J2EE Hotspots

PROGRAMMER'S CHOICE

Die Wahl für professionelle Programmierer und Softwareentwickler. Anerkannte Experten wie z. B. Bjarne Stroustrup, der Erfinder von C++, liefern umfassendes Fachwissen zu allen wichtigen Programmiersprachen und den neuesten Technologien, aber auch Tipps aus der Praxis.
Die Reihe von Profis für Profis!

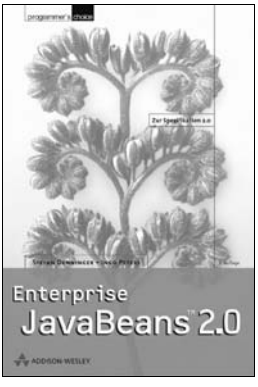
Hier eine Auswahl:

Entwurfsmuster
Erich Gamma et al.
484 Seiten
€ 49,95 [D], € 51,40 [A]
ISBN 3-8273-1862-9

Dieses Buch ist der Klassiker für Entwurfsmuster und objektorientierte Softwareentwicklung! Die Autoren formulieren 23 Entwurfsmuster, benennen und beschreiben sie und erläutern ihre Verwendung. Diese Entwurfsmuster bieten einfache und prägnante Lösungen für häufig auftretende Programmieraufgaben. Sie erlauben die Wiederverwendung bewährter Lösungsstrategien und ermöglichen die Verständigung über die eigene Arbeit. Übersetzung aus dem Amerikanischen von Dirk Riehle.

J2EE Patterns
Adam Bien
244 Seiten
€ 39,95 [D], € 41,10 [A]
ISBN 3-8273-1903-X

Bei dem Entwurf von J2EE-Anwendungen spielen die Erfahrungen des Entwicklers mit den einzelnen APIs eine wichtige Rolle. Besonders das Zusammenspiel der einzelnen Spezifikationen kann über Erfolg oder Scheitern eines J2EE-Projekts entscheiden.
Dieses Buch beschreibt die von Sun Java Center entwickelten Patterns anhand von praktischen Beispielen. Die unterschiedlichen Ansätze beim Umgang mit den J2EE 1.2 und 1.3 APIs werden ausführlich behandelt und die Einsatzmöglichkeiten jedes Patterns mithilfe gemessener Performanceunterschiede aufgezeigt.



Enterprise JavaBeans™ 2.0
Stefan Denninger, Ingo Peters
448 Seiten
€ 49,95 [D], € 51,40 [A]
ISBN 3-8273-1765-7

Erstellen Sie Java-Webservices mit Enterprise Java Beans 2.0! Die 2., aktualisierte und erweiterte Auflage des erfolgreichen Buchs erklärt ausführlich die Architektur der Enterprise JavaBeans und die konkrete Programmierung von Enterprise Beans. Alle wichtigen Bestandteile werden detailliert in eigenen Kapiteln behandelt: Entity-Beans, Session-Beans, Message-Driven-Beans, Transaktionen und das Thema Sicherheit. Viele praxisrelevante Beispiele erleichtern den Einstieg.

PEARSON Education

ADDISON-WESLEY

Adam Bien, Rainer Sawitzki

J2EE Hotspots

Professionelle Lösungen für die Java-Entwicklung

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Umwelthinweis:
Dieses Produkt wurde auf chlorfrei gebleichtem Papier gedruckt.

5 4 3 2 1

06 05 04 03

ISBN 3-8273-1950-1

ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany

Einbandgestaltung: Christine Rechl, München
Titelbild: Valeriana officinalis, Baldrian © Karl Blossfeldt Archiv
Ann und Jürgen Wilde, Zülpich/ VG Bild-Kunst Bonn, 2003
Lektorat: Martin Asbach, masbach@pearson.de
Korrektorat: Christine Depta, Unterschleißheim
Herstellung: Monika Weiher, mweiher@pearson.de
Satz: reemers publishing services gmbh, Krefeld, www.reemers.de
Druck und Verarbeitung: Bercker Graphischer Betrieb, Kevelaer
Printed in Germany

Inhalt

Vorwort I

»Es wurde bereits alles programmiert – aber noch nicht von jedem«

[Teilnehmer eines J2EE Workshops]

