prinzipals dem Rückgabewert der *HttpServletRequest* Methode *getUserPrincipal*, des aufrufenden Servlets oder JSP-Seite, entsprechen. Wenn die *HttpServletRequest* Methode *getUserPrincipal* jedoch *null* zurückgibt, ist der Prinzipal, der genutzt wird, um die EJB zu rufen, durch die J2EE-Spezifikation nicht definiert. Es muss daher möglich sein, die EJBs so zu konfigurieren, dass sie durch solche „anonymen“ Komponenten dennoch aufrufbar sind.

4.4.4 Run-As Identitäten

J2EE-Produkte müssen eine *Run-As* Funktionalität bieten, die es dem Application Component Provider und Deployer erlaubt, die Identität zu spezifizieren, unter der ein Enterprise Bean oder eine Webkomponente ausgeführt werden muss. Ruft eine mit *Run-As* konfigurierte Komponente eine andere Komponente, so propagiert sie ihre *Run-As* Identität und nicht die ihres Aufrufers.

4.4.5 Abbildung auf Rollen (Role Mapping)

Alle J2EE-Produkte müssen die Zugriffskontrollsemantiken, wie sie in den EJB-, JSP- und Servlet-Spezifikationen beschrieben sind, implementieren und ein Deployment Tool zum Abbilden der Sicherheitsrollen im Deployment Descriptor bereitstellen.

Die Durchsetzung der Zugriffsbeschränkungen auf Web-Ressourcen und Enterprise Beans, ob nun programmatic oder deklarativ, hängt davon ab, ob der Prinzipal, der mit der eingehenden Anfrage assoziiert ist, einer gegebenen Rolle angehört. Der Container trifft diese Entscheidung auf der Grundlage der Sicherheitsattribute des rufenden Prinzipals.

Beispiel für die Ermittlung der Sicherheitsrolle:

Ein Deployer hat in der operativen Umgebung eine Sicherheitsrolle auf eine Benutzergruppe abgebildet. In diesem Fall wird die Benutzergruppe des rufenden Prinzipals aus seinen Sicherheitsattributen gewonnen. Der Principal ist in einer gültigen Sicherheitsrolle, wenn die Benutzergruppe des Prinzipals zu der Benutzergruppe passt, auf welche die Sicherheitsrolle abgebildet wurde.

Die Quellen der Sicherheitsattribute können je nach Implementation des J2EE-Produkts variieren. Sicherheitsattribute können durch das Credential (den Identitätsnachweis) des rufenden Prinzipals oder durch den Security Context übertragen werden. In anderen Fällen werden die Sicherheitsattribute durch Dienste von vertrauenswürdigen Dritten, wie zum Beispiel Verzeichnisdienste, gewonnen.

4.4.6 Deklarative Zugriffskontrolle bei Web-Ressourcen

Nach der Servlet-Spezifikation sind *Security Constraints* der deklarative Weg, um die Schutzanforderungen der Webressourcen auszudrücken. Ein Security Constraint beinhaltet folgende Elemente:

- *Web Resource Collection*
- *Authorization Constraint*
- *User Data Constraint*

Eine *Web Resource Collection* ist eine Menge von URL-Mustern und HTTP-Methoden, die eine Menge von zu schützenden Ressourcen beschreibt. Alle Anfragen (Requests), die einen Pfad beinhalten, der mit einem dieser URL-Muster übereinstimmt, werden diesem Constraint zugeordnet. Der Container überprüft die Übereinstimmung mit dem URL-Muster nach dem gleichen Algorithmus, mit dem er auch Anfragen auf Servlets und statische Ressourcen überprüft. Dies wird in der Servlet-Spezifikation ausführlich beschrieben.

Ein *Authorization Contraint* ist die Menge der Sicherheitsrollen, denen der Zugriff auf eine Ressource gestattet ist. Werden keine Rollen definiert, darf kein Benutzer auf diesen Teil der Webapplikation zuzugreifen.

Ein *User Data Contraint* beschreibt die Anforderungen an die Transportschicht. Diese Anforderungen können die Integrität oder Vertraulichkeit der Verbindung betreffen. Der Container muss mindestens SSL verwenden, um Anfragen zu beantworten, die Ressourcen betreffen, welche mit *integral* oder *confidential* gekennzeichnet wurden. Wurde die ursprüngliche Anfrage über das HTTP-Protokoll gesendet, muss der Container den Client auf einen HTTPS Port umleiten.

Die Servlet-Spezifikation ([SERVLETS23] Seite 80) weist darauf hin, dass dieses Sicherheitsmodell nicht anwendbar ist, wenn ein Servlet den *RequestDispatcher* benutzt, um eine statische Ressource oder ein Servlet, mit Hilfe von *forward* oder *include*, aufzurufen!

Beispiel eines Deployment Descriptors (aus der Servlet-Spezifikation):

```
<!DOCTYPE web-app PUBLIC "-//SUN Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.SUN.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
<display-name>A Secure Application</display-name>
<security-role>
        <role-name>manager</role-name>
</security-role>
<servlet>
    <servlet-name>catalog</servlet-name>
    <servlet-class>com.mycorp.CatalogServlet</servlet-class>
    <init-param>
        <param-name>catalog</param-name>
        <param-value>Spring</param-value>
    </init-param>
<security-role-ref>
    <role-name>MGR</role-name>
    <!-- role name used in code -->
    <role-link>manager</role-link>
</security-role-ref>
</servlet>
<servlet-mapping>
    <servlet-name>catalog</servlet-name>
    <url-pattern>/catalog/*</url-pattern>
</servlet-mapping>
<security-constraint>
    <web-resource-collection>
        <web-resource-name>SalesInfo</web-resource-name>
        <url-pattern>/salesinfo/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>manager</role-name>
    </auth-constraint>
    <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
</security-constraint>
</web-app>
```

4.4.7 Deklarative Zugriffskontrolle bei Enterprise JavaBeans

Der Zugriff auf Methoden von Enterprise JavaBeans kann im Deployment Descriptor durch *Method-Permission* Elemente kontrolliert werden. Sie enthalten eine Liste von Methoden, auf die eine Sicherheitsrolle zugreifen darf.

Method Permissions werden im Deployment Descriptor als eine binäre Relation von einer Menge mit Sicherheitsrollen auf eine Menge mit Methoden der Home- und Component-Interfaces der EJBs, einschließlich ihrer Superinterfaces (d.h. einschließlich aller Methoden der *EJBHome*- und *EJBObject-Interfaces* und/oder der *EJBLocalHome*- und *EJBLocalObject-Interfaces*) abgebildet.

Ist der rufende Prinzipal in einer der Sicherheitsrollen, denen der Zugriff auf die Methode gestattet ist, so darf der Prinzipal die Methode ausführen. Umgekehrt gilt, ist der Prinzipal in keiner der Sicherheitsrollen, so darf der die Methode nicht ausführen.

Die präzise Definition der Aufgaben und Verantwortlichkeiten ist bei deklarativer Sicherheit sehr wichtig. Diese Definitionen finden sich in der EJB-Spezifikation:

Der Bean Provider und Application Assembler kennen im allgemeinen nicht die Sicherheitsumgebung der operativen Umgebung, daher sind die Sicherheitsrollen als *logische* Privilegien zu betrachten. Jede Sicherheitsrolle repräsentiert eine Klasse von Benutzern, in der alle Benutzer die gleichen Zugriffsrechte auf die Applikation besitzen.

Bei der Verwendung von Programmatic Security muss der Bean Provider für jeden im Programmcode der Enterprise Bean verwendeten Namen einer Sicherheitsrolle eine Sicherheitsrollenreferenz mit dem *security-role-ref* Element deklarieren. Wenn der Bean Provider eine Sicherheitsrollenreferenz deklariert hat, muss der Application Assembler jede einzelne Referenz auf die Sicherheitsrollen abbilden, die in *security-role* Elementen definiert sind.

Das folgende Beispiel eines Deployment Descriptors zeigt die Abbildung einer Security Role namens *payroll* auf eine Security Role namens *payroll-department.*

```
...
<security-role-ref>
        <description>
                BESCHREIBUNGSTEXT
        </description>
        <role-name>ImCodeVerwendeteRollenBezeichnung</role-name>
        <role-link>DefinierteRolleImDeploymentDescriptor</role-link>
</security-role-ref>
...
```

Der Bean Provider und Application Assembler können optional einen *Security View* der EJBs einer ejb-jar Datei definieren. Der Security View besteht aus einer Menge von Sicherheitsrollen und vereinfacht allerdings die Arbeit des Deployers.

Der Application Assembler definiert die Method-Permission-Relation im Deployment Descriptor unter Benutzung der *method-permission* Elemente wie folgt:

- Die *Method-Permission-Relation* bildet ein Paar (R,M), dann und nur dann, wenn es der Security Role R erlaubt ist, auf eine Methode M zuzugreifen.

- Eine Method-Permission-Relation ist die Vereinigung aller Method-Permissions, die in einen individuellen *method-permission* Element definiert wurden.

- Eine Sicherheitsrolle oder Methode kann in mehreren *method-permission* Elementen erscheinen.

- Jedes *method-permission* Element umfasst eine Liste mit einer oder mehreren Sicherheitsrollen und eine Liste mit einer oder mehreren Methoden. Allen gelisteten Sicherheitsrollen ist es erlaubt, auf die gelisteten Methoden zuzugreifen. Jede Sicherheitsrolle der Liste wird durch ein *role-name* Element identifiziert und jede Methode (oder eine Menge von Methoden, siehe unten) wird durch ein *method* Element identifiziert. Eine Beschreibung kann im optionalen *description* Element angegeben werden.

Der Application Assembler kann vermerken, dass die Autorisierung bei einigen Methoden nicht vor dem Aufruf durch den Container geprüft wird. Dazu nutzt der Application Assembler anstelle des *role-name* Elements das *unchecked* Element, welches anzeigt, dass die Methode bezüglich der Autorisierung nicht geprüft wird. Wenn die Method-Permission-Relation beides, das *unchecked* Element und eine oder mehrere Sicherheitsrollen, für eine Methode spezifiziert, so sollte die Autorisierung nicht geprüft werden.

Der Application Assembler kann das *exclude-list* Element dazu nutzen, eine Menge von Methoden zu definieren, die auf keinen Fall gerufen werden dürfen. Der Deployer muss die Sicherheit der Enterprise Bean so konfigurieren, dass der Zugriff auf die Methoden in der *exclude-list* nie erlaubt wird. Wenn für eine gegebene Methode beides, *exclude-list* Element und Method-Permission-Relation, spezifiziert wurde, so muss der Deployer das Enterprise Bean so konfigurieren, dass kein Zugriff auf diese Methode erlaubt ist.

Es ist möglich, dass einige Methoden keine Security Roles zugeordnet wurden und sie auch nicht im *exlude-list* Element enthalten sind. In diesem Fall liegt es in der Verantwortlichkeit des Deployers, jeder unspezifizierten Methode Method-Permissions zuzuweisen, indem er sie Sicherheitsrollen zuordnet oder sie als *unchecked* markiert.

Das *method* Element nutzt *ejb-name*, *method-name* und *method-params* Elemente, um eine oder mehrere Methoden des Home- und Component-Interfaces des Enterprise Beans zu bezeichnen. Es gibt drei gültige Arten, das *method* Element zu verwenden:

Typ 1:

```
<method>
        <ejb-name> EJBNAME</ejb-name>
        <method-name>*</method-name>
</method>
```

Dieser Typ wird verwendet, um alle Methoden der Home- und Component-Interfaces einer spezifizierten Enterprise Bean zu referenzieren.

Typ 2:
```
<method>
    <ejb-name> EJBNAME</ejb-name>
    <method-name> METHOD</method-name>
</method>
```

Dieser Typ wird verwendet, um eine spezielle Methode der Home- oder Component-Interfaces der spezifizierten Enterprise Bean zu referenzieren. Gibt es mehrere Methoden mit dem selben überladenen Name, dann referenziert dieser Typ alle überladenen Methoden.

Typ 3:
```
<method>
    <ejb-name> EJBNAME</ejb-name>
    <method-name> METHOD</method-name>
        <method-params>
        <method-param> PARAMETER_1</method-param>
        ...
        <method-param> PARAMETER_N</method-param>
    </method-params>
</method>
```

Dieser Typ wird verwendet, um eine Methode innerhalb einer Menge von Methoden mit überladenen Namen zu referenzieren.

Das optionale *method-intf* Element kann für die Differenzierung zwischen den Home- und Component Interfaces genutzt werden, wenn Methoden des gleichen Namens und der gleichen Signatur mehrfach in den Home- und Component Interfaces einer Enterprise Bean definiert wurden. Zum Beispiel: *<method-intf>Remote</method-intf>*

Das folgende Beispiel zeigt, wie die Sicherheitsrollen in Method-Permissions zugewiesen werden:

```
...
<method-permission>
    <role-name>ROLLENNAME</role-name>
    <method>
        <ejb-name>EJBNAME</ejb-name>
        <method-name>METHODENNAME_A</method-name>
    </method>
    <method>
        <ejb-name>EJBNAME</ejb-name>
        <method-name>METHODENNAME_B</method-name>
    </method>
    <method>
        <ejb-name>EJBNAME</ejb-name>
        <method-name>METHODENNAME_C</method-name>
    </method>
</method-permission>
...
```

Der Application Assembler spezifiziert üblicherweise, ob die Identität des Aufrufers oder eine spezifische *Run-As* Identität für die Ausführung der Methoden der Enterprise Bean genutzt wird. Dieses Verhalten wird im *security-identity* Element, mit den Elementen *use-caller-identidy* und *run-as*, definiert. Das *security-identity* Element kann nicht für Message-Driven Beans verwendet werden und seine Angabe ist für den Application Assembler optional.

Das folgende Beispiel zeigt die Definition einer *run-as* Identität:

```
...
<enterprise-beans>
...
<session>
        <ejb-name>EJBNAME</ejb-name>
...
        <security-identity>
                <run-as>
                        <role-name>admin</role-name>
                </run-as>
        </security-identity>
...
</session>
...
</enterprise-beans>
...
```

Letztlich ist der Deployer verantwortlich für:

- die Sicherheit der zusammengestellten und deployten Applikation.
- die Zuweisung der J2EE-Applikation zu einer Security Policy Domain.
- die Abbildung der Prinzipale der operativen Umgebung auf die Sicherheitsrollen.
- die Überprüfung der Delegation und Run-As Funktionalitäten.
- die Absicherung der Ressource Manager.
- die Überprüfung und Vervollständigung der Deployment Descriptoren.

4.4.8 Autorisierung von Programmcode

J2SE-Klassen werden nach dem klassischen Permission Konzept autorisiert, J2EE-Komponenten basieren auf dem Konzept von Sicherheitsrollen.

Ein J2EE-Produkt kann die Nutzung von gewissen J2SE Klassen und Methoden beschränken, um das System abzusichern und ein ordnungsgemäßes Verhalten des Systems sicherzustellen.

Die minimale Menge an Rechten, die für ein J2EE-Produkt erforderlich sind, um diese an eine J2EE-Applikation zu vergeben, wird in der J2EE-Spezifikation im Kapitel „Java 2 Platform, Standard Edition (J2SE) Requirements“ beschrieben.

Alle J2EE-Produkte müssen fähig sein, Applikationskomponenten mit genau diesen Rechten zu deployen. Ein J2EE Product Provider kann wählen, ob er den selektiven Zugriff auf Ressourcen, mit Hilfe des J2SE-Sicherheitsmodells, ermöglicht. Der Mechanismus, der dazu genutzt wird, ist abhängig vom J2EE-Produkt. Eine zukünftige Version der Deployment Descriptor Definition (siehe Kapitel „Application Assembly and Deployment“ in der J2EE-Spezifikation) könnte es erlauben, weitere Rechte zu definieren, die eine Komponente für den Zugriff braucht.

4.5 Zukünftige Anforderungen

In Bezug auf die Sicherheit von J2EE-Produkten nennt die J2EE-Spezifikation 1.3 folgende zukünftige Anforderungen:

Auditing

Die J2EE-Spezifikation v1.3 spezifiziert noch keine Anforderungen an das Auditing von sicherheitsrelevanten Ereignissen, noch spezifiziert sie APIs, mit denen Applikationskomponenten Auditrecords generieren könnten. In der J2EE-Spezifikation 1.4 ist dieses Feature noch nicht zu finden. In zukünftigen Versionen soll dies aber geschehen. Es stellt sich die Frage, warum SUN für die Spezifikation eines so wichtigen Features so lange Zeit benötigt.

Im Rahmen des *Java Community Process* wurde unter JCP-85 „Rules-based Authorization and Audit” [JCP85] eine entsprechende Funktionalität beantragt. Das *Execuritve Commitee for SS/EE* hat den Antrag abgelehnt. Die Begründung von IBM ist höchst interessant:

“On 2000-10-09 IBM voted No with the following comment:
This JSR needs to be split into 2 as there are two large topics included, each of which needs different kinds of expertise. We believe that there should be one JSR for rules based authorization and one for audit. We strongly recommend that the JSR be re-written and re-submitted as 2 JSRs. In addition our other concerns are:

The JSR points out that EJB and JAAS have conflicting security models. We disagree, as JAAS can be used to implement a J2EE security model with additional classes. We are concerned where this comment may be leading and would like to understand more.
While we agree that there may be a need for rules-based policies, we are very concerned about the added complexity and believe this needs serious debate and discussion. We need to carefully think through the value of adding a lot of complexity to the security model, if the flexibility is worth the added complexity or will it just cause programmers to take another approach to avoid the complexity.”

Management

In Version 1.4 der J2EE-Spezifikation, die mittlerweile als public final draft 2 vorliegt, sind die Java Management Extensions (JMX) integriert. Die JMX Spezifikation 1.1 [JMXS11] definiert eine Architektur, Design Patterns, APIs und Dienste für das Management und Monitoring von Applikationen und Netzwerken in der Programmiersprache Java.

Exemplarbasierte Zugriffskontrolle (Instance-based Access Control)

Einige Applikationen benötigen eine exemplarbasierte Zugriffskontrolle, die auf den Inhalten der Daten der Exemplare basiert. Die verwendete deklarative Sicherheit ist klassenbasiert. SUN hofft dieses Feature, in der Zukunft mit einbringen zu können. In der Programmatic Security ist dies bereits der Fall. Ziel ist es jedoch, die Sicherheit außerhalb der Applikation zu lokalisieren. Programmatic Security bietet die exemplarbasierte Zugriffskontrolle nur innerhalb von Applikationen. In der J2EE-Spezifikation 1.4 ist dieses Feature noch nicht zu finden. In zukünftigen Versionen soll dies aber geschehen.

User Registration

Webbasierte Internetapplikationen müssen oft viele Kunden dynamisch verwalten, wobei sie Benutzern erlauben, sich als neue Kunden zu registrieren. Dieses Szenario wurde ausführlich in der Servlet Expert Group (JSR-53) diskutiert, aber es war unmöglich, einen Konsens über die angemessene Lösung herbeizuführen. So wurde dieser Punkt nicht in die J2EE-Spezifikation v1.3 aufgenommen. Auch in der J2EE-Spezifikation 1.4 ist dieses Feature noch nicht zu finden. In zukünftigen Versionen soll dies aber geschehen.