Beim Schreiben des Buches »J2EE Patterns« ist mir aufgefallen, dass noch sehr viele Fragen ungeklärt sind. Diese Vermutung hat sich in durchgeführten Schulungen und Beratungen bestätigt, da oft ähnliche oder sogar die gleichen Fragen über die J2EE-Technologie von den Teilnehmern gestellt wurden. Nach dieser Feststellung habe ich begonnen die Fragen aufzuschreiben, um sie auch schriftlich zu beantworten. Die Idee für dieses Buch und auch dieses Buch selbst sind auf diese Art und Weise entstanden. Die Wahl des Titels wurde allerdings zum Problem. Das Buch sollte »J2EE FAQ« heißen (versuchen Sie es mal auszusprechen). Mein Lektor, Herr Asbach, meinte auch berechtigterweise, dass dieser Titel falsches Zielpublikum ansprechen könnte (also Leser, die eher an Titeln wie »J2EE in 1,5 Tagen« interessiert sind ☺). Mein zweiter Vorschlag war dann »J2EE Hotspots I« – mit der Hoffnung, dass die restlichen Fragen im zweiten Teil beantwortet werden könnten.

Ich wurde nicht nur von Fragen der Teilnehmer sondern auch von einigen (suboptimalen) Projektvorgaben in J2EE Projekten inspiriert. Oft wird die J2EE-Technologie vor dem eigentlichen Projektbeginn mit immensem Aufwand evaluiert (EJB ja oder nein, brauchen wir Entity Beans?, welcher Applikationsserver soll eingesetzt werden?) – über die Architektur macht man sich hier weit weniger Gedanken.

Zu diesem Zeitpunkt hat auch Herr Dr. Sawitzki mein Buch »J2EE Patterns« evaluiert. Ich fragte Herrn Sawitzki, ob er an diesem Projekt mitarbeiten könnte. Für seine spontane Zustimmung und Unterstützung bei der Umsetzung dieses Buchs möchte ich mich an dieser Stelle bedanken.

Außerdem möchte ich mich bei meinem (noch) kleinen Bruder Michael Bien bedanken. Er hatte die Starviewer-Software (Visualisierung der Sterne) entworfen und implementiert. Auf seine etwas unkonventionelle Art und Weise hat er demonstriert, wie auch größere Datenmengen sich in JAVA visualisieren lassen. Michael ist 16 Jahre alt – es ist seine zweite Anwendung in JAVA. Seine erste Anwendung war ein Roboter (robocode.net), mit dem Michael den 3-ten Platz in der Beginner-Liga geschafft hat! Wenn Sie fragen zu Starviewer oder Robocode haben, können Sie Ihn direkt erreichen unter: *mbien@adam-bien.com*.

Meine Frau Kinga Bien hat sich durch die zahllosen Korrekturen, meine Erklärungen und die Schilderung meiner Ideen einen so umfangreiches Wortschatz an Fachwörtern aufbauen können, dass sie locker die meisten Teambesprechungen und presales-Gespräche moderieren kann. Ferner hat sie meine zahlreichen »Deadlocks« (also Momente, in denen ich an J2EE bzw. JAVA gedacht hatte und nicht ansprechbar war) stets toleriert und mich weiterhin unterstützt. Für die graphische Gestaltung meiner Homepage (*www.adam-bien.com*), der Homepage des Starfinders (*www.star-finder.com*) und die Aufbereitung der Graphiken, ihre Unterstützung und Geduld möchte ich mich an dieser Stelle besonders bedanken.

Meinen Eltern, die trotz meiner Standardantwort: »jetzt keine Zeit – das Buch muss fertig werden« mich stets unterstützt hatten. Ferner für das Verständnis für meinen Bruder, der Nächte im Keller (sein R&D Labor) am »hacken« des Starfinders verbracht hat. Ich bin auch meinen Eltern dankbar, dass sie Michael aus dem Keller »rauszogen« und so die möglichen physischen (Unterkühlung) und psychischen (Vereinsamung) Schäden noch rechtzeitig verhindern konnten.

Bedanken möchte ich mich auch bei meinen Schwiegereltern für den Urlaub in Sölden (ich schreibe dieses Vorwort gerade in einem Gasthof namens »Waldesruh«), die Kegelabende und die gemeinsamen Ski-Abfahrten (es werden nur noch selten: UpsideDown-Exception, UnknownDirectionError oder SnowCrashException geworfen).

Auch bei meinem Lektor, Herrn Martin Asbach möchte ich mich vor allem für seine Geduld (der Abgabetermin wurde auf meine Bitte bereits mehrmals verschoben …), seine Offenheit für neue Ideen und die Unterstützung bedanken.

Für Fragen und Anregungen stehe ich jederzeit gerne zur Verfügung – schreiben Sie mir einfach eine eMail an abien@adam-bien.com oder besuchen Sie meine Homepage *www.adam-bien.com*

Die lauffähige StarFinder Anwendung kann auch unter *www.star-finder.com* live ausprobiert werden.

Heimstetten/Sölden, den 03.11.2002

Adam Bien

(http://www.adam-bien.com)

Vorwort 2

Der Beruf eines freiberuflichen Java-Beraters vermittelt den persönlichen Kontakt zu einer großen Anzahl engagierter Entwickler und Anwendungs-Architekten. In jedem durchgeführten Seminar, Workshop oder Projekt taucht eine Vielzahl interessanter Fragen mit Bezug zur praktischen Programmierung auf. Jede gefundene Lösung entspricht einer konkreten Strategie, die auch in anderen Szenarien wieder verwendet werden kann. Dieser glückliche Umstand führt dazu, dass im Laufe der Jahre ein eigenes Repertoire von Antworten entsteht, das zwar einerseits mit jedem weiteren Projekt vergrößert wird, andererseits aber immer weitere Bereiche abdecken kann.

Als ich seit Herbst letzten Jahres eng mit Adam Bien zusammenarbeiten konnte und er mich nach Fertigstellung seines »J2EE Patterns«-Buch fragte, ob wir nicht unsere Erfahrungen zusammenlegen und ein Buch über J2EE-FAQs schreiben wollen, war ich natürlich sofort »dabei«. Verschiedenste Demonstrationsbeispiele aus Seminaren, anskizzierte Klassendiagramme und realisierte Projektlösungen dienten als Grundlage für die behandelten Themen. Auswahl und Zusammenstellung der einzelnen »Hotspots« deckt meiner Meinung nach ein weites Spektrum der aktuellen Problembereiche ab, kann aber natürlich nur subjektiv und unvollständig sein. Und kaum ist ein Komplex behandelt, fallen einem schon wieder andere Möglichkeiten ein, die natürlich auch höchst interessant zu analysieren und zu schildern wären. Die Thematik des Buches gleicht einem Fass ohne Boden …

Trotzdem bin ich überzeugt davon, dass jeder Leser eine Menge nutzbringender Informationen gewinnen kann und interessante Sichtweisen präsentiert bekommt. Es werden alle Schichten der J2EE angesprochen und viele unterschiedliche Technologien im Einsatz gezeigt. Obwohl auch Schwierigkeiten und Unzulänglichkeiten der J2EE-Plattform aufgetreten sind und diese natürlich nicht verschwiegen werden, ist mein persönliches Vertrauen in die J2EE-Plattform und die diese unterstützenden Produkte nach dieser Arbeit nochmals gestiegen, da schließlich doch praktische Lösungen gefunden werden konnten.

Ohne Unterstützung kann nur schwer effizient gearbeitet werden. Bedanken möchte ich mich deshalb bei allen Beteiligten, insbesondere auf Verlagsseite bei Herrn Martin Asbach für Anregungen und die kritische Durchsicht der Manuskripte. Ohne das Verständnis und den Rückhalt meiner Frau Carola Fabricius hätte ich niemals Zeit und

Gelegenheit für diese Arbeit gefunden, auch ihr sei an dieser Stelle herzlich gedankt. Den größten Anteil an diesem Buch hat aber zweifellos mein Bruder Helmuth, ohne dessen stille Ermunterung und Förderung ich die Realisierung wohl niemals gewagt hätte.

München, den 02.11.2002

Rainer Sawitzki

(http://www.rainer-sawitzki.de)

1 Der Client

Die Client-Schicht des J2EE-Programmiermodells ist auf dem Endbenutzer-System angesiedelt. Während die Komponenten der Präsentationsschicht (Servlets und Java ServerPages) und der Geschäftslogik (Enterprise JavaBeans) in wohldefinierten Containern installiert werden und diesen damit vom Applikationsserver eine komfortable Umgebung zur Verfügung gestellt werden kann, ist auf dem Client eine große Variationsbreite gegeben: Schon im reinen Java-Umfeld begegnen wir Swing-GUIs, HTML-Formularen mit und ohne Applets und immer häufiger auch Anwendungen der Java 2 Micro Edition, die auf einem Kleingerät installiert sind. Nachdem neben RMI/IIOP mit http, IIOP SOAP weitere Protokolle zum Ansprechen des Servers sprachunabhängig gehalten sind, erhöht sich die Anzahl potenzieller Clients nochmals drastisch.

1.1 Installation einer EJB-Anwendung auf dem Client-Rechner

1.1.1 Problemstellung

Welche Klassen-Bibliotheken müssen im Klassenpfad des Clients gefunden werden, um den direkten Zugriff auf eine EnterpriseBean zu ermöglichen?