4.6 Ein einfaches Beispiel aus der J2EE-Spezifikation

Die J2EE-Spezifikation enthält ein einfaches Beispiel einer Webapplikation. Dieses wird, zum besseren Verständnis der J2EE-Security, in diesem Kapitel kurz wiedergegeben.

Gegeben sei eine einfache Applikation mit:

- einem Webclient.
- einer JSP, die als Benutzerschnittstelle dient.
- einer Enterprise JavaBean, welche die Businesslogik enthält.

In diesem Beispiel vertraut der Webclient dem Webserver, welcher als Authentisierungsproxy handelt. Der Webserver etabliert eine Session mit den Authentisierungsdaten des Clients.

Schritt 1: Initial Request

Der Webclient erfragt eine geschützte Ressource unter der Haupt-URL der Applikation beim Webserver.

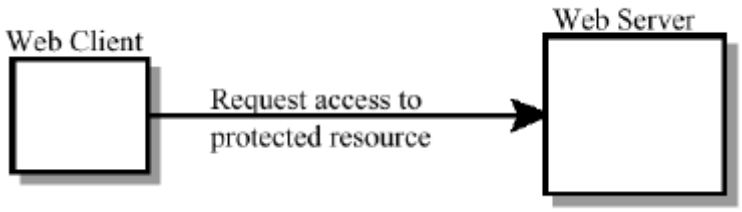

Abbildung 37: Initial Request aus [J2EES13]

Der Webclient hat sich noch nicht gegenüber der Applikationsumgebung authentisiert. Der Webserver erkennt dies und initiiert einen, zur Ressource passenden, Authentisierungsmechanismus.

Schritt 2: Initial Authentication

Der Webserver antwortet mit einem Formular, das der Webclient nutzt, um die Authentisierungsdaten des Benutzers zu sammeln (z.B. Benutzername und Passwort).

Der Webclient leitet die Authentisierungsdaten weiter zum Webserver, wo sie überprüft werden (siehe Abbildung 38).

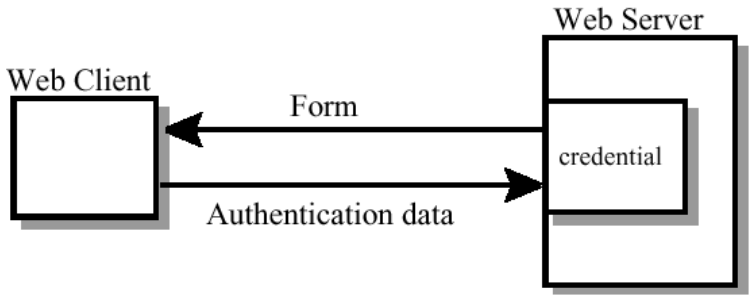

Abbildung 38: Initial Authentication aus [J2EES13]

Der Validierungsmechanismus kann lokal im Webserver liegen oder es werden darunter liegende Sicherheitsdienste verwendet. Auf der Basis der Validierung erzeugt der Webserver einen Identitätsnachweis (Credential) für den Benutzer.

Schritt 3: URL Autorisierung

Der Web-Container konsultiert die Security Policy, die mit der Webressource assoziiert ist, um die Sicherheitsrollen zu bestimmen, denen der Zugriff auf die Ressource gestattet ist. Dann überprüft er den Identitätsnachweis des Benutzers auf seine Rollenzugehörigkeit.

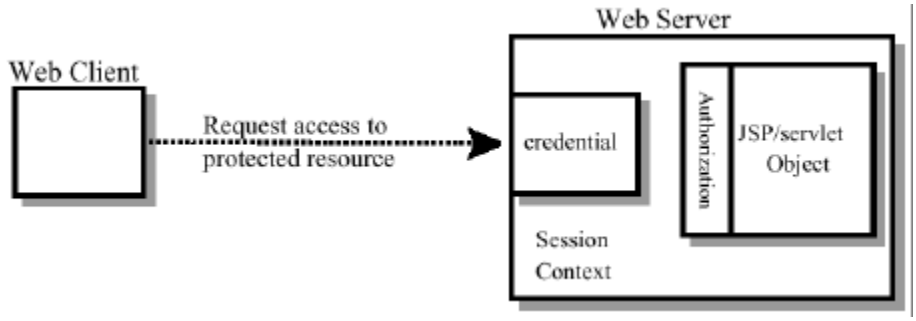

Abbildung 39: URL Authorization aus [J2EES13]

Wenn der Benutzer einer gültigen Rolle angehört, endet die Auswertung des Webservers mit dem Ergebnis „is authorized“. Wenn nicht, dann ist das Ergebnis „is not authorized“.

Schritt 4: Ausführung der ursprünglichen Anfrage

Wenn der Benutzer autorisiert ist, liefert der Webserver das Ergebnis des ursprünglichen URL-Requests (wie in Abbildung 40 gezeigt) zurück. In diesem Beispiel wird als Antwort die URL einer JSP-Seite zurückgegeben, die es dem Benutzer ermöglicht, Formulardaten an eine Enterprise Bean zu senden.

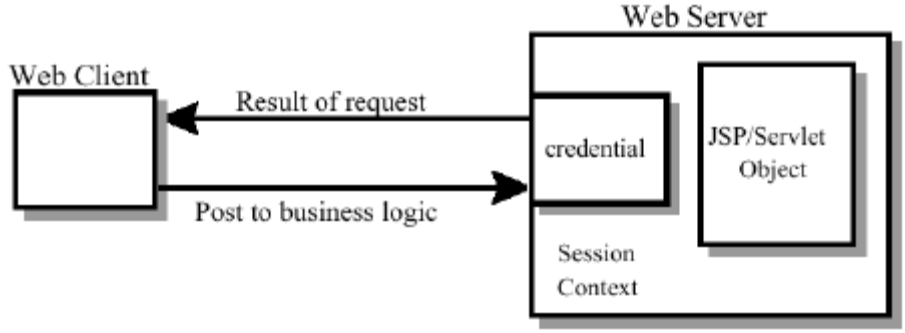

Abbildung 40: Ausführung der ursprünglichen Anfrage aus [J2EES13]

Schritt 5: Aufrufen der Enterprise JavaBean Businessmethoden

Die JSP-Seite ruft eine Methode der Enterprise Bean, wobei sie den Identitätsnachweis des Benutzers nutzt, um eine sichere Assoziation (*secure association*) zwischen der JSP-Seite und der Enterprise Bean zu etablieren (siehe Abbildung 41).

Die Assoziation wird durch zwei in Beziehung stehende Security Contexte implementiert: einer im Webserver (Web-Container) und einer im EJB-Container.

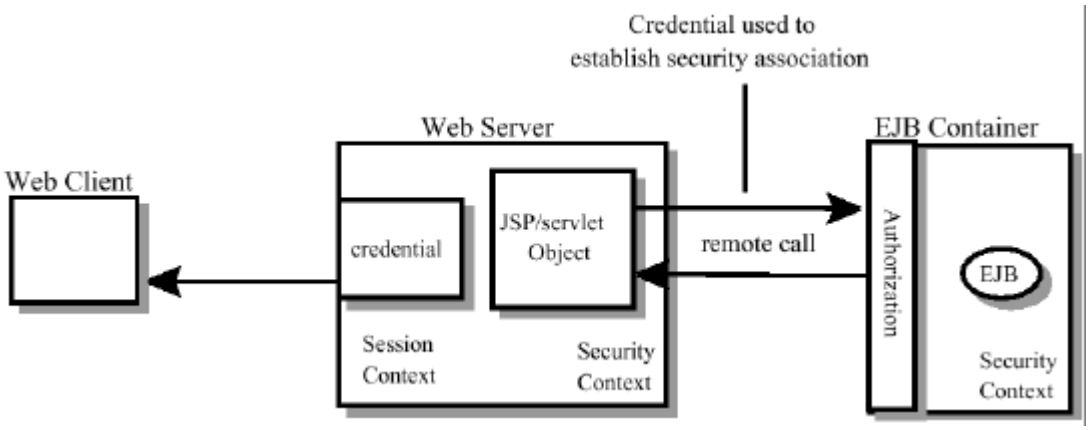

Abbildung 41: Aufrufen der Enterprise Bean Businessmethoden aus [J2EES13]

Der EJB-Container ist verantwortlich für die Durchsetzung der Zugriffskontrolle auf die Enterprise Bean Methoden. Er konsultiert die Security Policy, die mit der Enterprise Bean assoziiert ist, um die Sicherheitsrollen zu bestimmen, denen der Zugriff auf die Methoden gestattet ist.

Nun nutzt der EJB-Container den Security Context, der mit Aufrufer assoziiert ist, und überprüft die Rollenzugehörigkeit des Aufrufers. Die Auswertung des Containers endet mit dem Ergebnis “is authorized”, wenn der Benutzer einer gültigen Sicherheitsrolle angehört. Anderenfalls lautet das Ergebnis “not authorized” und der Container erzeugt eine Ausnahme (Exception), die zur rufenden JSP-Seite weitergeleitet wird.

Ist der Aufruf autorisiert, so übergibt der Container die Kontrolle an die Enterprise Bean Methode. Das Ergebnis der Ausführung der Bean wird an die JSP und letztlich an den Benutzer durch den Webserver und Webclient weitergegeben.

5 Der WebSphere Application Server

In diesem Kapitel wird die Architektur und Funktionsweise des IBM Websphere Application Server Advanced Edition v4.0 konzeptionell dargestellt. Ziel ist es, eine konkrete Implementation der J2EE-Technologie darzustellen.

WebSphere v4.0 basiert auf der J2EE1.2-Spezifikation. Sein Nachfolger WebSphere v5.0 basiert auf der J2EE1.3-Spezifikation, die in dieser Arbeit beschrieben wird. Er ist bisher nur als „Developer Version“ erhältlich und es mangelt an hinreichender Literatur. Daher wird die Version 4 dargestellt und wenn möglich Bezug auf Version 5 genommen. Dies bedeutet auch, dass nicht alle Anforderungen der J2EE1.3-Spezifikation erfüllt sein müssen. Ich habe mich in dieser Arbeit auf die J2EE1.3-Spezifikation konzentriert, da sie die aktuelle Erweiterung der J2EE 1.2 Spezifikation darstellt und mehrere Korrekturen enthält.

Die Untersuchung der Architektur, Funktionsweise und der Sicherheitsmechanismen findet rein konzeptionell statt, da IBM die Spezifikationen und Implementationsdetails von WebSphere der Öffentlichkeit nicht zur Verfügung stellt.

Als Literatur diente primär:

- IBM Redbook “IBM WebSphere V4.0 Advanced Edition Handbook”
- IBM Redbook “IBM WebSphere V4.0 Advanced Edition Security”
- Die Online-Dokumentation zu WebSphere v4.0
- Email Korrespondenz mit IBM

WebSphere 4.0 ist in drei unterschiedlichen Editionen erhältlich, wobei alle einen gemeinsamen „J2EE-kompatiblen“ Kern besitzen. Eine Übersicht über die Unterschiede der verschiedenen Editionen bietet Abbildung 42.

- **Advanced Edition Version 4.0, Single-Server (AEs)**
  Diese Edition ist für die Entwicklung und zum Testen von J2EE-Applikationen gedacht. Sie ist eine leichtgewichtige Version, in der sich der Administrationsserver und der Applikationsserver eine JVM teilen. Die Administration findet über ein einfaches Web-Interface statt und die Administrationsdaten werden in einer XML-Datei verwaltet.

- **Advanced Edition Version, Development Only (AEd**)
  Diese Edition entspricht der Single-Server Version mit einem anderen Lizenzmodell. Sie ist für Windows NT/2000 frei erhältlich.

- **Advanced Edition Version 4.0 (AE)**
  Diese Edition ist die Vollversion. Sie ist für Produktionsumgebungen und hochskalierbare Umgebungen gedacht. Sie unterstützt mehrere verteilte Server und verteilte Sicherheit. Das Clustering und Cloning von Servern ist ebenfalls möglich. Der Administrationsserver und der Applikationsserver haben je ihre eigene JVM. Die Administration ist vollständig IIOP-basiert und als Client dient die *Administrative Console*. Die Administrationsdaten werden in einem Repository in einer Datenbank verwaltet. Zusätzlich bietet sie einige Quality-of-Service Erweiterungen.

Feature	AEs	AEd	AE
Full J2EE compliance	Yes	Yes	Yes
Web services support	Yes	Yes	Yes
Connection management and pooling	Yes	Yes	Yes
XML parsing	Yes	Yes	Yes
Expanded DB support	Yes	Yes	Yes
Built-in Web server	Yes	Yes	Yes
Web-based admin console	Yes	Yes	No
Firewall support	Yes	Yes	Yes
Production certified	Yes	No	Yes
Application server	One	One	Multiple
Machine configuration	Single	Single	Multiple
Security	Local OS	Local OS	OS, LDAP, Custom
Integration with Enterprise Edition	No	No	Yes
Directory Services	No	No	Yes
Application-level WLM	No	No	Yes
Clustering and cloning	No	No	Yes
Additional caching	No	No	Yes
Distributed security	No	No	Yes
IIOP-based admin console	No	No	Yes
Multi-node administration	No	No	Yes
IBM DB2 included	No	No	Yes
Merant JDBC drivers included	No	No	Yes
J2C supported	No	No	Yes

Abbildung 42: WebSphereVersionen aus [SG246176]

5.1 Die WebSphere Architektur

In diesem Kapitel wird die Architektur des WebSphere v4.0 dargestellt. Abbildung 43 gibt einen Überblick über alle wichtigen Komponenten.

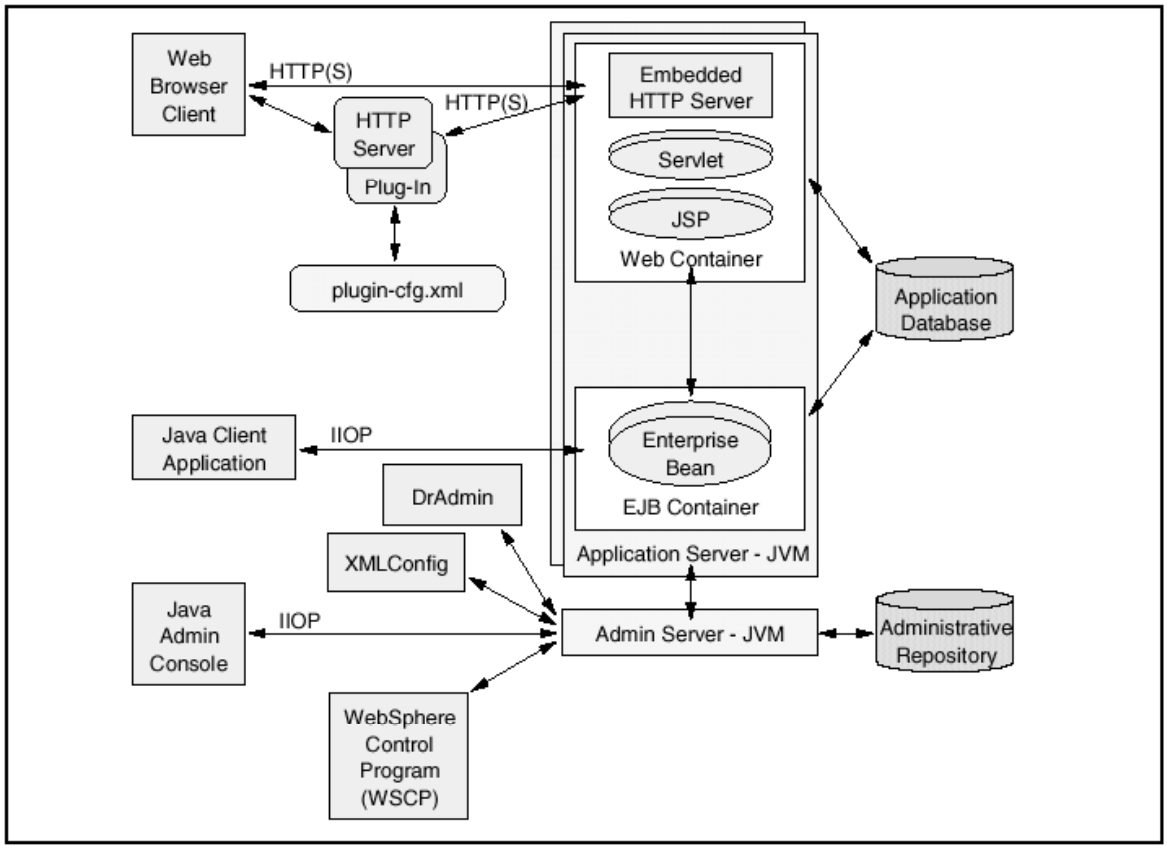

Abbildung 43: Hauptkomponenten des WAS40AE aus [SG246176]

5.1.1 Application Server

Jeder *Application Server* enthält eine JVM und implementierte alle Dienste. In seiner JVM läuft je ein Web- und ein EJB-Container. Die Container stellen selbst keine Dienste zur Verfügung, diese sind nur im Server implementiert. Sie kapseln die, in ihnen beherbergten, Komponenten und bilden so eine Schnittstelle. Fordert eine Komponente einen Dienst an, vermittelt der Container den Dienst des Servers an die Komponente. Dies ist nach der J2EE-Spezifikation zulässig.

Anmerkung: Die Angaben von IBM sind bezüglich der Dienste der Container widersprüchlich. Mal wird behauptet, dass die Container gewisse Dienste anbieten und mal nicht. Teilweise wird auch von logischen Containern gesprochen. Ich nehme an, dass es sich „nur“ um logische Container handelt, die softwaretechnisch eine Schnittstelle implementieren, welche die Komponenten kapselt. Möglicherweise ein intelligenter Wrapper. Dafür spricht:

- Nur eine JVM pro Application Server, Web- und EJB-Container.
- Die Anzahl der Container ist auf zwei begrenzt (je ein Web- und ein EJB-Container).
- Die Container lassen sich weder entfernen, noch lassen sich weitere hinzufügen.

- Die Historie von WebSphere.
- Widersprüchliche Aussagen im Redbook [SG246176] und der restlichen Dokumentation.

Aufgrund des Umfanges einer näheren Betrachtung kann ich dies in dieser Arbeit leider nicht untersuchen. Daher werde ich im folgenden von einer „indirekten“ Diensterbringung sprechen.

Der Web-Container beherbergt Servlets, JSPs und statische Web-Ressourcen. Der EJB-Container beherbergt EJBs. Dies bedeutet, dass im Application Server alle J2EE-Komponenten in einer einzigen JVM ablaufen und von den gleichen Class Loadern geladen werden. Die Auswirkungen einer Applikation auf die Sicherheit einer anderen Applikation wird im Kapitel „WebSphere Class Loader“ beschrieben.

Der Web-Container erhält Client-Anfragen über einen Webserver, mit dem er durch ein Plug-In verbunden ist. Der Web-Container bzw. Application Server enthält zwar einen eingebauten Webserver, dieser ist allerdings nur für Testzwecke gedacht und bietet nicht die Performanz und Sicherheit eines echten Webservers. Der EJB-Container kann als einziger Container direkt mit Application Clients kommunizieren, daher sollte er besonders gesichert werden.

Administriert wird der Application Server über den *Admin Server* (siehe Kapitel „WebSphere Administrationsmodell“), der über eine eigene JVM verfügt. An den Application Server ist in der Regel eine Datenbank angeschlossen, die von einigen Applikationskomponenten benötigt wird. Ein voreingestellter Application Server, *Default Server* genannt, wird automatisch während der normalen Installation von WebSphere konfiguriert. Der Default Server besitzt, wie jeder andere Application Server, einen Web-Container und einen EJB-Container.

5.1.2 HTTP-Server und Plug-in

Der Webserver bzw. *HTTP-Server* ist über ein *Plug-In* mit dem Application Server verbunden. Die Kommunikation findet über HTTP oder HTTPS statt. Das Plug-In wird durch eine XML-Konfigurationsdatei administriert, in der das Kommunikationsprotokoll und die Ressourcen definiert werden, die der Application Server bedient. Anhand dieser Daten kann das Plug-In Anfragen an den Webserver oder Applikationsserver weiterleiten. Das Plug-In ist für folgende Webserver verfügbar: Apache, Microsoft IIS und Netscape iPlanet.

5.1.3 Embedded HTTP-Server

Der *Embedded HTTP-Server* ist ein, in den Application Server integrierter, Webserver. Er ist nur für Test- und Entwicklungszwecke gedacht und sollte aus Performanz- und Sicherheitsgründen auf keinen Fall in der Produktionsumgebung verwendet werden.

5.1.4 Virtual Hosts

Ein *Virtual Host* ist eine Konfiguration, die eine einzelne Hostmaschine wie viele verschiedene Hostmaschinen erscheinen lässt. Dies erlaubt einer einzelnen physischen Maschine, mehrere, unabhängig von einander konfigurierte und administrierte, Applikationen zu unterstützen. Jeder Virtual Host hat einen logischen Namen und eine Liste mit einem oder mehreren DNS-Aliases, unter denen er bekannt ist. Ein DNS-Alias enthält einen TCP/IP-Hostname und eine Port-Nummer.

Die voreingestellten Ports und Aliases sind:

*:80 : für die Verwendung eines externen HTTP-Ports, der nicht gesichert ist.
*:9080 : für die Verwendung eines eingebunden HTTP-Ports, der nicht gesichert ist.
*:443 : für die Verwendung eines gesicherten externen HTTPS-Ports.
*:9443 : für die Verwendung eines gesicherten eingebundenen HTTPS-Ports.

Zum Routing der Anfragen vergleicht der Server den Hostname und Port der URL der Anfrage mit den bekannten Aliases der Liste. Wird keine Übereinstimmung gefunden, so antwortet der Server mit einem Error 404.

5.1.5 Server Groups

Ein Application Server lässt sich auf eine *Server Group* abbilden. Diese ist dann die logische Repräsentation des Application Servers und seines Inhaltes. Sie ist weder mit einem bestimmten physischen Node assoziiert, noch entspricht sie einem realen Server, der auf irgendeinem Node läuft. Mit ihr lassen sich Kopien (*Clones*) des Application Servers und seines Inhaltes erstellen. Die Server Group gestattet die Betrachtung und Modifizierung jeder Eigenschaft, die mit den logischen Objekten assoziiert ist. Änderungen an der Server Group führen dazu, dass alle Klone geändert und neu gestartet werden.

5.1.6 Clones

Die Abbildung eines vollständig konfigurierten Application Servers auf eine Server Group nennt man *Cloning*. Die Kopien des Servers werden *Clones* (Klone) genannt. Diese Kopien sind vollständig identisch mit der Server Group, aus der sie gebildet wurden. Im Gegensatz zur Server Group repräsentieren die Klone Application Server Prozesse, die auf echten physischen Nodes laufen.

Eine einzelne Maschine kann mehrere Klone beherbergen, wobei man von vertikaler Skalierung spricht, oder sie sind auf mehrere Maschinen verteilt, was einer horizontalen Skalierung entspricht. Durch Skalierung können Klone zur Lastverteilung eingesetzt werden. Eine Anfrage nach einer Serverressource kann von mehreren Klonen bearbeitet werden (Abbildung 44).

Eine Modifizierung der Server Group wird automatisch zu allen Klonen propagiert. Die Modifikation wird wirksam, wenn die Klone neu gestartet werden. Wird ein Klon direkt modifiziert, so ist er nicht mehr identisch mit der Server Group. Er bleibt jedoch Teil seiner Server Group, solange bis er von ihr getrennt wird.

5.1.7 Web-Container

Jeder Application Server beherbergt einen einzigen logischen *Web-Container*, der modifiziert werden kann. Dieser lässt sich weder entfernen, noch lassen sich weitere hinzufügen. Ein Web-Container enthält einen Session Manager und beherbergt Servlets, JSPs und statische Web-Ressourcen, wie in der J2EE-Spezifikation beschrieben. Er erbringt indirekt alle von der J2EE-Spezifikation geforderten Dienste[1].

[1] Eine genauere Betrachtung findet sich im Kapitel: „Differentialanalyse“.

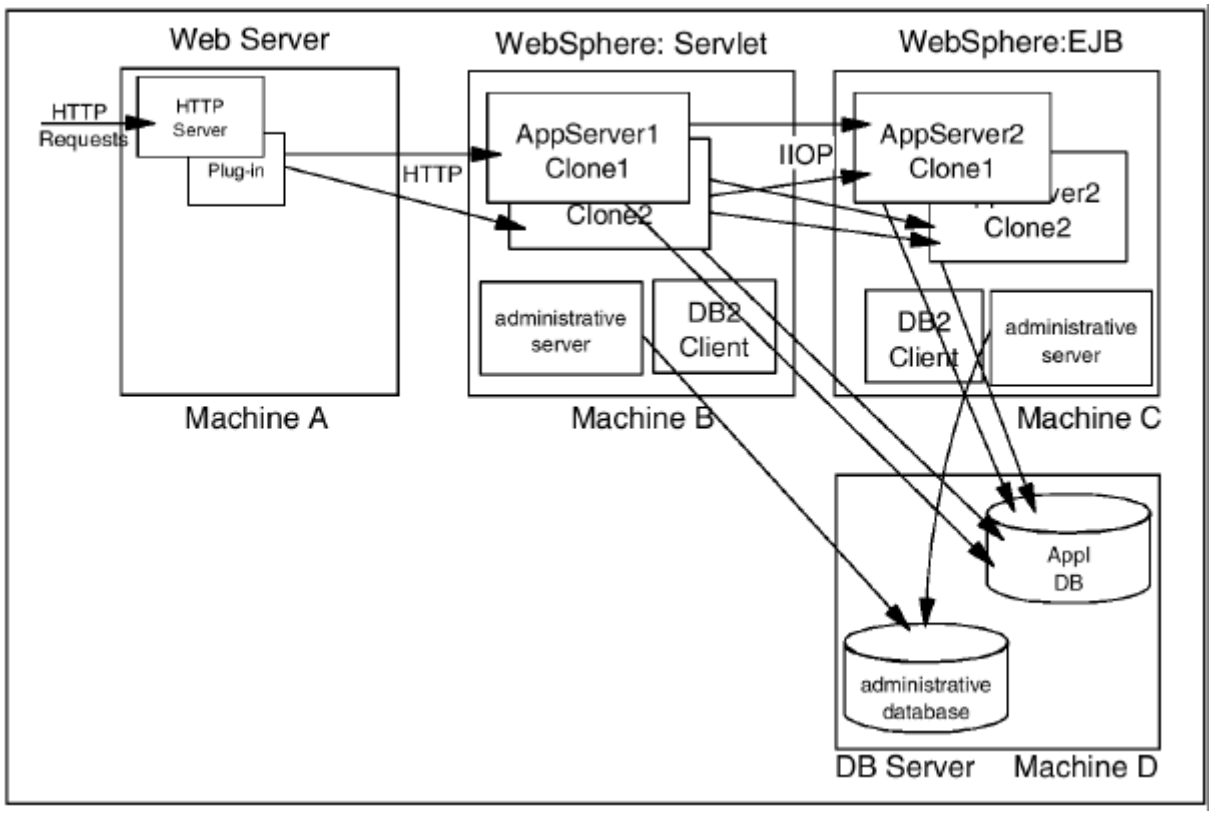

Abbildung 44: WebSphere Skalierung aus [SG246176]

.

5.1.8 Web-Module

Ein Web-Modul ist eine *Web Archive Datei* (.war), die Servlets, JSPs, Deployment Descriptoren und statischen Inhalte einer Webapplikation enthält. Eine WAR-Datei ist eine standardisierte Java Archive Datei, die einen zusätzlichen Deployment Descriptor (*web.xml*) enthält. Die Datei *web.xml* enthält Informationen über die Struktur und externen Abhängigkeiten der Webkomponenten des Moduls und beschreibt, wie diese Komponenten zur Laufzeit genutzt werden. Das Web-Modul kann als alleinstehende Applikation genutzt werden oder es wird mit anderen Modulen (Web-Module, EJB-Module oder beides) zu einer J2EE-Applikation kombiniert. Web-Module werden im Web-Container deployed und laufen in diesem.

5.1.9 EJB-Container

Jeder Application Server beherbergt einen einzigen logischen *EJB-Container*, der modifiziert werden kann. Er kann nicht entfernt werden und es lassen sich keine weiteren hinzufügen. Der EJB-Container beherbergt Session- und Entity-Beans und bietet ihnen indirekt die benötigten Dienste[1], die von der J2EE-Spezifikation gefordert werden.

5.1.10 EJB-Module

Ein EJB-Modul ist eine Java Archive Datei (.jar), die eine oder mehrere EJBs mit ihren Deployment Descriptoren enthält. Diese lässt sich direkt in einem EJB-Container deployen. Das EJB-Modul kann als alleinstehende Applikation genutzt werden oder es wird mit anderen Modulen (Web-Module, EJB-Module oder beides) zu einer J2EE-Applikation kombiniert.

[1] Eine genauere Betrachtung findet sich im Kapitel: „Differenzialanalyse“.

5.1.11 EAR-Module

Ein EAR-Modul repräsentiert eine J2EE-Applikation. Sie ist eine Enterprise Archive Datei (.ear), die mehrere Web- und EJB-Module sowie Deployment Descriptoren enthält. Die Datei *application.xml* ist der Deployment Descriptor der Applikation. EAR-Module werden beim Deployment, durch den *EARExpander*, in EJB- und Web-Module expandiert, die sich dann in den Containern deployen lassen.

5.2 WebSphere administrative Komponenten

Nachfolgende Abbildung 45 zeigt das WebSphere Administrationsmodell. Die einzelnen physischen Maschinen werden *Nodes* genannt. Auf den Nodes laufen *Application Server* und *Administrative Server*. Alle Administrative Server speichern ihre administrativen Daten in einem gemeinsamen *Administrative Repository*, in Form einer relationalen Datenbank. Die Menge aller Nodes, die sich ein Administrative Repository teilen nennt, man *Administrative Domain*. Sie ist ein logischer Raum, der die Konfigurationen von vielen verschiedenen Objekten der WebSphere Umgebung enthält. Die WebSphere Ressourcen eines Nodes werden in der Administrative Domain als *Administrative Ressourcen* repräsentiert.

Administrative Benutzerschnittstellen, außerhalb der Administrative Domain, kommunizieren mit den administrativen Servern mit Hilfe des IIOP- oder HTTP-Protokolls. WebSphere Advanced Edition bietet die *Java Administrative Console* und IIOP. WebSphere Advanced Edition Single Server bietet lediglich die *HTTP Web Administrative Console*.

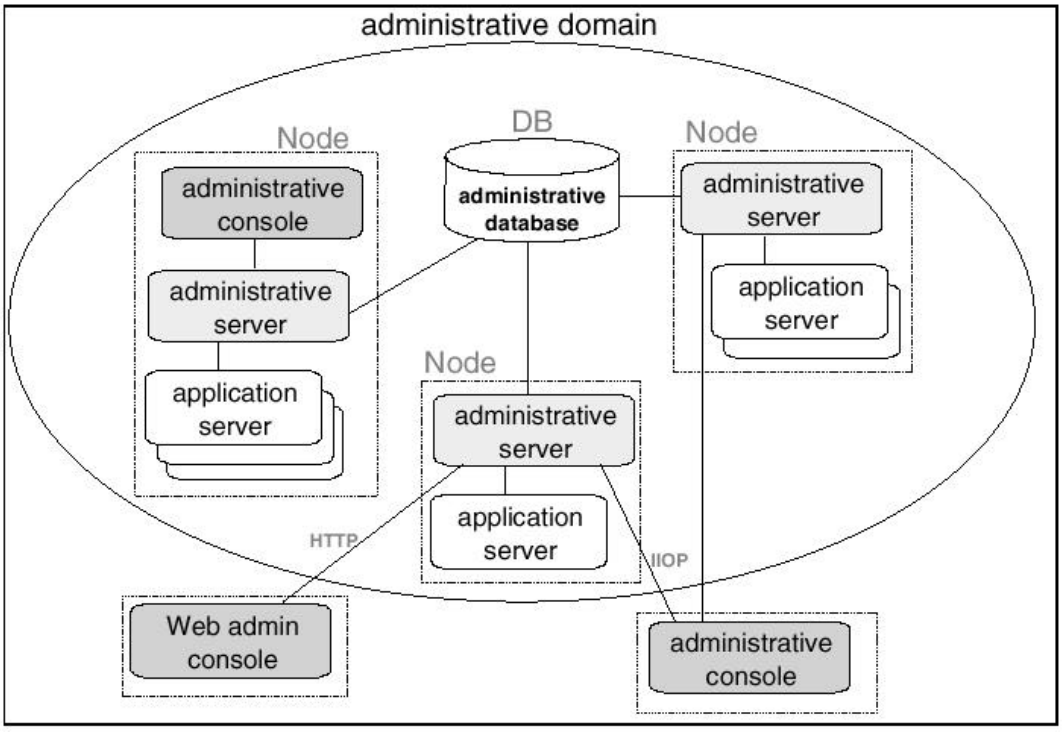

Abbildung 45: WAS Administrationsmodell aus [SG246176]

5.2.1 Administrative Server

Der *Administrative Server* (Administrationsserver) ist eine Laufzeitkomponente zur Systemverwaltung. In den meisten Fällen läuft der Administrationsserver auf allen Nodes einer Administrative Domain und kontrolliert die Interaktion zwischen den Nodes und den Applikationsserverprozessen. Er ist verantwortlich für die Verwaltung der Laufzeitumgebung, Sicherheit, Transaktionskoordination und Lastverteilung. Er erlaubt die zentrale Administration von allen verteilen Ressourcen einer Administrative Domain.

5.2.2 Administrative Repository

Alle Konfigurationsinformationen und globalen Sicherheitseinstellungen einer Administrative Domain werden in einem persistenten *Administrative Repository* gespeichert. Die gesamte Administration geschieht durch Manipulation der Objekte im Administrative Repository. Das Repository kann in DB2, Oracle, Informix, MS SQL Server oder Sybase Datenbanken gespeichert werden. In der Single Server Edition wird das Repository in einer XML-Konfigurationsdatei gespeichert.

5.2.3 Administrative Benutzerschnittstellen

Die Überwachung und Konfiguration von administrativen Ressourcen sowie das Starten und Stoppen der Server kann durch vier Benutzerschnittstellen, wie in der folgenden Abbildung 46 gezeigt, geschehen.

Java Administrative Console

Die *Java Administrative Console* oder auch WebSphere Administrative Console, kurz Admin Console, ist eine graphische Benutzerschnittstelle. Mit ihr sind in einer Administrative Domain alle administrativen Aktivitäten durchführbar. Sie kann auf jedem Node mit einem Administrationsserver ausgeführt werden.

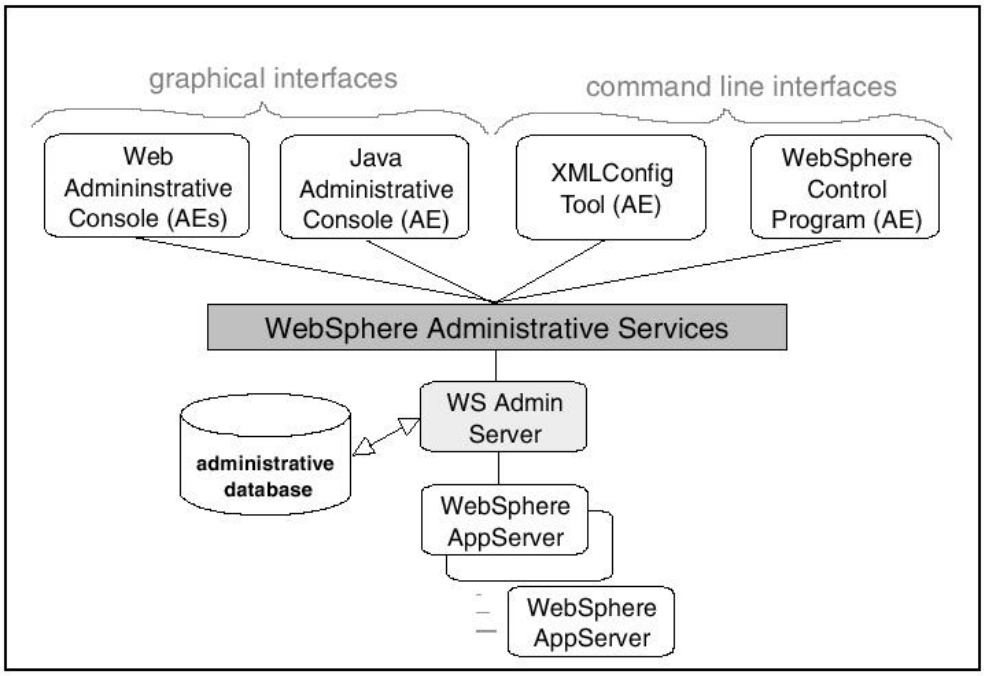

Abbildung 46: WebSphere administrative Benutzerschnittstellen aus [SG246176]

Web Administrative Console

Die *Web Administrative Console* ist eine webbasierte GUI, die nur in der Single Server Edition verwendet wird. Mit ihr lässt sich die XML-Konfigurationsdatei im Webbrowser bearbeiten.

WebSphere Control Program

Das *WebSphere Control Program* ist ein interaktives kommandozeilen-basiertes Administrationstool. Mit ihm lassen sich viele Aufgaben per Skript automatisieren. Es basiert auf Tcl (Tool Command Language), welche um Instruktionen zur Manipulation von WebSphere-Objekten erweitert wurde.