1.1.2 Technischer Hintergrund

Die Kommunikation zwischen dem Client und dem Applikationsserver erfolgt durch entfernte Methodenaufrufe. In der J2EE-Spezifikation ist hierfür RMI/IIOP vorgesehen. Auf Client-Seite repräsentiert der Stub das Server-Objekt und dieser delegiert die Methodenaufrufe über das Netzwerk-Protokoll an den Server weiter. Der Vorteil dieses Ansatzes im Vergleich zu http und SOAP ist evident: Spezielle Implementierungen enthalten Logik wie beispielsweise client-seitiges Fail Over oder die transparente Übermittlung von Authentifizierungsinformationen. Weiterhin ist die Geschwindigkeit der Kommunikation häufig wesentlich größer. Nachteilig ist jedoch, dass innerhalb der Stub-Implementierung applikationsserverabhängige Bytecode-Anweisungen auf dem Client ausgeführt werden.

Beim momentanen Stand der Technik müssen zwar für konkrete EnterpriseBeans keine speziellen Stub-Klassen mehr generiert werden, da der Dynamic Proxy des `java.lang.reflect`-Pakets verwendet werden kann. Die Stub-Klasse bleibt aber dennoch eng an den konkreten Applikationsserver gekoppelt und muss in den Klassenbibliotheken des Herstellers enthalten sein. Eine Java-Applikation muss zur Laufzeit im Klassenpfad hersteller-abhängige Klassen finden.

1.1.3 Lösung

Greift ein Client auf ein EnterpriseBean zu, so müssen zumindest die Typen des Paketes `javax.ejb` auf dem Client installiert werden. Als Jar-Datei benötigen diese Klassen etwa 13 KByte. Eine Datei namens `jndi.properties` im Klassenpfad enthält den vollqualifizierten Klassennamen und die URL der `InitialContextFactory` des Applikationsservers:

```
#JNDI-Konfigurationsdatei:
#Referenz-Implementierung
java.naming.factory.initial=com.sun.enterprise.naming.SerialInitContextFactory
#Weblogic
#java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
#java.naming.factory.url=t3://localhost:7001
#JBoss
#java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
#java.naming.provider.url=jnp://localhost:1099
```

Durch das JNDI-API wird ein Import der hersteller-abhängigen Pakete `com.sun`, `weblogic` und `org.jnp` vermieden. Dennoch kann selbst der einfachst denkbare J2EE-Client zwar kompiliert, nicht jedoch gestartet werden, da die angegebene ContextFactory nicht gefunden werden kann:

```
import javax.naming.*;
import javax.ejb.*;
import javax.rmi.PortableRemoteObject;
public class EJBClient{
public static void main(String[] args){
try{
 InitialContext ctx = new InitialContext();
 Object object = ctx.lookup(System.getProperty("com.hotspots.lookup"));
 EJBHome home =
 (EJBHome)PortableRemoteObject.narrow(object, EJBHome.class);
}
catch(Exception pException){
 pException.printStackTrace();
}}}
```

Alle weiteren Klassen, die zur Laufzeit benötigt werden, müssen entweder lokal auf dem Client installiert werden, oder zumindest einmal über das Netz geladen werden. Dies sind für die Referenz-Implementierung etwas über 400 Klassen. Diese können leicht über die Aufrufoption –`verbose:class` aufgelistet werden und sind als Teilmenge für die Referenz-Implementierung von Sun im `j2ee.jar` enthalten (Größe noch etwa 650 KByte). Bea Weblogic (`weblogic.jar`) und der JBoss (Mehrere Archive im %JBOSS_HOME%\client-Verzeichnis) benötigen etwa die gleiche Menge Klassen. Diese Java-Archive müssen auf dem Client installiert bzw. über das Netz verfügbar gemacht werden und müssen bei einem Wechsel des Applikationsservers oder einem Versionswechsel komplett ausgetauscht oder ergänzt werden.