XMLConfig

Ebenfalls kommandozeilen-basiert ist *XMLConfig*. Es dient zum Import und Export von Konfigurationsdaten aus dem Administrative Repository. Mit ihm lassen sich wiederholende Änderungen im Repository vereinfachen und Konfigurationsdateien für das HTTP-Server Plug-In generieren. XMLConfig ist kein interaktives Tool und kann nicht zur Gewinnung von Statusinformationen von WebSphere verwendet werden.

DrAdmin

DrAdmin ist ein kommandozeilen-basiertes Tool, das hauptsächlich zum Troubleshooting verwendet wird.

6 WebSphere Security

In WebSphere wird zwischen *Global Security* und *Application Security* unterschieden:

Global Security bezeichnet zum einen eine Option, mit der alle Sicherheitsmechanismen aktiviert werden, und zum anderen die globalen Sicherheitskonfigurationen. Diese werden, bis auf wenige Ausnahmen, im Administrative Repository gespeichert. Wenn Global Security nicht aktiviert ist, sind die Sicherheitsmechanismen nicht aktiv und es werden keine Security Policies durchgesetzt. Der Server und die Applikationen sind dann vollkommen ungeschützt. Dies ist nur für Testzwecke sinnvoll.

Application Security bezeichnet die Sicherheitskonfigurationen einer Applikation, die, bis auf einige Ausnahmen, in den Deployment Descriptoren der Applikation gespeichert sind. Diese können Konfigurationen der Global Security überschreiben.

Die einzelnen Tools und die Speicherorte der Sicherheitsinformationen sind in Abbildung 47 dargestellt. Während der Assemblierungsphase der Applikation wird primär das Application Assembly Tool (AAT) eingesetzt. Die anderen Tools werden erst beim Deployment der Applikation und zur Administration des Systems eingesetzt.

Security information	Administration tool	Storage location
Global security, Authentication, SSL settings	Administrative console	Administrative repository
Application security roles	AAT	EAR DD application.xml
Security role binding to users, groups, special subjects	AAT, Administrative Console	EAR DD ibm-application-bnd.xmi, Administrative repository
EJB ► Security role ► Role reference ► Method permission	AAT	EJB DD ejb-jar.xml
EJB ► Security identity - RunAs	AAT	EJB DD ibm-ejb-jar-ext.xmi
EJB ► RunAs mapping	Administrative Console	EAR DD ibm-application-bnd.xmi
Web resources ► Security role ► Role reference ► Security constraints	AAT	WAR DD web.xml

Abbildung 47: Administration und Speicherort aus [SG246176]

6.1 WebSphere Sicherheitsarchitektur

Wie der letzte Abschnitt zeigte, ist der Administrative Server der zentrale Punkt der Sicherheit in WebSphere. Seine und die übrigen sicherheitsrelevanten Komponenten und Dienste werden im folgenden beschrieben und in Beziehung mit der J2EE-Spezifikation von SUN gesetzt.

Das Sicherheitsmodell von WebSphere wurde mit der Version 4.0 verändert, damit es den Anforderungen der J2EE 1.2, EJB 1.1 und Servlet 2.2 Spezifikationen genügt. Die J2EE-Kompatibilität wurde von SUN [J2EECOMP] bestätigt.

Die Sicherheitsarchitektur von WebSphere lässt sich in Schichten einteilen:

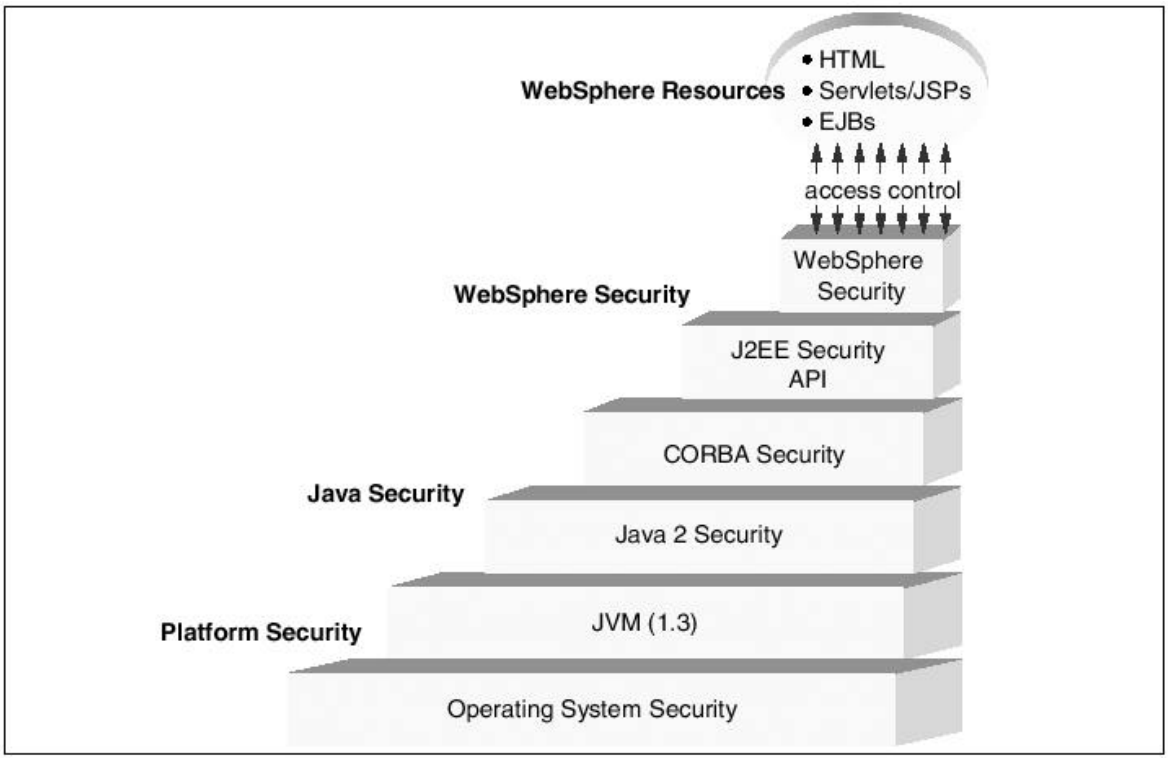

Abbildung 48: WebSphere Security Layers aus [SG246520]

Operating System Security: Die Sicherheitsmechanismen des Betriebssystems werden zum Schutz der sicherheitskritischen Dateien und zur Authentisierung von Benutzern mittels der *User Registry* des Betriebssystems verwendet.

Java Language Security: Die Java Virtual Machine und ihre Sicherheitsmechanismen werden von WebSphere und durch spezielle Sicherheitsklassen von Java genutzt.

CORBA Security: Die Sicherheitsmechanismen von CORBA werden für die Kommunikation zwischen Applikationen in sicheren ORBs durch den *Secure Association Service* (SAS) genutzt.

J2EE-Security: Der *Security Collaborator* setzt die J2EE-basierenden Security Policies durch und unterstützt J2EE-Security APIs.

WebSphere Security: Die Sicherheit in WebSphere beruht auf allen oben genannten Technologien und erweitert diese zum Teil. Security Policies für Web- und EJB-Ressourcen werden standardisiert durchgesetzt.

In WebSphere sind verschiedene sicherheitsrelevante Komponenten zu finden. Die folgende Abbildung 49 soll die Zusammenhänge verdeutlichen und als Übersicht dienen. Die einzelnen Komponenten werden anschließend erläutert.

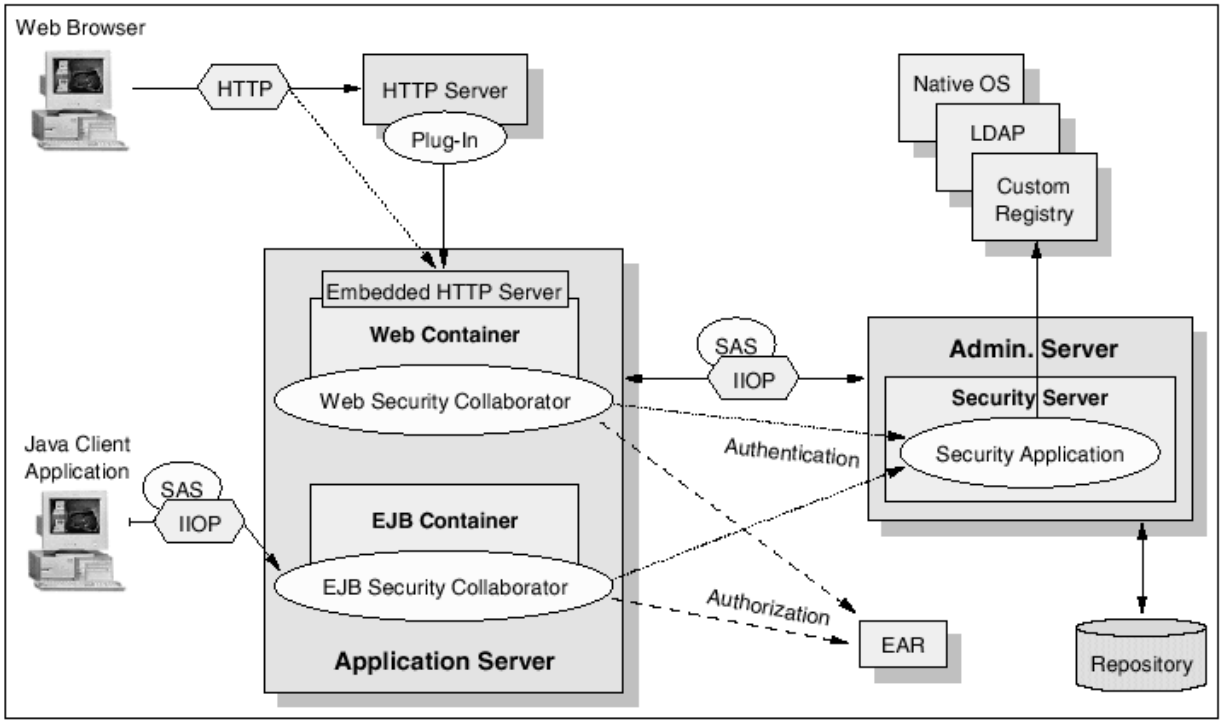

Abbildung 49: WebSphere Sicherheitsarchitektur aus [SG246176]

6.1.1 Web- und Java-Clients

Web-Clients und *Java-Clients* können auf Komponenten in WebSphere zugreifen. *Web-Clients* sind zum Beispiel Webbrowser, die über HTTP bzw. HTTPS mit Web-Komponenten kommunizieren. Applikationen und Applets sind *Java-Clients*, die über IIOP (über den SAS[1]) mit EJB-Komponenten kommunizieren. Bei beiden Clienttypen wird die Kommunikation von den Containern bzw. den Security Collaborators der Container kontrolliert, die ihrerseits mit dem Security Server zusammenarbeiten.

6.1.2 Security Collaborators

Security Collaborators sind Prozesse des Application Servers. Sie sind zur Laufzeit die wichtigsten Komponenten zur Durchsetzung der, im Deployment Descriptor spezifizierten, Security Constraints und Method-Permissions. Jeder Container besitzt seinen eigenen Security Collaborator.

[1] siehe Kapitel: „Secure Association Service“

Der *Web-Collaborator* bietet Authentisierung, Autorisierung und Logging bzw. Tracing von sicherheitsrelevanten Informationen. Er authentisiert Web-Clients mit dem Mechanismus, der im *login-configuration* Element des Deployment Descriptors (web.xml) angegeben ist.

Der *EJB-Collaborator* bietet Autorisierung, Unterstützung von User-Registries, Logging bzw. Tracing von sicherheitsrelevanten Informationen und Delegation nach der Delegation Policy. Er selbst führt keine Authentisierung durch. Diese delegiert er an den Secure Association Service (SAS). Nach der Autorisierung setzt der EJB-Collaborator die run-as Identität basierend auf der Delegation Policy. Die Delegation Policy bestimmt die Identität, unter der die Enterprise Bean ausgeführt wird. Sie wird auch zum Rufen von Methoden anderer Enterprise Beans verwendet. Die Delegation Policy bzw. der run-as Modus wird im ejb-jar.xml Deployment Descriptor spezifiziert.

Anmerkung: WebSphere v4.0 entspricht der J2EE1.2-Spezifikation. Delegation wird für Web-Ressourcen nicht gefordert! Dies erschwert die Entwicklung und das Deployment von Applikationen. Die Zugriffspfade müssen, unter dem Aspekt der Zugriffsrechte, genau definiert werden.

6.1.3 Security Server

Der *Security Server* ist Teil des Administrationsservers. Er unterstützt die Administration der Sicherheit und bietet den Web- und EJB-Containern: Authentisierung, Autorisierung und Delegation Policies. Zur Laufzeit enthält der Security Server eine *Security Application*, welche den Security Collaborators Dienste zur Authentisierung bereitstellt. Die verfügbaren Authentisierungsmechanismen werden im Kapitel „User-Registries“ ausführlich dargestellt. Zum Beispiel bietet der Security Server, bei der Verwendung von *Lightweight Third Party Authentication* (LTPA), einen Token-Dienst oder er konsultiert einen *Lightweight Directory Access Protocol* (LDAP) Server. Seine Sicherheitskonfigurationen werden im Administrative Repository gespeichert.

6.1.4 Security Policies

Die Sicherheitsattribute von Enterprise Beans und Web-Applikationen werden, wie in der J2EE-Spezifikation spezifiziert, in ihren Deployment Descriptoren definiert.

6.1.5 User-Registries

In einer *User-Registry* werden die Sicherheitsattribute von verschiedenen Entitäten gespeichert. Oft sind dies Verzeichnisdienste oder Datenbanken. Die Security Application kann verschiedene User-Registries zur Authentisierung verwenden, die sich wie folgt gruppieren lassen:

- **Lokale Registries,** die auf Umgebungen mit einen einzelnen Application Server auf einem einzelnen Node oder einem Windows NT Domain Controller beschränkt sind.
- **Zentralisierte Registries,** welche das LTPA Protokoll für den Zugriff auf eine der folgenden Registries nutzen:
    - **LDAP Registry** auf einem unterstützten LDAP Server.
    - **Custom User Registry** unter Verwendung des WebSphere Custom Registry Interfaces.

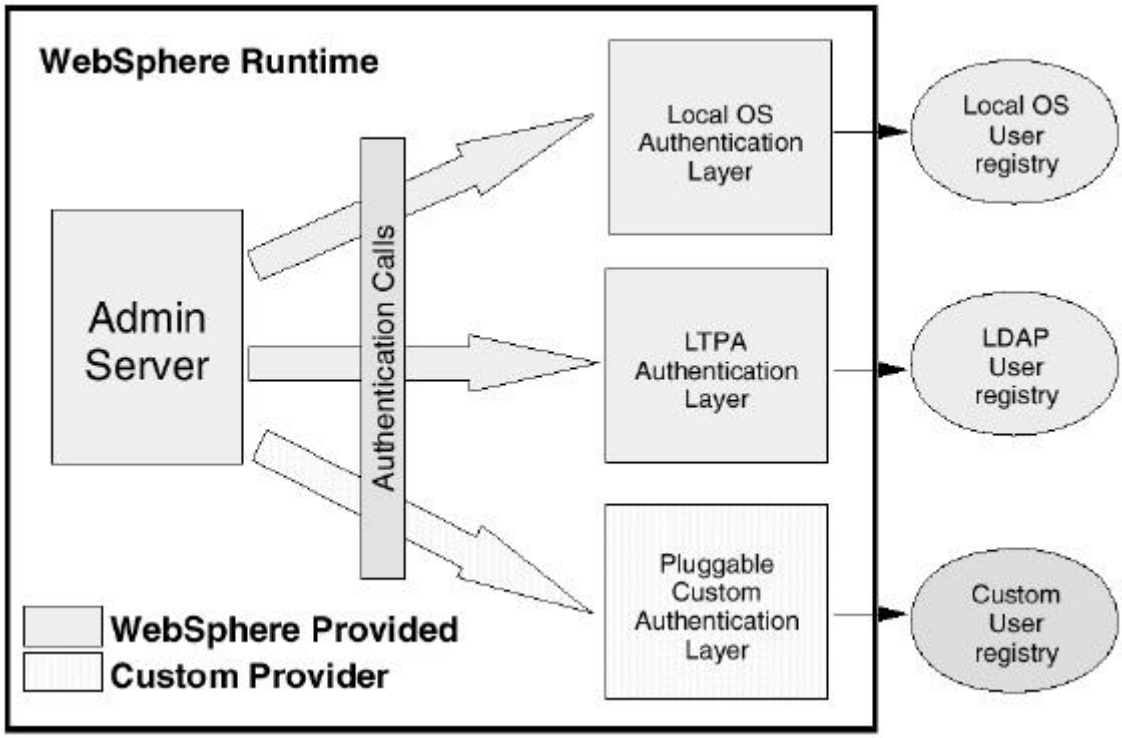

Abbildung 50: WAS User Registry Optionen aus [SG246176]

WebSphere unterstützt folgende lokale Registries:

- AIX
- HP-UX
- Linux
- Solaris
- Windows 2000
- Windows NT (Domain, Workgroup)

Von WebSphere unterstützte zentralisierte (LTPA) Registries:

- Custom User Registry (nur in der Advanced Edition)
- Domino 4.6, 5.0
- IBM Secure Way Directory
- LDAP (nur in der Advanced Edition)
- Lightweight Third Party Authentication (LTPA)
- Netscape Directory Server
- Windows 2000 Active Directory

Die *Custom User Registry* liegt in der Verantwortlichkeit der Entwickler. Jedes System, das eine Java-Schnittstelle besitzt, kann mit der WebSphere Custom Registry SPI angeschlossen werden. So lassen sich Produkte, wie zum Beispiel DB2, MQSeries, und Kombinationen von mehreren Registries, wie LDAP und RACF[1], als User Registry einsetzen. Abbildung 51 zeigt die, von WebSphere unterstützten, Authentisierungsmöglichkeiten. Die Betriebssysteme unterstützen die einfache und formular-basierte Authentisierung. LTPA mit LDAP oder Custom User Registries bietet zusätzlich eine zertifikat-basierte Authentisierung.

[1] IBM Resource Access Control Facility [RACF]

	OS (Unix)	OS (NT)	LDAP	Custom
Basic	Authentication using system calls	Authentication using security Access Manager through system calls	An LDAP search is performed.	A password check is performed against the custom registry.
Form-Based	Authentication using system calls	Authentication using security Access Manager through system calls	An LDAP search is performed.	A password check is performed against the custom registry.
Certificate	N/A	N/A	The certificate content is a credential mapped to an LDAP entry (based on trust of the Web server).	The certificate content is a credential mapped to a custom entry (based on trust of the Web server).
Digest	N/A	N/A	N/A	N/A

Abbildung 51: Authentisierungsmechanismen in WebSphere aus [SG246520]

Abbildung 52 zeigt die Authentisierungsmöglichkeiten für verschiedene Clienttypen. Web- und Java-Clients können keine oder eine einfache Authentisierung nutzen. Bei der einfachen Authentisierung kann LTPA oder das lokale Betriebssystem verwendet werden. Bei LTPA besteht die Wahl zwischen LDPA und einer Custom User Registry. Web-Clients können auch eine zertifikat-basierte LDPA Authentisierung und eine formular-basierte Authentisierung durchführen.

Challenge type	Authentication mechanism	User registry	Client
None	None	None	Web / Java
Basic	LTPA	LDAP	Web / Java
		Custom	Web / Java
	OS	OS	Web / Java
Certificate	LTPA	LDAP	Web
		Custom	Web

Challenge type	Authentication mechanism	User registry	Client
Form Login	LTPA	LDAP	Web
		Custom	Web
	OS	OS	Web

Abbildung 52: Authentisierung von Web- und Java-Clients aus [SG246520]

6.2 Web Single Sign-On

Die J2EE-Spezifikation fordert eine Unterstützung von Single Sign-On (SSO). WebSphere unterstützt SSO durch den Lightweight Third Party Authentication (LTPA) Mechanismus.

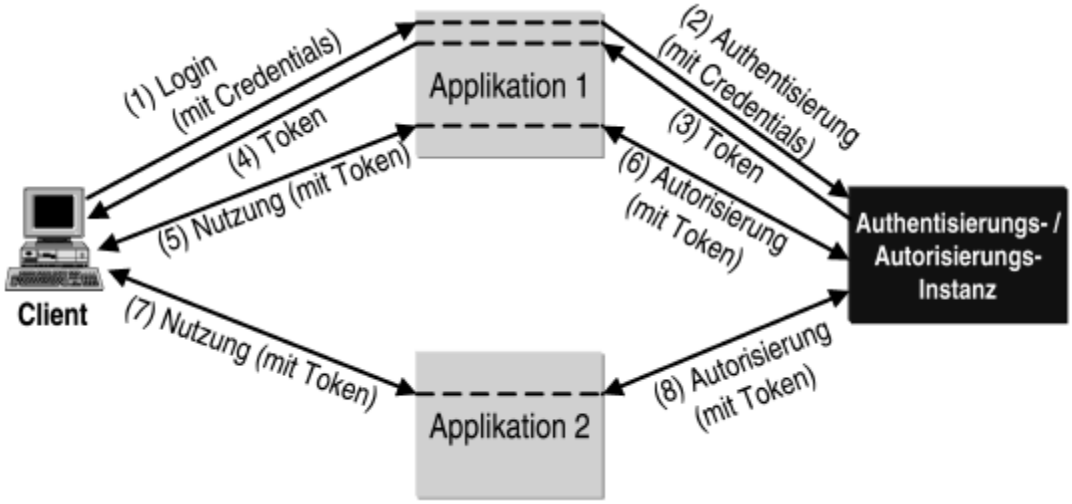

Abbildung 53: token-basierter SSO aus [Dormanns02]

LTPA verwendet einen *LTPAToken*, das u.a. folgende Elemente besitzt:

- Benutzer-Authentisierungsinformationen.
- Netzwerkdomänenname, in welcher das Token gültig sein soll.
- Verfallsdatum, nachdem der Benutzer sich erneut Authentisieren muss.

Nach einer erfolgreichen Authentisierung wird das Token, in einem proprietären Format, erstellt und mit einem LTPA Schlüssel proprietär verschlüsselt, der von allen am SSO beteiligten Server gemeinsam verwendet wird. Der Benutzer erhält das Token in Form eines *transienten Cookie* (Session-Cookie) mit der Bezeichnung *LtpaToken*, welches nur im Speicher des Browsers existiert und nicht persistent auf dem Computer des Benutzers gespeichert wird. Er verfällt, wenn der Browser geschlossen wird. SSO über LTPA kann zwischen WebSphere Servern, zwischen Domino Servern und zwischen WebSphere und Domino Servern eingesetzt werden. Die Voraussetzungen für den Einsatz von SSO sind:

- Die Verwendung einer gemeinsamen Registry (LDAP-Server) zur Authentisierung.
- Alle an SSO teilhabenden Server müssen sich in der gleichen DNS-Domäne befinden.
- Die URLs müssen den Namen der DNS-Domäne explizit beinhalten.
- Der Browser des Benutzers muss Session-Cookies akzeptieren.
- Die Zeiten und Zeitzonen der Server müssen korrekt und synchronisiert sein, denn das Verfallsdatum des Tokens ist absolut.
- Alle Server müssen die gleichen LTPA Schlüssel verwenden.

6.3 Secure Sockets Layer (SSL)

Die Kommunikation zwischen Clients, Applikationsservern, Administrationsservern, Verzeichnisdiensten und externen Ressourcen kann in WebSphere mit SSL gesichert werden. WebSphere unterstützt SSL 2.0, SSL 3.0, und SSL 3.1 bzw. TLS 1.0. Dies entspricht den Anforderungen der J2EE-Spezifikation. Eine Aufstellung unterstützter Cipher Suites findet sich unter [WAS40DOCS]. Cipher Suites sind normierte Bezeichnungen. Zum Beispiel bezeichnet SSL_RSA_WITH_RC4_128_MD5 das SSL Protokoll mit RSA zum Schlüsselaustausch, einer Verschlüsselung mit RC4 und 128 Bit Schlüssellänge und der Einweg-Hashfunktion MD5.

Zur Erleichterung der Administration teilt WebSphere die Cipher Suites in verschiedene Sicherheitsstufen (HIGH, MEDIUM, LOW) ein. Die Angaben möglicher Schlüssel ($x*10^y$) habe ich [WASSSL] entnommen:

In der Sicherheitsstufe HIGH werden die folgenden Cipher Suites unterstützt:

SSL_RSA_WITH_RC4_128_MD5	$(3,4*10^{38})$
SSL_RSA_WITH_RC4_128_SHA	$(3,4*10^{38})$
SSL_RSA_WITH_DES_CBC_SHA	$(7,2*10^{16})$
SSL_ RSA_WITH_3DES_EDE_CBC_SHA	$(3,7*10^{50})$
SSL_DHE_RSA_WITH_DES_CBC_SHA	$(7,2*10^{16})$
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA	$(3,7*10^{50})$
SSL_DHE_DSS_WITH_DES_CBC_SHA	$(7,2*10^{16})$
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	$(3,7*10^{50})$

In der Sicherheitsstufe MEDIUM werden die folgenden Cipher Suites unterstützt:

SSL_RSA_EXPORT_WITH_RC4_40_MD5	$(1,1*10^{12})$
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	$(1,1*10^{12})$
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	$(1,1*10^{12})$
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	$(1,1*10^{12})$
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	$(1,1*10^{12})$

Ist der Security Level LOW und handelt es sich um eine Server-Konfiguration, so sind die unterstützten Cipher Suites:

SSL_RSA_WITH_NULL_MD5	(0)
SSL_RSA_WITH_NULL_SHA	(0)
SSL_DH_anon_WITH_RC4_128_MD5	$(3,4*10^{38})$
SSL_DH_anon_WITH_DES_CBC_SHA	$(7,2*10^{16})$
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA	$(3,7*10^{50})$
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5	$(1,1*10^{12})$
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA	$(1,1*10^{12})$

Ist der Security Level LOW und handelt es sich um eine Client-Konfiguration, so sind die unterstützten Cipher Suites:

SSL_RSA_WITH_NULL_MD5	(0)
SSL_RSA_WITH_NULL_SHA	(0)

In WebSphere lässt sich die gesamte Kommunikation (HTTP, LDAP und IIOP) durch SSL sichern, dies setzt eine Aktivierung der *Global Security* voraus. Abbildung 54 zeigt die gesicherten Protokolle.

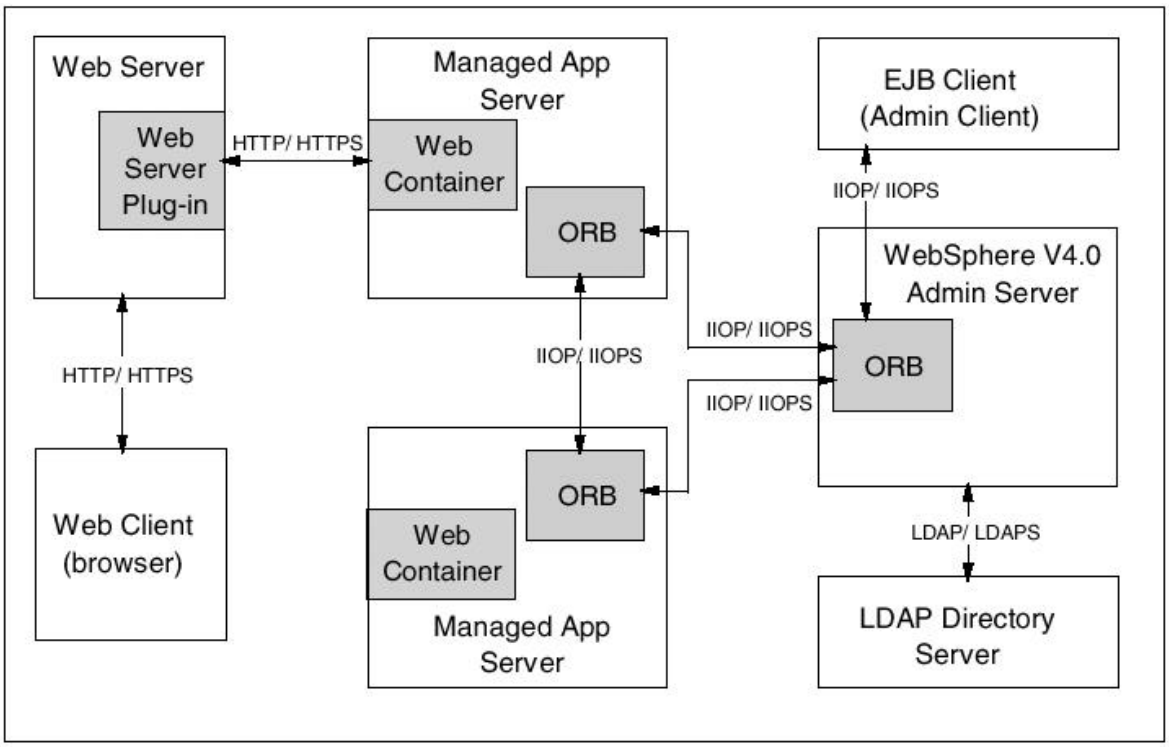

Abbildung 54: Gesicherte Protokolle in WAS40AE aus [SG246520]

6.4 Secure Association Service (SAS)

Die Kommunikation zwischen EJB-Clients und EJBs sichert WebSphere durch den *Secure Association Service* (SAS). Dieser sollte nicht mit dem CORBA Security Attribute Service des OMG CSIv2 Protokolls verwechselt werden, den WebSphere in der Version 5 unterstützt.

Die Recherche zu diesem Thema war nicht einfach. Die Dokumentation von IBM-SAS ist sehr minimal gehalten. Es stellte sich die Frage, ob IBM-SAS und CSIv2-SAS äquivalent sind. Schließlich konnte ich Kontakt zu Rob High aufnehmen. Er ist ein „Distinguished Engineer“ aus dem „WebSphere Architecture and Development Team“ und war maßgeblich an den Spezifikationen beider Protokolle beteiligt. Der Antrag an OMG zur Spezifikation von CSIv2 basierte auf der IBM-SAS Implementation, die von Rob High entwickelt wurde. Nach Auskunft von Rob High sei CSIv2-SAS ein Dienst, der für die Bereitstellung von Privilegeinformationen von Prinzipalen verantwortlich ist, wie zum Beispiel die Gruppenzugehörigkeit, Sicherheitsstufe (Security Clearence) und den Rollen, die mit dem Prinzipal assoziiert sind.

IBM-SAS sei ein proprietäres Protokoll, das vor CSIv2-SAS für den Einsatz im IBM Component Broker entwickelt wurde, um eine Sichere Assoziation (Secure Association) zwischen zwei verteilten Endpunkten zu etablieren. IBM-SAS wird für die Authentisierung von Prinzipalen in kommunizierenden Prozessen verwendet und ist ab Version 3 Teil von WebSphere. Es besitzt einen einfachen token-basierten Authentisierungsmechanismus. Zur Authentisierung ruft der Client vor dem ersten Methodenaufruf eine *non_existent* Methode, die vom Server-ORB bereitgestellt wird und nichts macht. Im Rahmen dieser ersten Anfrage wird der Client authentisiert.

Anfragen von Clients an Enterprise Beans werden als RMI/IIOP Nachrichten über einen Object Request Broker (ORB) zum Server, der die Enterprise Beans hostet, gesendet. Bei jeder Anfrage oder Antwort ruft der ORB client- und serverseitig den Secure Association Service. Abbildung 55 zeigt die Interaktion zwischen dem EJB-Client und der EJB, wenn *Global Security* aktiviert ist.

Auf der Seite des Clients fängt SAS die Anfrage ab (Interception), sammelt die Security Credentials, fügt sie der Anfrage hinzu und sendet die Anfrage.

Auf der Seite des Servers fängt SAS die Anfrage erneut ab, extrahiert den Security Context, authentisiert den Client und sendet die Anfrage an den EJB-Container. Der EJB-Container autorisiert die Anfrage und reicht sie an die EJB weiter. Die Antwort bewegt sich auf dem gleichen Weg über die SAS Interceptors zurück. Die Kommunikation zwischen den beiden ORBs ist durch SSL gesichert.

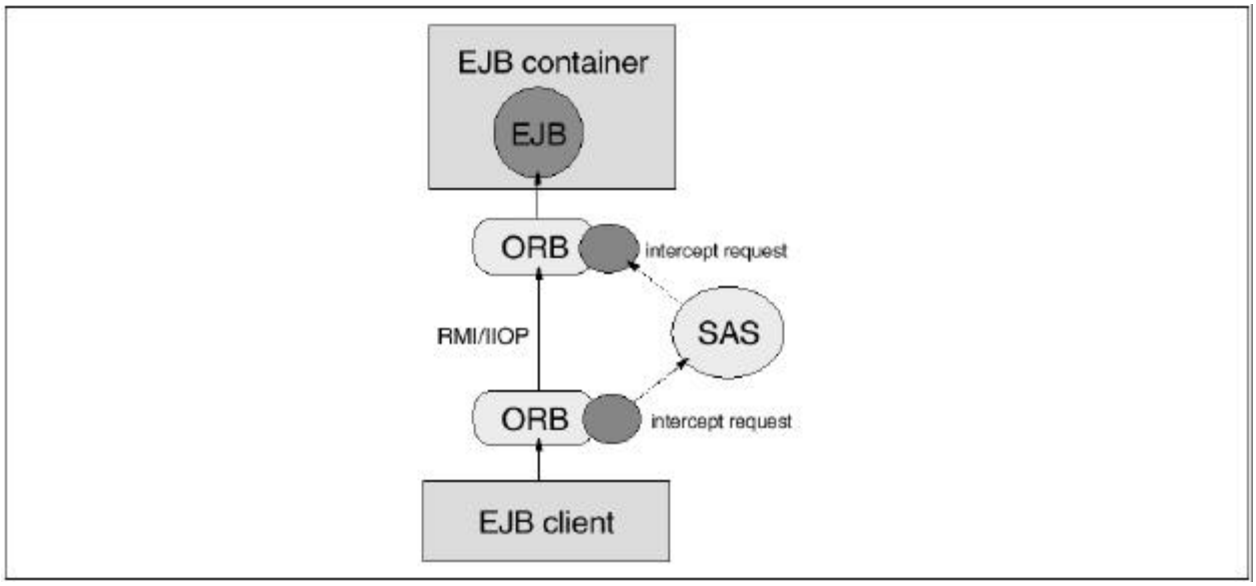

Abbildung 55: SAS Interception aus [SG246520]

6.5 WebSphere Delegationsmodell

Die Delegation von Prinzipalen wird in Websphere für EJB-Ressourcen unterstützt. Bei Web-Ressourcen ist keine Delegation möglich. Das WebSphere Delegationsmodell ist eine Erweiterung der EJB 1.1 Spezifikation und orientiert sich an der EJB 2.0 Spezifikation. Es erfüllt daher die Anforderungen der J2EE-Spezifikation.

Enterprise Beans können eine Delegation Policy besitzen. Der Application Assembler definiert in der Delegation Policy ein *SecurityIdentity* Element, das folgende Werte enthalten kann:

- *UseCallerIdentity* – Die Komponente nutzt die Identität seines Aufrufers.
- *UseSystemIdentity* - Die Komponente nutzt die Identität seine Containers.
- *RunAsSpecifiedIdentity* - Die Komponente nutzt eine spezifizierte Identität.

Wenn kein Wert gesetzt wurde, wird die *UseCallerIdentity* Konfiguration verwendet.

6.6 WebSphere Authentisierungsmodell

6.6.1 Authentisierung auf Ebene der Web-Container

Zugriffe auf Web-Ressourcen im Web-Container werden durch die folgenden Mechanismen authentisiert:

- Verzögerte Authentisierung (Lazy Authentication)
- Einfache HTTP Authentisierung
- Formular-basierter Login
- Zertifikat-basierte Authentisierung (X.509 Zertifikat und SSL mit wechselseitiger Authentisierung).

Die in der Servlet-Spezifikation erwähnte, aber nicht geforderte, HTTP Digest Authentication wird von Websphere nicht unterstützt. Die Anforderungen der J2EE-Spezifikation sind erfüllt. Nicht authentisierte Benutzer erhalten einen *Anonymous Credential* mit dem intern reservierten Prinzipal *UNAUTHENTICATED*.

Der Authentisierungsmechanismus wird für jede Web-Applikation individuell definiert. Daher kann jede Web-Applikation in einer Enterprise-Applikation einen anderen Authentisierungsmechanismus verwenden.

6.6.2 Authentisierung auf Ebene der EJB-Container

Die J2EE1.1-Spezifikation definierte keine Authentisierungsmechanismen, die ein EJB-Container unterstützen muss. Entsprechende Anforderungen werden erst in der J2EE1.2-Spezifikation spezifiziert.

In WebSphere können J2EE Application Clients, Thin Java Applications, Applets, Servlets, JSPs und EJBs auf EJB-Container zugreifen. Eine Authentisierung dieser Clients ist möglich, denn alle Clients kommunizieren mit dem EJB-Container über IIOP, daher wird IBM-SAS zur Authentisierung eingesetzt. Die Authentisierungsdaten (Benutzername, Passwort oder Zertifikat im Key File) werden clientseitig (*client side login*) bzw. serverseitig (*server side login*) einer CORBA ORB Datenstruktur (Security Context) übergeben. Diese Übergabe muss der Entwickler im Programmcode der Anwendung programmieren. Die Programmierung wird durch die Klassen *LoginHelper* (clientseitig) und *ServerSideAuthenticator* (serverseitig) erleichtert, denn diese Klassen wrappen die verwendeten CORBA-Klassen, so dass nur geringe CORBA Kenntnisse notwendig sind. Anschließend führt der CORBA Security Service die Authentisierung, anhand der Authentisierungsdaten im Security Context des ORBs, durch. Diese Authentisierung kann er an den Security Server delegieren.