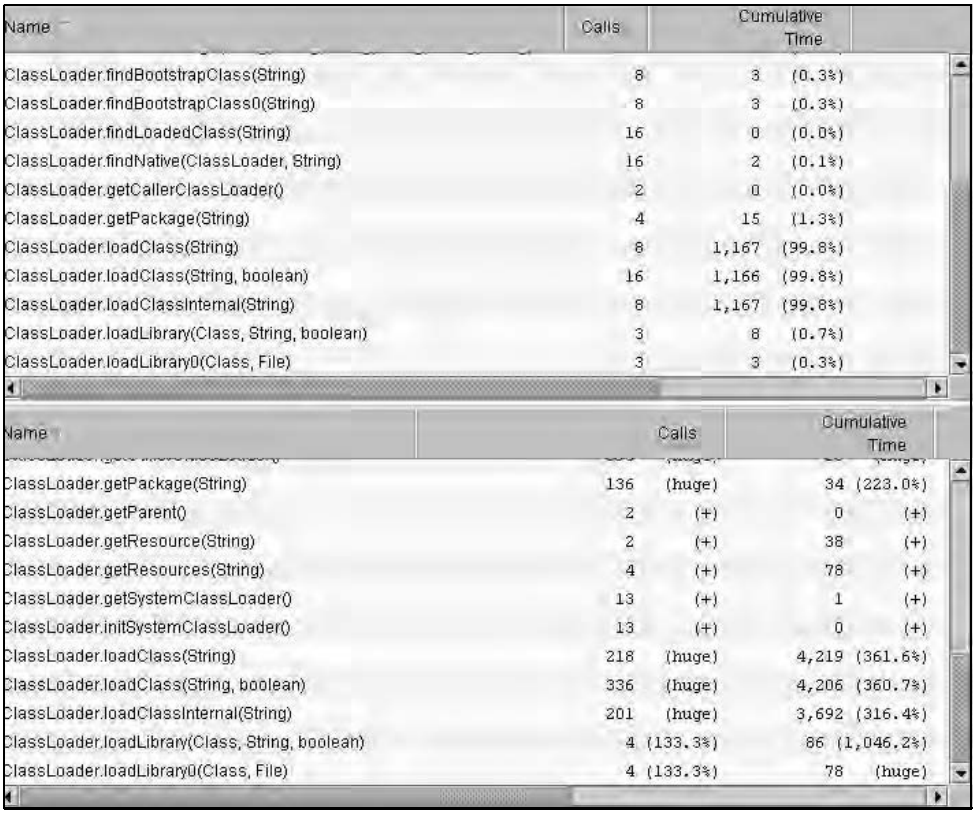
1.1.4 Performance

Jede laufende Java-Anwendung, die mit einem Applikationsserver kommuniziert, hat damit einen Overhead von mehreren Megabyte an Klassenobjekten im Heap-Speicher der virtuellen Maschine, die, falls sie durch den normalen Klassenlader geladen wurden, zur Laufzeit der Anwendung niemals mehr entfernt werden können. Für eine einfache Java-Anwendung alloziert die virtuelle Maschine unter Windows 2000 etwas über 5 Megabyte Hauptspeicher. Werden zusätzlich nur all jene Klassen, die zum Ansprechen des Applikationsservers der J2EE-Referenzimplementierung benötigt werden, geladen, werden bereits weit über 9 Megabyte verbraucht. Bei der Messung des Speicherbedarfs ist darauf zu achten, dass die Methoden der Klasse `java.lang.System` zur Messung des Speicherverbrauchs ungeeignet sind, da die Klassenobjekte in einem davon nicht zugänglichen Teil der virtuellen Maschine abgelegt werden.

Anwendungstyp	**Speicherverbrauch in kByte**
einfache Anwendung	4.960
zusätzlich geladene Klassen für die Kommunikation mit dem Applikationsserver	11.500

Tabelle 1.1: Speicherverbrauch einer einfachen Anwendung im Vergleich zu einer Anwendung mit geladenen Kommunikationsklassen zum Applikationsserver. Windows 2000 mit Laufzeitumgebung des J2SE 1.4.0 und J2EE 1.3.1 Referenzimplementierung

Auch die Ladezeit dieser Klassen ist nicht zu unterschätzen: Stehen die Bytecode-Dateien lokal zur Verfügung, werden dafür, wenn auch nur einmalig, auf einem 1,5 GHz-Client etwa 1.200 Millisekunden benötigt. Um dieses Problem beim Aufruf einer Client-Anwendung zu umgehen, können die Klassen entweder in einem separaten Thread beim Start der Anwendung geladen werden oder aber eine Erweiterung des J2EE-Patterns »Business Delegate« stellt allen Java-Anwendungen auf einem Rechner eine gemeinsam nutzbare Schnittstelle zur Verfügung. Die letzte Anwendung ist Inhalt eines der folgenden Hotspots.