Die folgende Abbildung 56 zeigt eine Zusammenfassung der Authentisierungsmechanismen für die verschiedenen Client-Typen.

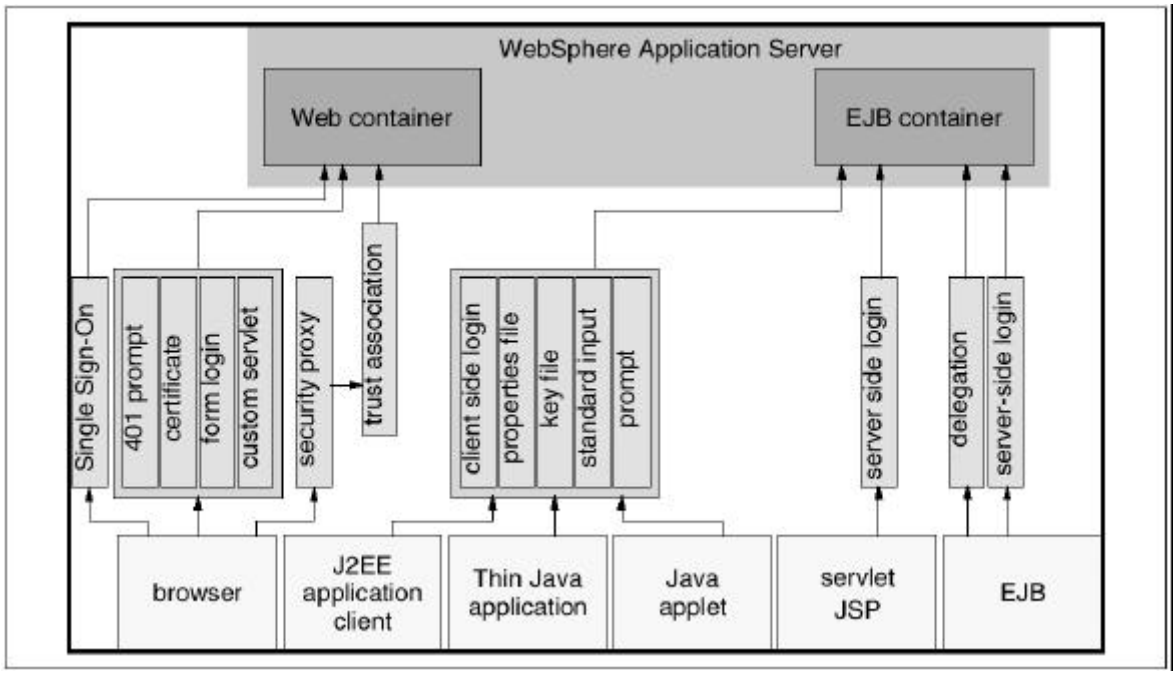

Abbildung 56: Authentisierungsmechanismen aus [SG246520]

6.7 WebSphere Autorisierungsmodell

WebSphere besitzt ein rollen-basiertes Autorisierungsmodell, das der J2EE-Spezifikation entspricht.

6.7.1 Deklarative Autorisierung

Die Security Collaborators setzen die Zugriffskontrolle, anhand der Security Constraints und Method-Permissions der Deployment Descriptoren, durch. Der EJB-Collaborator delegiert die Authentisierung an den Secure Association Service (SAS). Alle EJB- und Web-Ressourcen sind per Voreinstellung ungeschützt, wenn Global Security aktiviert ist. Eine Ressource wird geschützt, wenn mindestens ein Security Constraint bzw. eine Method-Permission für diese Ressource definiert ist. Alle anderen Ressourcen bleiben ungeschützt!

6.7.2 Programmatic Authorization

Die J2EE-Spezifikation fordert eine Möglichkeit zur *Programmatic Authorization*, also einer Autorisierung im Programmcode der Applikation.

In Servlets werden die folgenden Methoden vom WebSphere Application Server unterstützt:

- *getUserPrincipal* oder *getRemoteUser* – liefert den Namen des Prinzipals
- *isUserInRole* – Test auf Rollenzugehörigkeit

In EJBs werden die folgenden Methoden vom WebSphere Application Server unterstützt:

- *getCallerPrincipal* – liefert den Namen des Prinzipals
- *isCallerInRole* – Test auf Rollenzugehörigkeit

6.8 WebSphere Class Loaders

WebSphere nutzt mehrere Class Loader und das Java 2 Parent Delegation Model. Die einzelnen Class Loader sind in Abbildung 57 dargestellt und werden im folgenden erläutert. Die Klassenhierarchie ist nicht abgebildet, da sie von IBM nicht dokumentiert wurde.

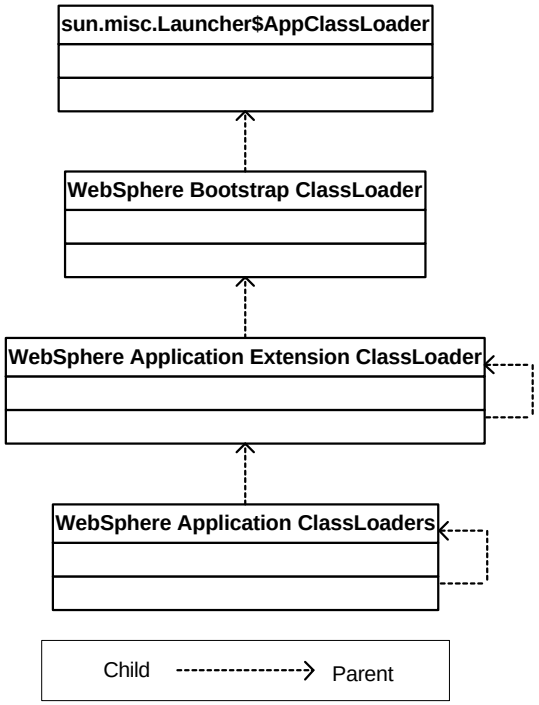

Abbildung 57: WebSphere Class Loader

Websphere Bootstrap Classloader

Der *Bootstrap Classloader* wird während der Aktivierung des Administrationsservers erzeugt. Sein Parent ist der Java System Classloader, der Applikationen und Klassen im *CLASSPATH* lädt. Der Bootstrap Classloader ist für das Laden der Klassen aus den folgenden Unterverzeichnissen verantwortlich (WAS_HOME ist das Installationsverzeichnis von WebSphere):

- <*WAS_HOME*>*classes* (Runtime Class Patches Directory, oder RCP)
- <*WAS_HOME*>*lib* (Runtime Classpath Directory, oder RC)
- <*WAS_HOME*>*lib**ext* (Runtime Extensions Directory, oder RE)

Neben den Basisklassen von WebSphere lädt dieser Class Loader Ressourcen, die vom Applikationsserver verwendet werden, wie zum Beispiel Datenbankressourcen. Der Classpath des Bootstrap Classloaders wird durch die Systemeigenschaft *ws.ext.dirs* bestimmt.

Das RCP Verzeichnis enthält Patches, die zur Laufzeit eingespielt werden können. Das RC Verzeichnis enthält die Hauptklassen der WebSphere Laufzeitumgebung, die durch Klassen im Verzeichnis RE erweitert werden kann.

Der Bootstrap Classloader sucht Klassen zuerst im Verzeichnis RCP, dann im Verzeichnis RC und abschließend im Verzeichnis RE (RCP -> RC -> RE).

WebSphere Application Extensions Classloader

Der *Application Extensions Classloader* ist Child vom WebSphere Bootstrap Classloader und wird während der Aktivierung des Administrationsservers erzeugt. Er durchbricht das Parent Delegation Model, da er in einem selbstdelegierenden Modus arbeitet, d.h. er sucht zuerst selbst nach einer Klasse. Kann er die Klasse nicht laden, so delegiert er den Auftrag an seinen Parent. Der Application Extensions Classloader lädt Klassen aus dem Verzeichnis:

- <*WAS_HOME*>*lib**app* (Application Extensions, oder AE)

Dieses Verzeichnis enthält die Klassen, die sich alle auf dem Server deployten J2EE-Applikationen teilen. Diese Klassen sind für den Bootstrap Classloader nicht sichtbar, da er in der Hierarchie höher liegt.

WebSphere Application Classloaders

Ein *Application Classloader* ist Child des Application Extensions Classloaders und wird beim Starten des Application Servers erzeugt. Für jedes deployte Modul eines Application Servers wird ein Application Classloader erzeugt. Daher ist die Tatsache, dass sich mehrere Applikationen eine JVM teilen, nicht kritisch. Die einzelnen Applikationen werden von verschiedenen Classloadern geladen und isoliert. Die Delegation des Ladevorgangs hängt vom Typ des Moduls ab.

Bei Web-Modulen wird zuerst im Modul selbst gesucht. Nach einer erfolglosen Suche delegiert der Application Classloader den Auftrag an den Application Extensions Classloader. Dieser versucht ebenfalls zuerst selbst die Klasse zu laden und delegiert dann an den Bootstrap Classloader (eigenes Modul ->AE->RCP->RC->RE). Innerhalb des Moduls (WAR-Archivdatei) wird nach einer Klasse in folgender Reihenfolge gesucht:

1. Zuerst im Wurzelverzeichnis (*root*) für Klassenbibliotheken der WAR-Datei.

2. Im *WEB-INF/classes* Verzeichnis der WAR-Datei.

3. In den Klassenbibliotheken des *WEB-INF/lib* Verzeichnisses der WAR-Datei.

4. In den Klassenbibliotheken, die durch die *Class-Path Directive* im *MANIFEST* der WAR-Datei spezifiziert wurden.

Bei EJB-Modulen wird die Suche zuerst an den Application Extensions Classloader delegiert. Findet er die Klasse nicht, delegiert er die Suche an den Bootstrap Classloader. Findet der Bootstrap Classloader die Klasse nicht, sucht der Application Classloader die Klasse im EJB-Modul. Es wird in den Verzeichnissen AE->RCP->RC->RE und anschließend im EJB-Modul selbst nach der Klasse gesucht. Innerhalb der EJB JAR-Datei werden die Klassen in folgender Reihenfolge gesucht:

1. Zuerst im Wurzelverzeichnis der EJB JAR-Datei.

2. In den Klassenbibliotheken, die durch die *Class-Path Directive* im *MANIFEST* der EJB JAR-Datei spezifiziert wurden.

Die Delegation des Application Classloaders kann durch die Definition zweier Systemeigenschaften, für jeden einzelnen Application Server, konfiguriert werden:

com.ibm.ws.classloader.warDelegationMode (Grundeinstellung: *false*)
com.ibm.ws.classloader.ejbDelegationMode (Grundeinstellung: *true*)

Diese Systemeigenschaften können die Werte *true* oder *false* annehmen. Die Suchreihenfolge ändert sich wie folgt:

true: AE -> RCP -> RC -> RE -> im Modul selbst.
false: im Modul selbst -> AE -> RCP -> RC -> RE.

Die Sichtbarkeit der Module kann in einem Application Server durch die Eigenschaft *Module Visibility* konfiguriert werden. Die vier möglichen Werte sind:

Server: Alle Application Classloaders eines Systems können sich untereinander sehen. Die Suchreihenfolge entspricht der Reihenfolge, in der die Module im System initialisiert wurden.

Application: Alle Classloader einer J2EE-Applikation können sich untereinander sehen. Die Suchreihenfolge entspricht der Reihenfolge, in der die Module in der *application.xml* Datei der EAR-Archivdatei definiert wurden.

Module: Jedes Modul (EAR, JAR, WAR) besitzt seinen eigenen Classloader. Die Module können sich untereinander nicht sehen und keine Klassen teilen. Eine Sichtbarkeit zwischen den Modulen einer Applikation kann nur hergestellt werden, wenn im *MANIFEST Class-Path* Einträge hinzugefügt werden.

Compatibility: Diese Einstellung kann für die Migration von Applikationen genutzt werden, die vorher auf WebSphere v3.5 oder v3.0.2 deployed waren. In diesem Modus können sich alle Classloader von EJB-Modulen untereinander sehen. Das gleiche gilt für die Classloader von Web-Modulen. Die Suchreihenfolge der EJB Classloader wird durch die Reihenfolge der Initialisierungen der EJB-Module bestimmt.

IBM empfiehlt die Sichtbarkeiten: *Module* und *Application*.

In dem, von IBM in WebSphere verwendeten, Delegationsmodell sehe ich folgende Schwierigkeiten:

1. Delegiert der Bootstrap Classloader an den Java System Class Loader? Dies wird in der dokumentierten Suchreihenfolge nicht berücksichtigt.

2. Der Application Extensions Classloader und der Application Classloader delegieren erst nach einem eigenen Ladeversuch an ihren Parent. Dies entspricht nicht der klassischen Delegation in Java. Es stellt sich die Frage, ob man bei WebSphere von einem echten Java 2 Parent Delegation Model sprechen kann.

3. Die Suchreihenfolge und Sichtbarkeit der Module kann durch eine entsprechende Konfiguration beeinflusst werden. Dies erschwert die Portierbarkeit einer Applikation.

6.9 WebSphere Firewall-Topologien

WebSphere wird in den unterschiedlichsten Umgebungen mit verschiedenen Skalierungsstrategien eingesetzt. Genauso unterschiedlich sind auch die denkbaren Firewall-Topologien. In der Praxis finden sich häufig Topologien mit einer horizontalen Skalierung, die einen Network Dispatcher (Reverses Proxy) erfordert, und komplexe Firewall-Topologien, in denen mehrere Protokoll-Firewalls (IP-Router) vergattern werden. Die möglichen Konfigurationen können, im Hinblick auf den Umfang dieser Arbeit, nicht vollständig dargestellt werden.

Abbildung 58 zeigt eine relativ einfache Firewall-Topologie (*Screened Host Gateway*) ohne horizontale Skalierung, die sich in externes Netz, *DMZ* (*Demilitarized Zone*) und internes Netz einteilen lässt. Die Bereiche werden durch je eine Firewall getrennt. Im externen Netz (z.B. Internet) befinden sich Web- und Java-Clients sowie eine Public Key Infrastruktur. In der DMZ befindet sich der Webserver und in einigen Fällen auch der Applikationsserver. In diesem Beispiel befinden sich der Applikationsserver, Datenbanken, EISs sowie Verzeichnis- und Sicherheitsdienste im internen Netz.

Die *Protocol-Firewall* bzw. der *Screening Router* bietet einen Schutz auf Netzwerkebene. Er filtert den ein- und ausgehenden Netzwerkverkehr nach Protokollen (TCP, UDP) und Ports. Üblicherweise wird ein IP-Router als Protokoll-Firewall eingesetzt, der i.d.R. kein *NAT* (*Network Address Translation*) unterstützt, daher sind die Netzwerkadressen der Hosts in der DMZ nach außen sichtbar.

Die *Domain Firewall* bzw. *Application Gateway* ist häufig ein speziell konfigurierter Rechner (*Bastionshost*), der einen Schutz auf Netzwerk- und Applikationsebene bietet. Der Application Gateway bietet für jede Applikation so genannte Proxy Dienste, die eine Stellvertreterrolle einnehmen und verschiedene Sicherheitsdienste implementieren, wie zum Beispiel Benutzerauthentisierung oder Auditing. Wenn der Application Gateway zwei Netzwerkschnittstellen besitzt, spricht man auch von einem *Dual-Homed Gateway*, der dann auch NAT unterstützt.

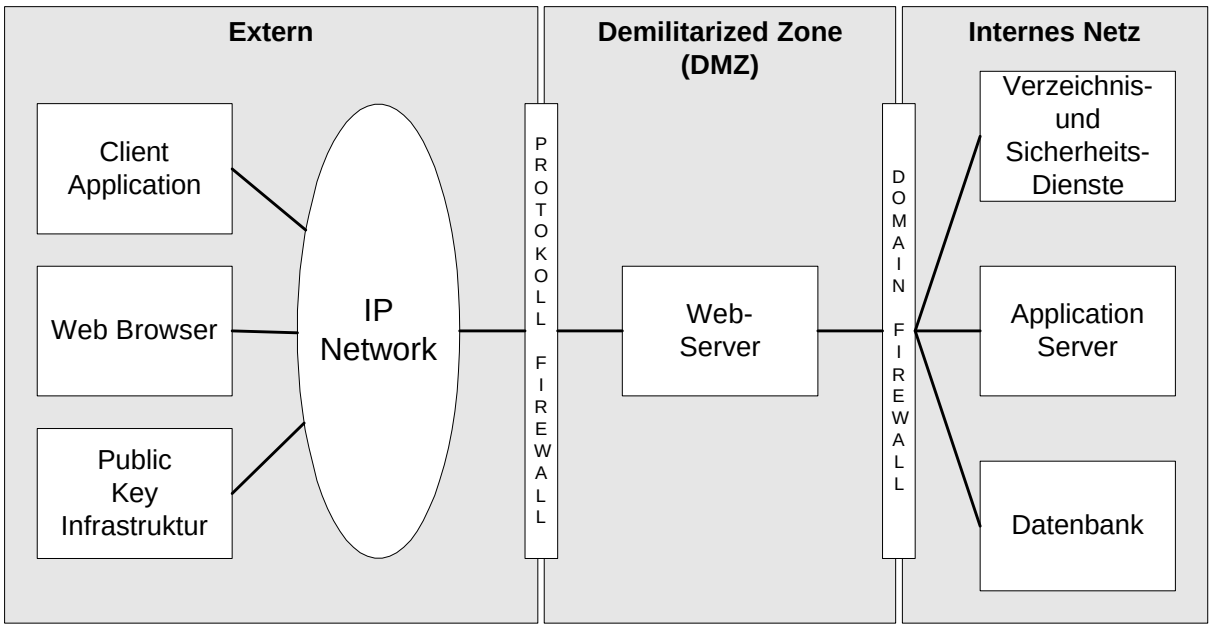

Abbildung 58: Firewall-Topologie

Der Schutz, der durch diese Firewall-Topologie erreicht wird, ist nicht ausreichend. Die Protokoll-Firewall lässt i.d.R. nur noch HTTP, HTTPS (SSL) und IIOP Ports offen, über die Webservices und Web-Applikationen, textbasierte Dokumente (z.B. HTML oder XML) austauschen. Im Falle der Webservices werden über SOAP[1] XML-Dokumente ausgetauscht, die Methodenaufrufe an server- oder clientseitigen Objekten darstellen. Auch Servlets und JSPs können, mit entsprechenden Parametern, gezielt gerufen werden. Das dynamische Verhalten von Applikationen wird mit statischen Dokumenten (Anfragen) beeinflusst.

Die Steuerung von Applikationen und der Informationsaustausch mit XML-Dokumenten nimmt stark zu. Es bestehen zwar Softwarelösungen zur Filterung von maliziösen XML-Dokumenten, so genannte XML-Proxies, diese sind aber nicht performant genug, um große Datenmengen effektiv zu filtern. Hardwareversionen sind zur Zeit noch nicht am Markt und intelligente Filterregeln befinden sich noch in der Entwicklung.

Die Sicherheit in Applikationen wird oft vernachlässigt. Man verlässt sich auf eine bestehende Sicherheitsinfrastruktur, die auf klassischer Betriebssystems-, Netzwerk- und Datenbanksicherheit basiert. Geeignete Sicherheitsmechanismen und Architekturen müssen beim Design der Applikationen und der Applikationsserver berücksichtigt werden. Ein neues Gebiet der IT-Sicherheit ist die *Web Application Security*. Projekte, wie das „Open Web Application Security Project“ [OWASP], leisten hier Pionierarbeit.

Neben XML-Attacken existieren weitere mögliche Angriffsformen:

- Parameter Injection (Cross-Site Scripting [CSS], SQL Injection [SQLINJ], Reverse Directory Traversal (../../password.dat), Buffer Overflow [Bar-Gad], Hidden Field Manipulation [Bar-Gad])
- Cookie Manipulation (Session-Hijacking)
- Vulnerability Scanning (Scannen nach bekannten Fehlern, Angriff durch Exploit, wie zum Beispiel: RDS Exploit, Code Red Worm Exploit, Nimda Virus)
- Extension Checking (einige Skripte und das *FileServingServlet* lassen sich ausnutzen)
- Diverse systematische Suchverfahren nach Dateien oder Verzeichnissen.

Noch gibt es sehr wenige Produkte, die diese Angriffe erkennen und abwehren können. Einen ersten Ansatz bietet [CodeSeeker] von OWASP. CodeSeeker ist eine Firewall auf Applikationsebene und Intrusion Detection System (IDS), die HTTP Netzwerkverkehr regelbasiert nach maliziösen Code untersucht. Ein weiteres Produkt ist [Web-Inspect] von SPI Dynamics. Web-Inspect ist ein Assessment Tool, mit dem sich die Sicherheit von Web-Applikationen überprüfen lässt.

Letztendlich steht und fällt die Sicherheit von Web-Applikationen mit dem „sicheren“ Design der Applikation, der Sicherheit des Web- und Applikationsservers und den, auf dem Application Gateway eingesetzten, Filterregeln. Die Performanz und Skalierbarkeit des Application Gateways und ist sehr wichtig. Denn eine Überlastung des Gateways würde eine Denial of Service (DoS) Attacke darstellen.

[1] Simple Object Access Protocol

6.10 WebSphere Vulnerabilities

WebSphere wird immer im Verbund mit anderen Serverprodukten eingesetzt, wie diverse Webserver, Verzeichnisdienste und Datenbanken. Eine Schwachstelle in einem dieser Produkte kann die Sicherheit des Gesamtsystems beeinflussen. Die vielen möglichen Produkte und ihre Schwachstellen können, aufgrund des Umfanges, in dieser Arbeit nicht dargestellt werden. Deshalb beschränkt sich dieses Kapitel auf die Darstellung der Schwachstellen in WebSphere.

Von WebSphere Advanced Server v4.0.3 ist bisher nur eine Vulnerability (Schwachstelle) bekannt: „IBM WebSphere Large HTTP Header Buffer Overflow“ [SecurityFocus bid5749], [BUGTRAQ2002035].

Das WebSphere Plug-In limitiert nicht die Größe der HTTP POST Daten, die zum Applikationsserver gesendet werden.

In [BUGTRAQ2002035] wird berichtet, dass eine Anfrage an eine JSP-Ressource (unabhängig davon, ob diese Ressource existiert), unter den folgenden Bedingungen, einen Buffer Overflow auslösen kann:

- Das HOST Feld enthält mehr als 796 Zeichen oder mehr.
- Ein beliebiges HTTP Feld enthält 4K Zeichen oder mehr.

Zum Teil wurde eine automatische Wiederherstellung des Dienstes beobachtet.

IBM beschreibt den Buffer Overflow unter [PQ62144, 4.0.3]. Das WebSphere Plug-In limitiert nicht die Größe der HTTP POST Daten, die zum Applikationsserver gesendet werden. Wenn der Webserver die Daten nicht begrenzt, sendet das Plug-In die Daten an den Applikationsserver. Dort tritt dann ein Fehler, unabhängig vom Namen (HTTP1.x) des Headers, auf. Mit einem E-Fix[1] bietet IBM eine Lösung des Problems für AIX, HP-UX, Linux, Solaris und Windows. Mit diesem E-Fix lässt sich die Menge der POST Daten spezifizieren. Der Standardwert beträgt 10 MB.

Die Frage ist, ob der Buffer Overflow im Webserver oder im Application Server auftritt. In der Dokumentation (Readme-Datei) des Fixes wird vom Webserver gesprochen und auf den Web-Seiten vom Application Server. Ich vermute, es ist ein Fehler in der Readme-Datei.

Mit diesem Buffer Overflow kann der Angreifer eine Denial of Service Attacke durchführen oder beliebigen Code auf einem Server ausführen.

[1] Ein E-Fix ist ein kumulativer Patch (Fix) von IBM, der einzelne Fehler behebt.

7 Differentialanalyse

In dieser Differentialanalyse sollen mögliche Unterschiede zwischen der J2EE-Spezifikation und der J2EE-Implementation in WebSphere aufgezeigt werden. Diese Untersuchung wurde rein konzeptionell, anhand der Spezifikationen von SUN und den Dokumentationen von IBM, durchgeführt. Aufgrund der Komplexität von WebSphere erhebt diese Untersuchung nicht den Anspruch auf Vollständigkeit.

Die J2EE1.2-Kompatibilität von WebSphere wurde von SUN und IBM bereits untersucht und bestätigt [J2EECOMP]. Die J2EE-Kompatibilität wurde anhand der J2EE Compatibility Test Suite (CTS) getestet, die SUN nur lizenzierten Kunden zur Verfügung stellt. Laut IBM [WASOSS] hat der WebSphere Advanced Server v4.0 AE alle Tests der CTS bestanden.

Im Rahmen meiner Untersuchung überprüfte ich zunächst das Vorhandensein der geforderten Java und XML APIs. Dazu verglich ich die Anforderungen der J2EE1.2 und J2EE1.3 Spezifikationen mit den Angaben in der WebSphere Dokumentation. Ich stellte fest, dass WebSphere alle geforderten APIs, zum Teil auch in höheren Versionen, besitzt und weitere APIs bietet, die sich an der J2EE1.3-Spezifikation orientieren. Das Ergebnis ist in der folgenden Tabelle 1 zusammengefasst.

	J2EE 1.2	WAS 4.0 AE	J2EE 1.3
J2SE	1.2	1.3	1.3
JavaIDL	in J2SE	in J2SE	in J2SE
JDBC Core API	in J2SE	in J2SE	in J2SE
Servlet	2.2	2.2	2.3
JSP	1.1	1.1	1.2
EJB	1.1	1.1	2.0
JDBC	2.0	2.0	2.0
JTA/JTS	1.0	1.1	1.0
JNDI	1.2	1.2.1	in J2SE
JAF	1.0	1.0	1.0
RMI-IIOP	1.0	1.0	in J2SE
JMS	1.0	1.0.1	1.0
Java-Mail	1.1	1.1	1.2
JAF		1.0.1	1.0
JAXP			1.1
Connector API			1.0
JAAS			1.0
XML4J		3.1.1	
XSL		2.0	

Tabelle 1: Geforderte Java APIs

Die J2EE-Spezifikationen referenzieren alle Spezifikationen der oben genannten APIs. In den einzelnen Spezifikationen werden die Anforderungen, zum Teil unscharf, definiert. Leider existiert keine Checkliste, anhand der sich eine J2EE-Konformität prüfen ließe. Ich habe dennoch versucht, eine solche Liste mit geforderten Funktionalitäten, aus den J2EE1.2 und J2EE1.3 Spezifikationen, zu erstellen. Anschließend prüfte ich informell, ob diese Funktionen in der WebSphere Dokumentation zu finden sind.

Die Ergebnisse wurden in die Tabelle 2 nach folgendem Schlüssel eingetragen:

0 : nicht existent.

1 : vorhanden und die Beschreibung entspricht der Spezifikation.

2 : in ähnlicher Form vorhanden und die Beschreibung ähnelt der Spezifikation.

Anforderung	**J2EE 1.2**		**WAS4.0AE**		**J2EE 1.3**	
	Web	EJB	Web	EJB	Web	EJB
Container: Applet Container	1	0	1	0	1	0
Container: Application Client Container	0	1	0	1	0	1
Container: EJB Container	0	1	0	1	0	1
Container: Web Container	1	0	1	0	1	0
Deployment: Deployment Tool	1	1	1	1	1	1
Deployment: Role Mapping	1	1	1	1	1	1
EJB: Aktivierung/Passivierung von Session Beans	0	1	0	1	0	1
EJB: Common Secure Interoperability (CSIv2)	0	0	0	0	0	1
EJB: CORBA/IIOP Protokoll	0	1	0	1	0	1
EJB: EJBQL	0	0	0	0	0	1
EJB: Entity Bean BMP	0	1	0	1	0	1
EJB: Entity Bean CMP	0	1	0	1	0	1
EJB: HTTP Login Gateway	0	0	0	0	0	1
EJB: Local Interfaces	0	0	0	0	0	1
EJB: Message Driven Bean (MDB)	0	0	0	2	0	1
EJB: Method-Permission-Modell für EJBs	0	1	0	1	0	1
EJB: Persistenzmechanismus	0	1	0	1	0	1
EJB: Relationships (CMR)	0	0	0	2	0	1
EJB: Secure Association Service (SAS)	0	0	0	1	0	0
EJB: Security Attribute Service (SAS) Protokoll	0	0	0	0	0	1
EJB: Stateful Session Bean	0	1	0	1	0	1
EJB: Stateless Session Bean	0	1	0	1	0	1
Format: class Dateiformat	1	1	1	1	1	1
Format: EAR Archivformat	1	1	1	1	1	1
Format: ejb-jar Archivformat	1	1	1	1	1	1
Format: GIF Bildformat	1	1	1	1	1	1
Format: HTML 3.2	1	1	1	1	1	1
Format: JAR Archivformat	1	1	1	1	1	1
Format: JPEG Bildformat	1	1	1	1	1	1
Format: WAR Archivformat	1	0	1	1	1	1
J2EE Server Ausfallsicherheit	1	1	1	1	1	1
J2EE Server Clustering und Lastverteilung	1	1	1	1	1	1
J2EE Server Thread- und Prozessmanagement	1	1	1	1	1	1
J2EE Server: Datenbankanbindung	1	1	1	1	1	1
Protokoll: COSNaming	0	0	0	0	0	1
Protokoll: EJB Interoperability Protocol	0	0	0	0	0	1
Protokoll: HTTP 1.0	1	1	1	1	1	1
Protokoll: HTTPS	1	1	1	1	1	1
Protokoll: IIOP-RMI	1	1	1	1	1	1
Protokoll: JRMP-RMI	1	1	1	1	1	1
Protokoll: SSL3.0 / TLS1.0	1	1	1	1	1	1

Protokoll: TCP/IP UDP/IP	1	1	1	1	1	1
Security: Benutzerauthentisierung mit Applicationclients	1	1	1	1	1	1
Security: Benutzerauthentisierung mit Web-Clients	1	0	1	0	1	0
Security: Configured Identity (Run As)	0	0	0	1	1	1
Security: Deklarative Sicherheit	1	1	1	1	1	1
Security: J2SE-Sicherheitsmodell für Programmcode	1	1	1	1	1	1
Security: Lazy Authentication (Verzögerte Authentisierung)	1	0	1	0	1	0
Security: Login Sessions / Web Single Sign-On	1	0	1	0	1	0
Security: Namens- und Verzeichnisdienst	1	1	1	1	1	1
Security: Principal Delegation	0	1	0	1	1	1
Security: Programmatic Authentication	1	1	1	1	1	1
Security: Programmatic Security	1	1	1	1	1	1
Security: Propagierung von Identitäten	1	1	1	1	1	1
Security: Run-As Identitäten	0	0	0	1	1	1
Security: Sessionmanagement	1	1	1	1	1	1
Security: Single SignOn	1	0	1	0	1	0
Security: Transaktionsmanagement	1	1	1	1	1	1
Web Login: form-based login	1	0	1	0	1	0
Web Login: HTTP basic authentication (over SSL)	1	0	1	0	1	0
Web Login: HTTPS client authentication	1	0	1	0	1	1
Web Login: SSL mutual authentication	1	0	1	0	1	1
Web: Anonymer Zugriff auf Web-Komponenten	1	0	1	0	1	0
Web: Filter	0	0	2	0	1	0
Web: JSPs	1	0	1	0	1	0
Web: Security Contraints für Web-Ressourcen	1	0	1	0	1	0
Web: Servlets	1	0	1	0	1	0
Web: Web Event Listener	0	0	2	0	1	0

Tabelle 2: Geforderte Funktionen

Auch hier zeigte sich, dass WebSphere die J2EE1.2 Spezifikation vollständig implementiert und erweitert.

Im nächsten Schritt überprüfte ich, ob IBM, als J2EE-Product Provider, die J2EE-Contracts erfüllt. Das Ergebnis ist in Tabelle 3 zu sehen. IBM erfüllt alle J2EE-Verträge.

Contract	**erfüllt**
J2EE APIs	ja
J2EE Service Provider Interfaces (SPIs) und Connector API	ja
Netzwerkprotokolle: HTTP und HTTPS für Servlets und JSPs	ja
Netzwerkprotokolle: IIOP für EJBs	ja
Deployment Descriptoren und Deployment Tools	ja

Tabelle 3: J2EE-Verträge für den J2EE-Product Provider

IBM entspricht, neben der Rolle J2EE Product Provider, noch anderen J2EE-Roles. Die einzelnen Anforderungen an diese J2EE-Rollen wurden von mir als nächstes untersucht. Das Ergebnis ist in Tabelle 4 dargestellt. IBM erfüllt die Anforderungen.

Plattform Rolle	geforderte Funktionalität	vorhanden
J2EE Product Provider	J2EE APIs sind den Applikationskomponenten durch den Container verfügbar	ja
	Mapping der Applikationskomponenten zu den Netzwerkprotokollen	ja
	Tools für das Deployment und Management der Applikationen	ja
J2EE Server Provider	keine	ja
EJB-Container Provider	Laufzeitumgebung für EJBs	ja
	Transaktionsmanagement	ja
	Securitymanagement	ja
	skalierbares Management von Ressourcen	ja
	Verwaltung von verteilen Remote-Clients	ja
Servlet Container Provider	Laufzeitumgebung Servlets	ja
	Securitymanagement	ja
	skalierbares Management von Ressourcen	ja

Tabelle 4: Plattform Rollen von IBM

Das Ergebnis meiner Differentialanalyse lautet: IBM Websphere Advanced Server v4.0 AE erfüllt die Anforderungen der J2EE1.2-Spezifikation und orientiert sich an der J2EE1.3 Spezifikation. Dieses Ergebnis ist nicht überraschend, denn SUN hat die J2EE-Konformität bereits bestätigt. Anhand der verfügbaren Informationen ist eine genauere Analyse nicht möglich. Viele Bestandteile von WebSphere sind proprietär (zum Beispiel das LTPAToken Format oder SAS). Präzise Beschreibungen des internen Aufbaus von WebSphere sind nur IBM Business Partnern, die eine Verschwiegenheitserklärung unterzeichnen müssen, zugänglich. Für einen vollständigen J2EE-Kompatibilitätstest ist die J2EE Compatibility Test Suite (CTS) von SUN unverzichtbar. Es stellt sich die Frage, warum SUN CTS nicht öffentlich zum Download anbietet. Diese Frage stellte ich SUN in einem Email. Leider habe ich noch keine Antwort erhalten. Die Qualität der Tests würde sicher steigen, wenn eine dritte neutrale Instanz diese Tests durchführt.

8 Ausblick

In den ersten Kapiteln wurde die J2EE1.3 Plattform mit ihren Komponenten vorgestellt. Sie bildet die Grundlage für viele Applikationsserver. Sie basieren neben Java hauptsächlich auf CORBA. Eine eingehendere Analyse der Sicherheit von CORBA könnte Gegenstand einer weiteren Untersuchung sein.

Das J2SE-Sicherheitsmodell wurde im darauf folgenden Kapitel dargestellt. Es zeigte sich, dass es viele Möglichkeiten bietet aber auch eine hohe Komplexität besitzt, die Entwickler überfordern könnte. Die Überforderung könnte zu einer Vermeidungsstrategie führen, die ein sicheres Design von Applikationen unmöglich macht.

Anschließend wurden die Sicherheitskonzepte der J2EE-Plattform beschrieben. Der Schwerpunkt lag hier bei der rollenbasierten Autorisierung und Authentisierung. Authentisierungsmechanismen sind in großen Organisationen von hoher Bedeutung. Insbesondere sind hier Verfahren zum Single Sign-On, wie Microsoft Passport oder Liberty Alliance, und Kerberos zu nennen. Diese konnten in dieser Arbeit nicht vollständig untersucht werden und könnten in zukünftigen Arbeiten analysiert werden.

In der zweiten Hälfte dieser Arbeit wurde der WebSphere Application Server v4.0 AE vorgestellt, der die J2EE1.2 Spezifikation implementiert. Mittlerweile ist WebSphere v5.0 erhältlich, der die in dieser Arbeit vorgestellte J2EE1.3 Spezifikation vollständig implementiert. Für eine Analyse von WebSphere v5 könnte diese Arbeit eine Grundlage bilden. Eine offizielle Kooperation des Fachbereichs mit IBM würde diese Untersuchung wesentlich erleichtern. Neben WebSphere sollten auch Applikationsserver, wie JBoss oder BEA WebLogic untersucht werden. Gerade das Open Source Projekt JBoss scheint mit höchst interessant zu sein.

Im Kapitel „WebSphere Firewall-Topologien“ wurde aufgezeigt, dass herkömmliche Firewalls zum Schutz von Applikationsservern nicht ausreichend sind. Mit dem Schutz von Web- und Applikationsservern beschäftigt sich die Web Application Security. Ich empfehle dieses Gebiet im Fachbereich Informatik zu etablieren und in den Lehrplan aufzunehmen. Speziell die Sicherheit von XML-Dokumenten und Webservices scheint mir enorm wichtig, da sie vermehrt eingesetzt werden. Sicherheit auf Applikationsebene und moderne Filtertechnologien bildet hierfür die Grundlage.