Name	Calls	Cumulative Time
ClassLoader.findBootstrapClass(String)	8	3 (0.3%)
ClassLoader.findBootstrapClass0(String)	8	3 (0.3%)
ClassLoader.findLoadedClass(String)	16	0 (0.0%)
ClassLoader.findNative(ClassLoader, String)	16	2 (0.1%)
ClassLoader.getCallerClassLoader()	2	0 (0.0%)
ClassLoader.getPackage(String)	4	15 (1.3%)
ClassLoader.loadClass(String)	8	1,167 (99.8%)
ClassLoader.loadClass(String, boolean)	16	1,166 (99.8%)
ClassLoader.loadClassInternal(String)	8	1,167 (99.8%)
ClassLoader.loadLibrary(Class, String, boolean)	3	8 (0.7%)
ClassLoader.loadLibrary0(Class, File)	3	3 (0.3%)

Name	Calls	Cumulative Time
ClassLoader.getPackage(String)	136 (huge)	34 (223.0%)
ClassLoader.getParent()	2 (+)	0 (+)
ClassLoader.getResource(String)	2 (+)	38 (+)
ClassLoader.getResources(String)	4 (+)	78 (+)
ClassLoader.getSystemClassLoader()	13 (+)	1 (+)
ClassLoader.initSystemClassLoader()	13 (+)	0 (+)
ClassLoader.loadClass(String)	218 (huge)	4,219 (361.6%)
ClassLoader.loadClass(String, boolean)	336 (huge)	4,206 (360.7%)
ClassLoader.loadClassInternal(String)	201 (huge)	3,692 (316.4%)
ClassLoader.loadLibrary(Class, String, boolean)	4 (133.3%)	86 (1,046.2%)
ClassLoader.loadLibrary0(Class, File)	4 (133.3%)	78 (huge)

Abbildung 1.1: Der zusätzliche Aufwand beim Laden der Klassen, dargestellt mit JProbe 4.0. Vergleich zweier Anwendungen, einmal ohne (oben) einmal mit (unten) Zugriff auf einen EJBHome-Stub des Applikationsserver.

1.2 Application Clients

1.2.1 Problemstellung

Was ist der »Application Client« der J2EE-Spezifikation?

1.2.2 Technischer Hintergrund

Die J2EE-Plattform spezifiziert ein durchgängiges Programmiermodell von der Anwendung auf dem Endbenutzer-PC bis hin zum Enterprise Information System mit der zentralen Datenhaltung in einem Großrechnersystem. Pragmatisch formuliert besteht dieses Modell aus folgenden Vereinbarungen:

- Referenzen auf andere J2EE-Komponenten werden stets durch einen Lookup auf einen JNDI-Kontext erhalten.

- Der JNDI-Kontext enthält neben einem für alle Applikationen zugänglichen Teil einen für jede Komponente privaten Bereich, der mit dem Subkontext »java:comp/env« angesprochen wird. Darin kann eine Komponente unter einem vom Entwickler frei wählbaren Namen Umgebungsvariablen, andere Komponenten und Ressourcen Factories angeben. Während des Deployments werden diese symbolischen Alias-Namen mit realen JNDI-Namen und damit gebundenen Objekten verknüpft.
- Alle Komponenten werden in einen Container installiert, der den Lebenszyklus kontrolliert und weitere Dienste zur Verfügung stellt.
- Die Konfiguration der Komponenten erfolgt innerhalb einer anhand einer öffentlich zugänglichen Document Type Definition prüfbaren XML-Datei, dem Deskriptor.
- Eine komplexere Anwendung wird durch Konfiguration mehrerer J2EE-Komponenten erreicht, die im Deskriptor vorgenommen wird.
- Die J2EE-Container halten auch Benutzerinformationen, die zur Authentifizierung und Authorisierung verwendet werden können. Die Propagierung dieser Informationen erfolgt automatisch durch den Container.
- Zur Anmeldung wird der *Java Authentication & Authorization Service* (JAAS) verwendet.

Als J2EE-Komponenten werden zwar meistens Servlets, Java ServerPages und EnterpriseBeans verstanden, aber auch die Standalone-Clients und Applets sind in der J2EE-Spezifikation enthalten und werden mit einem Deskriptor konfiguriert. Die zugehörige Document Type Definition namens »application-client_1_3.dtd« enthält die unterstützten Elemente.

Eine Implementierung eines Application Clients ist im Wesentlichen eine Klasse, die jede vorhandene lauffähige Klasse mit zusätzlichen Funktionalitäten umhüllt:

- Auslesen des Client-XML-Deskriptors.
- Authentifizierung mit Eingabe von Benutzer und Kennwort durch Verwendung des angegebenen JAAS-Callback-Handlers. Damit muss sich die eigentliche Anwendung nicht mehr um die Anmelderoutine und das Übermitteln von Prinzipal und Credential kümmern.
- Erzeugung eines InitialContext, der die symbolischen JNDI-Namen der Anwendung in die im Deskriptor konfigurierten realen Namen umsetzt.