Zusammengefasst könnten die folgenden Themen Gegenstand weiterer Untersuchungen sein:

- Web Application Security
- XML-Security
- Webservices-Security
- CORBA Security Services
- Single Sign-On Verfahren, wie Liberty Alliance oder Microsoft Passport
- Applikationsserver, wie WebSphere v5.0, JBoss, BEA WebLogic

9 Abbildungsverzeichnis

10 Literaturverzeichnis

[Bellovin 89] S.M. Belloin, “Security Problems in the TCP/IP Protocol Suite”, Computer Communication Review, Vol. 19, No. 2, pp. 32-48, April 1989

[Berg 00] Clifford J. Berg, "Advanced Java 2 development for enterprise applications", SUN Microsystems Press/Prentice Hall PTR 2000

[Denninger 00] S. Denninger, I. Peters, “Enterprise JavaBeans” Addison-Wesley 2000

[Hoque 00] Faisal Hoque, "E-enterprise", Cambridge Univ. Press 2000

[Huegen 98] Craig A. Huegen, “The Latest in Denail of Service Attacks: “Smurfing”, Description and nformation to Minimize Effects”, April 1989

[Kerner 92] Helmut Kerner, „Rechnernetze nach OSI“, Addison-Wesley 1992

[Kyas 98] Othmar Kyas, "Sicherheit im Internet", Internat. Thomson Publ. 1998

[Li Gong 99] Li Gong, "Inside java 2 platform security", Addison-Wesley 2000

[Lienemann 00] Gerhard Lienemann, "TCP/IP-Grundlagen", Heise 2000

[Luerssen 01] André Luerssen, Studienarbeit : „Netzwerkprogrammierung mit Java2: Risiken und Lösungsansätze“

[Mahmoud 00] Qusay H. Mahmoud, "Distributed programming with Java", Manning 2000

[McGraw 99] Gary McGraw, Edward W. Felten, "Securing Java", Wiley 1999

[Middendorf 99] Stefan Middendorf, Reiner Singer, “ Java Programmierhandbuch und Referenz füt die Java-2-Plattform”, dpunkt 1999

[Monson-Haefel 99] Richard Monson-Haefel, "Enterprise JavaBeans", O'Reilly 1999

[Oaks 98] Scott Oaks, "Java security", OReilly 1998

[Orfali 98] Robert Orfali, Dan Harkey, "Client/Server programming witsch Java and CORBA", Wiley 1998

[Seshadri 99] Govind Seshadri, Gopalan Suresh Raj, "Enterprise Java computing" Cambridge univ. Press 1999

[Slama 99] Dirk Slama, Jason Garbis, Perry Russel, "Enterprise CORBA", Prentice hall 1999

[Smith 98] Richard E. Smith, "Internet-Kryptographie", Addison-Wesley Longmann 1998

[Sohr 00] Karsten Sohr, “Sandkastenspiele”, ct-Magazin 11/2000 S. 226

[SS95] V. Samar and R. Schemers, “Unified Login with Pluggable Authentication Modules (PAM),” Request For Comments: 86.0, Open Software Foundation (October 1995).

[Venners 99] Venners, B.(1999) Inside the Java 2 Virtual Machine. McGraw-Hill, Inc.

[Vitek 99] Jan Vitek, "Secure Internet programming", Springer 1999

[Vogel 98] Andreas Vogel, Keith Duddy, "Java programming with CORBA", Wiley 1998

11 Internetquellen

[Bar-Gad] Auditing and Securing Web-enabled Applications, http://www.theiia.org/itaudit/index.cfm?fuseaction=forum&fid=435

[BobbyRite] Java / MSIE / Netscape Cache Exploit - Jan '97, BobbyRite, http://www.alcrypto.co.uk/java/

[Bugtraq] Bugtraq archives for 4th quarter (Oct-Dec) 1997: Another way to exploit local classes in Java, http://www.dataguard.no/bugtraq/1997_4/0055.html

[BUGTRAQ2002035] IBM Websphere Large Header DoS, http://online.securityfocus.com/archive/1/292458

[CERT] Inet CERT Coordination Center, http://www.cert.org/

[CERTPATH] Java Certification Path API, http://java.sun.com/j2se/1.4/docs/api/java/security/cert/package-summary.html

[CheckPoint] Check Point Software Technologies, http://www.checkpoint.com/

[CodeSeeker] CodeSeeker Project, http://www.owasp.org/codeseeker/

[CONS10] J2EE Connector Architecture 1.0, http://java.sun.com/j2ee/connector

[CORBA231] The Common Object Request Broker: Architecture and Specification (CORBA 2.3.1 specification), http://cgi.omg.org/cgi-bin/doc?formal/99-10-07.

[CORBA30] http://www.omg.org/technology/documents/vault.htm#CORBA_IIOP

[Counterpane] Counterpane Internet Security, Inc., http://www.counterpane.com/

[Covalent] Covalent Technologies, http://www.covalent.net/raven/ssl/index.php

[CrackingDES] Cracking DES: Secrets of Encryption Research, Wiretap Politics, and Chip Design, http://cryptome.org/cracking-des.htm

[CSI] Computer Security Information, http://www.alw.nih.gov/Security/

[CSS] CERT Cross Site Scripting Vulnerabilities,
http://www.cert.org/archive/pdf/cross_site_scripting.pdf

[DEAJ2EE] Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition, Second Edition,
http://java.SUN.com/blueprints/guidelines/designing_enterprise_applications_2e/

[DES Cracker] EFF DES Cracker Project , http://www.eff.org/descracker.html

[DFN-CERT] DFN-CERT GmbH, DFN-PCA und DFN-FWL, http://www.cert.dfn.de/

[DOPRIVILEGED] API for Privileged Blocks,
http://java.sun.com/j2se/1.4/docs/guide/security/doprivileged.html

[Dormanns02] E-Business Systemarchitekturen
http://kbs.cs.tu-berlin.de/teaching/ws2001/ebusiness/ folien1fach/008_Sichere_Arch.pdf

[EJBS20] Enterprise JavaBeans Specification, Version 2.0, http://java.sun.com/products/ejb

[Freeswan] FreeS/WAN Project: Home Page, http://www.xs4all.nl/~freeswan/

[Fritzinger] Java Security, whitepaper by J. Steven Fritzinger, Marianne Mueller,
http://java.sun.com/security/whitepaper.ps

[HigherLearning] : For all the real hackers & phreakers, http://www.xs4all.nl/~l0rd/

[Hopwood] David Hopwood, http://www.users.zetnet.co.uk/hopwood/,

[IBM alphaWorks] IBM alphaWorks, http://www.alphaworks.ibm.com/Home/

[IBMDevCon] IBM Developer Connection,
http://www.developer.ibm.com/devcon/titlepg.htm

[IBMDeveloperWorks] IBM developerWorks : Java technology,
http://www.ibm.com/developer/java/

[IBMVADD] VisualAge Developer Domain, http://www7.software.ibm.com/vad.nsf/

[IDL2JS] IDL To Java Language Mapping Specification, Object Management Group,
http://cgi.omg.org/cgi-bin/doc?ptc/2000-01-08.

[IETF] IETF Home Page, http://www.ietf.org/

[IONSS] Interoperable Naming Service, Object Management Group,
http://cgi.omg.org/cgi-bin/doc?ptc/00-08-07 und
http://cgi.omg.org/cgi-bin/doc?formal/2000-06-19

[iPlanet] iPlanet Developer, http://developer.iplanet.com/index.html

[J2DOC] Java 2 SDK Documentation, http://java.sun.com/products/jdk/1.2/docs/index.html

[J2EECOMP] Java 2 Platform, Enterprise Edition - J2EE 1.2 Compatibility,
http://java.sun.com/j2ee/1.2_compatibility.html

[J2EES13] Java 2 Platform, Enterprise Edition Specification Version 1.3,
http://java.sun.com/j2ee/docs.html.

[J2EETO] Java 2 Platform, Enterprise Edition Technical Overview (J2EE Overview),
http://java.sun.com/j2ee/white.html.

[J2IDLS] Java Language To IDL Mapping Specification, Object Management Group,
http://cgi.omg.org/cgi-bin/doc?ptc/2000-01-06.

[J2SEC] SUN Java Security Architecture: Contents,
http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html

[J2SES13] Java 2 Platform, Standard Edition, v1.3 API Specification,
http://java.sun.com/j2se/1.3/docs/api/index.html.

[J2SES14] Java 2 Platform, Standard Edition, v1.4 API Specification,
http://java.sun.com/j2se/1.3/docs/api/index.html.

[JAASS10] Java Authentication and Authorization Service (JAAS) 1.0,
http://java.sun.com/products/jaas

[JAFS10] JavaBeans Activation Framework Specification Version 1.0,
http://java.sun.com/beans/glasgow/jaf.html

[JARS] JAR-Specification http://java.sun.com/j2se/1.3/docs/guide/jar

[JAXPS10] Java API for XML Parsing, Version 1.0 Final Release (JAXP specification).
Copyright 1999-200, SUN Microsystems, Inc. Available at http://java.sun.com/xml

[JCES122] Java Cryptography Extension 1.2.2 API Specification & Reference
http://java.sun.com/products/jce/doc/guide/API_users_guide.html

[JCP85] Java Community Process „JCP-85 Rules-based Authorization and Audit”
http://www.jcp.org/en/jsr/results?j=85&t=1&c=1

[JDBCS21] JDBC 2.1 API, http://java.sun.com/products/jdbc

[JDBCSES20] JDBC™ 2.0 Standard Extension API, http://java.sun.com/products/jdbc

[JDKS11] SUN JDK 1.1.x Documentation,
http://java.sun.com/products/jdk/1.1/docs/index.html

[JGURU] jGuru.com(Home): Your view of the Java universe,
http://www.jguru.com/portal/index.jsp?tab=1

[JMAILS11] JavaMail API Specification Version 1.1, http://java.sun.com/products/javamail

[JMSS102] Java Message Service, Version 1.0.2, http://java.sun.com/products/jms

[JMXS11] Java Management Extensions (JMX) Specification 1.1,
http://java.sun.com/products/JavaManagement/

[JNDIS12] Java Naming and Directory Interface 1.2 Specification,
http://java.sun.com/products/jndi

[JSEC] Java Security API, http://java.sun.com/security/

[JSPS12] Java Server Pages Specification, Version 1.2, http://java.sun.com/products/jsp

[JSSE103] Java Secure Socket Extension (JSSE) 1.0.3_01
http://java.sun.com/products/jsse/index-103.html

[JSUN] SUN The Source for Java Technology, http://java.sun.com/

[JTAS101] Java Transaction API, Version 1.0.1, http://java.sun.com/products/jta

[JTSS10] Java Transaction Service, Version 1.0, http://java.sun.com/products/jts

[JVMS1] The Java Virtual Machine Specification,
http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html

[JVMS2] The Java Virtual Machine Specification, Second Edition,
http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

[KESINFO] KES online - Lexikon der IT-Sicherheit
http://www.kes.info/_lexikon/lexdata/authentifizierung.htm

[Kimera 00] Kimera: A Java System Architecture, http://kimera.cs.washington.edu/

[Lee 99] Berners-Lee: Weaving the Web,
http://www.w3.org/People/Berners-Lee/Weaving/Overview.html

[Luerssen 01] Studienarbeit André Luerssen “Netzwerkprogrammierung mit Java2: Risiken und Lösungsansätze”,
http://agn-www.informatik.uni-hamburg.de/papers/doc/studarb_andre_luerssen.pdf

[Mahmoud] Distributed Programming with Java,
http://www.manning.com/Mahmoud/index.html

[MattBlaze] Matt Blaze's cryptography resource on the web, http://www.crypto.com/

[McQueen 00] Miles McQueen Java Virtual Machine Security and the Brown Orifice Attack,
http://www.sans.org/infosecFAQ/java_sec.htm

[mladue 00] Hostile Applets Home Page, http://metro.to/mladue/hostile-applets/index.html

[Nachenberg 99] Carey Nachenberg, JavaApp.BeanHive,
http://www.symantec.com/avcenter/venc/data/javaapp.beanhive.html

[Nachenberg 98] Carey Nachenberg, Eric Chien, Stephen Trilling, JavaApp.Strange Brew,
http://www.symantec.com/avcenter/venc/data/javaapp.strangebrew.html

[OWASP] Open Web Application Security Project, http://www.owasp.org/

[PERMISSIONS] Permissions in the Java(TM) 2 SDK
http://java.sun.com/j2se/1.3/docs/guide/security/permissions.html

[PoisonJava] Poison Java, Richard Comerford,
http://csl.iit.edu/~java/papers_others/java_security/jav.html

[PQ62144, 4.0.3] IBM, Possible security exposure with web servers plugin
http://www-1.ibm.com/support/docview.wss?rs=180&context=SSEQTP&q=PQ62144&uid=swg24001610

[RACF] IBM Resource Access Control Facility (RACF),
http://www-1.ibm.com/servers/eserver/zseries/zos/racf/racfhp.html

[RFC768] Jon Postel, User Datagram Protocol

[RFC791] Jon Postel, Internet Protocol - DARPA Internet Program Protocol Specification

[RFC792] Jon Postel, Internet Control Message Protocol

[RFC793] Jon Postel, Transmission Control Protocol - DARPA Internet Program Protocol Spezification, September 1981

[RFC1034] P. Mockapetris, Domain Names – Concepts and Facilities

[RFC1035] P. Mockapetris, Domain Names – Implementation and Specification

[RFC1630] RFC 1630 Uniform Resource Identifiers (URI)

[RFC1738] RFC 1738 Uniform Resource Locators (URL)

[RFC1808] RFC 1808 Relative Uniform Resource Locators

[RFC1883] S. Deering, Xerox PARC, Internet Protocol, Version 6 (IPv6) Specification, December 1995

[RFC1884] S. Deering, Xerox PARC, IP Version 6 Addressing Architecture, December 1995

[RFC1945] RFC 1945 Hypertext Transfer Protocol (HTTP/1.0)

[RFC2045] RFC 2045 MIME Part One: Format of Internet Message Bodies

[RFC2046] RFC 2046 MIME Part Two: Media Types

[RFC2047] RFC 2047 MIME Part Three: Message Header Extensions for non-ASCII text

[RFC2048] RFC 2048 MIME Part Four: Registration Procedures

[RFC2049] RFC 2049 MIME Part Five: Conformance Criteria and Examples

[RFC2109] RFC 2109 HTTP State Management Mechanism

[RFC2145] RFC 2145 Use and Interpretation of HTTP Version Numbers

[RFC2324] RFC 2324 Hypertext Coffee Pot Control Protocol (HTCPCP/1.0)

[RFC2396] RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax

[RFC2616] RFC 2616 Hypertext Transfer Protocol (HTTP/1.1)

[RFC2617] RFC 2617 HTTP Authentication: Basic and Digest Authentication

[RFC2743] RFC 2743 Generic Security Service Application Program Interface v.2 upd.1

[RFC2853] RFC 2853 Generic Security Service API Version 2 : Java Bindings

[RMIS13] Java Remote Method Invocation Specification 1.3, http://java.sun.com/j2se/1.3/docs/guide/rmi

[RSA] RSA Security Inc., http://www.rsasecurity.com/

[SALTZER75] J.H. Saltzer and M.D. Schroeder. „The Protection of Information in Computer Systems". Proceedings of the IEEE, 63(9):1278--1308, September 1975 - http://web.mit.edu/Saltzer/www/publications/protection/index.html

[SecurityFocus bid 1121] MS IE 5.01 JSObject Cross-Frame Vulnerability, http://www.securityfocus.com/bid/1121

[SecurityFocus bid 1209] Microsoft Internet Explorer for Macintosh java.net.URLConnection Vulnerability, http://www.securityfocus.com/bid/1209

[SecurityFocus bid 1339] Microsoft Internet Explorer for Macintosh getImage and Class Loader Vulnerabilities, http://www.securityfocus.com/bid/1339

[SecurityFocus bid 957] ,Microsoft Java Virtual Machine getSystemResource Vulnerability, http://www.securityfocus.com/bid/957

[SecurityFocus bid1336] Multiple Vendors java.net.URLConnection Applet Direct Connection Vulnerability, http://www.securityfocus.com/bid/1336

[SecurityFocus bid1337] Multiple Vendors HTTP Redirect Java Applet Vulnerability, http://www.securityfocus.com/bid/1337

[SecurityFocus bid1545] Multiple Vendor Java Virtual Machine Listening Socket Vulnerability, http://www.securityfocus.com/bid/1545

[SecurityFocus bid1546] Netscape Communicator URL Read Vulnerability, http://www.securityfocus.com/bid/1546

[SecurityFocus bid1754] Microsoft Virtual Machine com.ms.activeX.ActiveXComponent Arbitrary Program Execution Vulnerability, http://www.securityfocus.com/bid/1754

[SecurityFocus bid1812] Microsoft Virtual Machine Arbitrary Java Codebase Execution Vulnerability, http://www.securityfocus.com/bid/1812

[SecurityFocus bid2051] JRE Disallowed Class Loading Vulnerability, http://www.securityfocus.com/bid/2051

[SecurityFocus bid5749] IBM WebSphere Large HTTP Header Buffer Overflow Vulnerability, http://online.securityfocus.com/bid/5749

[SERVLETS23] Java Servlet Specification, http://java.sun.com/products/servlet.

[SG245755] IBM Redbook “Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java”, http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245755.pdf

[SG246176] IBM Redbook “IBM WebSphere V4.0 Advanced Edition Handbook”, http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg246176.pdf

[SG246520] IBM Redbook “IBM WebSphere V4.0 Advanced Edition Security”, http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg246520.pdf

[SIPL] Secure Internet Programming Laboratory, http://www.cs.princeton.edu/sip/

[SQLINJ] NGSSoftware Insight Security Research (NISR), Advanced SQL Injection In SQL Server Applications, http://www.nextgenss.com/papers/advanced_sql_injection.pdf

[SSL30] The SSL Protocol, Version 3.0, http://home.netscape.com/eng/ssl3

[SUN Java History] JAVA(TM) TECHNOLOGY: THE EARLY YEARS, http://java.sun.com/features/1998/05/birthday.html

[SUN SecCron 00] Security FAQ - Java(TM) Software Technology, http://java.sun.com/sfaq/chronology.html

[SUNDevCon] SUN Java Developer Connection, http://developer.java.sun.com/

[SUNFAQ] SUN Frequently Asked Questions - Applet Security, http://java.sun.com/sfaq/

[W3C] W3C - The World Wide Web Consortium, http://www.w3.org/

[WAS40DOCS] IBM WebSphere Online Documentation, http://www-3.ibm.com/software/webservers/appserv/doc/v40/aee/

[WASOSS] WebSphere Open Standards Support, http://www7b.boulder.ibm.com/wsdd/products/supportj2ee.html

[WASSSL] IBM WebSphere Advanced & Advanced Single Server Editions 4.0--SSL/TLS in WebSphere: Usage, Configuration, and Performance, http://www-900.ibm.com/websphere/library/techarticles/webspheressl/webspheressl_gui_eng.shtml

[Web-Inspect] SPI Dynamics, http://www.spidynamics.com/product.html

[Woude] Rob van der Woude's Scripting Pages: Batch Files for DOS, Windows (all flavours) and OS/2; Rexx; JavaScript; HTML, http://home.wanadoo.nl/r.woude/robmain.html

[XMLPROXY] Zapthink, http://www.zapthink.com/news/pr07262002-xmlproxy.html