Application Clients sind bei rein web-basierten J2EE-Anwendungen, die auf Applets verzichten können, nicht erforderlich. Das Wissen um die konkrete Verwendung des Application Clients ist deshalb überraschend wenig verbreitet.

1.2.3 Lösung

Genau wie ein Servlet oder ein Enterprise JavaBean enthält der Deskriptor des Application Client Umgebungseinträge, Referenzen auf EnterpriseBeans, und konfiguriert Ressourcen. Innerhalb des Client-Codes werden diese Referenzen, wie im J2EE-Modell üblich, durch den Subkontext »java:comp/env« angesprochen.

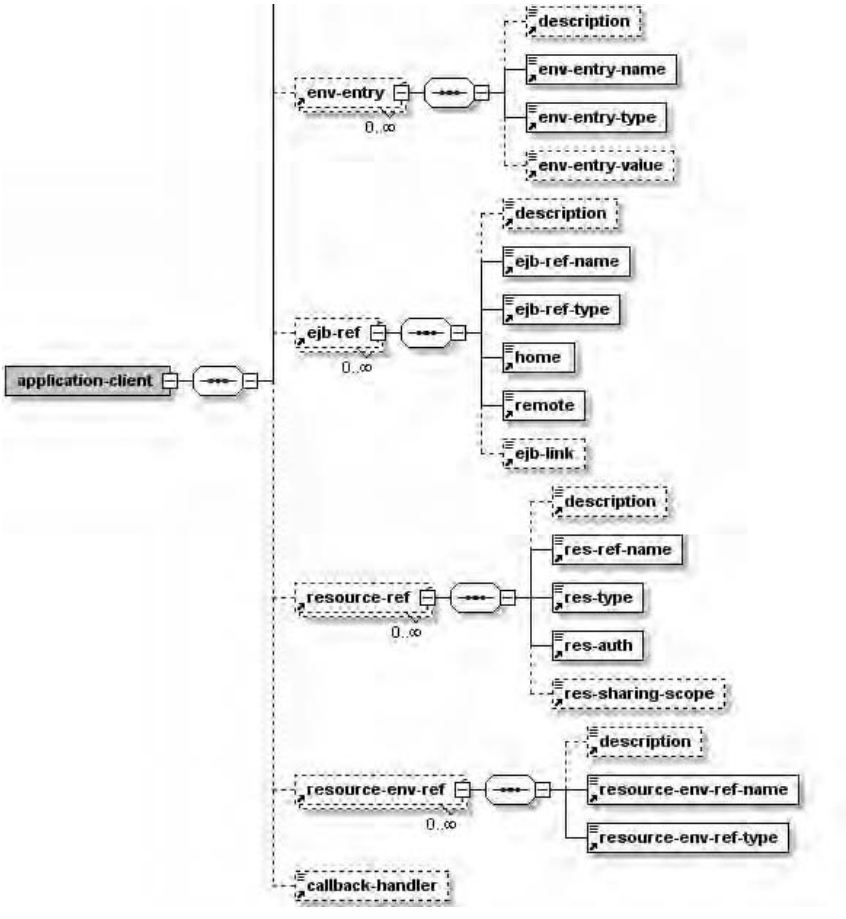


Abbildung 1.2: Die Document Type Definition eines J2EE-Application Clients, J2EE-Spezifikation 1.3

Damit kann ein Client natürlich genau so einfach konfiguriert werden wie beispielsweise eine EnterpriseBean: Sämtliche Lookup-Strings können im Programm beliebig definiert und im Deskriptor dann den konkreten Namen angepasst werden. Die Notwendigkeit, eigene Properties-Dateien zur Konfiguration zu verwenden, wird damit umgangen.

Im Quellcode des Clients:

```
ServiceLocator locator = ServiceLocator.getInstance() ;
String title =
(String)locator.getEnvironment().lookup("java:comp/env/Titel");
EJBHome home = locator.getHome("java:comp/env/ejb/StarFassade");
```

Im Deskriptor des Application Clients:

```
<application-client>
<env-entry>
<env-entry-name>Titel</env-entry-name>
<env-entry-type>java.lang.String</env-entry-type>
<env-entry-value>Starfinder</env-entry-value>
</env-entry>
<ejb-ref>
<ejb-ref-name>ejb/StarFassade</ejb-ref-name>
<ejb-ref-type>remote</ejb-ref-type>
<ejb-ref-home>com.abien.j2ee.starsearch.facade.StarFacadeHome</ejb-ref-home>
<ejb-ref-remote>com.abien.j2ee.starsearch.facade.StarFacade</ejb-ref-remote>
<ejb-link>StarFacadeBean</ejb-link>
</ejb-ref>
</application-client>
```

1.2.4 Praxis

Die Implementierung des Client Containers ist Aufgabe des Herstellers des Applikationsservers. Für die Referenz-Implementierung von Sun ist dies die Klasse `com.sun.enterprise.appclient.Main`, die mit dem `runclient`-Batchjob gestartet wird. Als Übergabeparameter wird eine Client-JAR-Datei erwartet, die die ausführbare Klasse, alle benötigten Hilfsklassen sowie den Deskriptor enthält. Der Aufruf erfolgt dann über

```
runclient -client application.jar -name AppClient
```

Jede Client-Anwendung lässt sich mit dem Application Client umhüllen, um die Authentifizierung beim Applikationsserver auszuführen. Dazu ist es nur notwendig, die Starter-Klasse (und alle abhängigen Klassen) als Application Client der J2EE-Anwendung hinzuzufügen.

Ein J2EE-Application Client wird für die Referenz-Implementierung von Sun mit Hilfe des Deployment Tools erstellt:

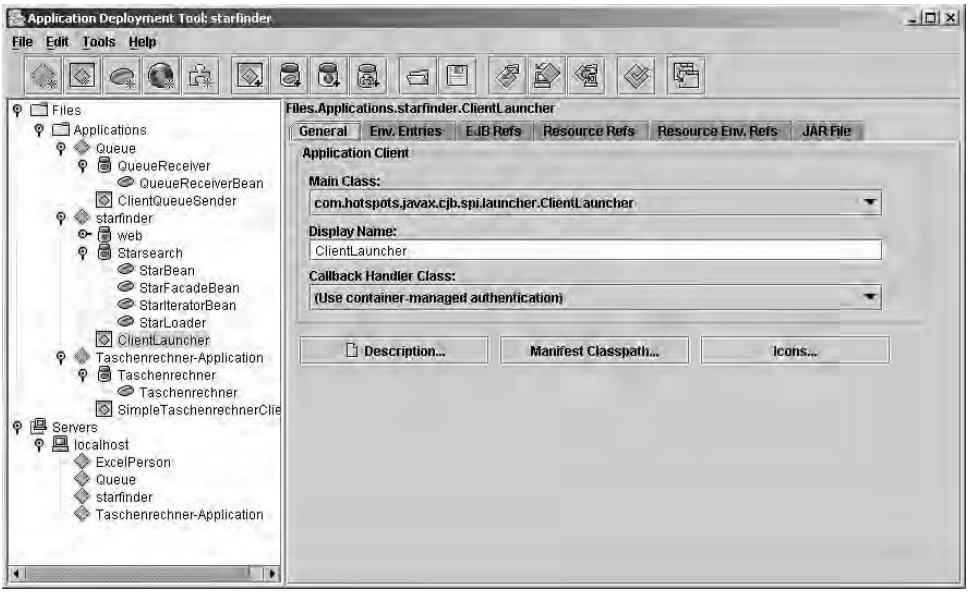


Abbildung 1.3: Ein Application Client wird im Deployment Tool der Referenz-Implementierung von Sun definiert.

Wird die Anwendung durch den Aufruf

```
runclient -client starfinderClient.jar -name ClientLauncher
```

gestartet, wird als Erstes die JAAS-Authentifizierung initiiert. Mit den hier gewählten Einstellungen öffnet sich ein einfacher Anmeldedialog:

Abbildung 1.4: Der JAAS-Anmeldedialog der Referenz-Implementierung von Sun

Dies ist nichts anderes als der Default Callback-Handler, nämlich die Klasse `com.sun.enterprise.security.auth.login.LoginCallbackHandler`, die als Dialog den `com.sun.enterprise.security.GUILoginDialog` verwendet. Der Callback-Handler-Mechanismus ist Bestandteil des nächsten Hotspots.

Erst nach Beenden dieses Dialogs wird die eigentliche Client-Anwendung gestartet. Die eingegebenen Daten (Benutzername und Passwort als Zeichenkette) werden innerhalb des Application Clients gehalten. Die Stubs senden diese Zusatzinformationen bei jedem Methodenaufruf an den Applikationsserver.

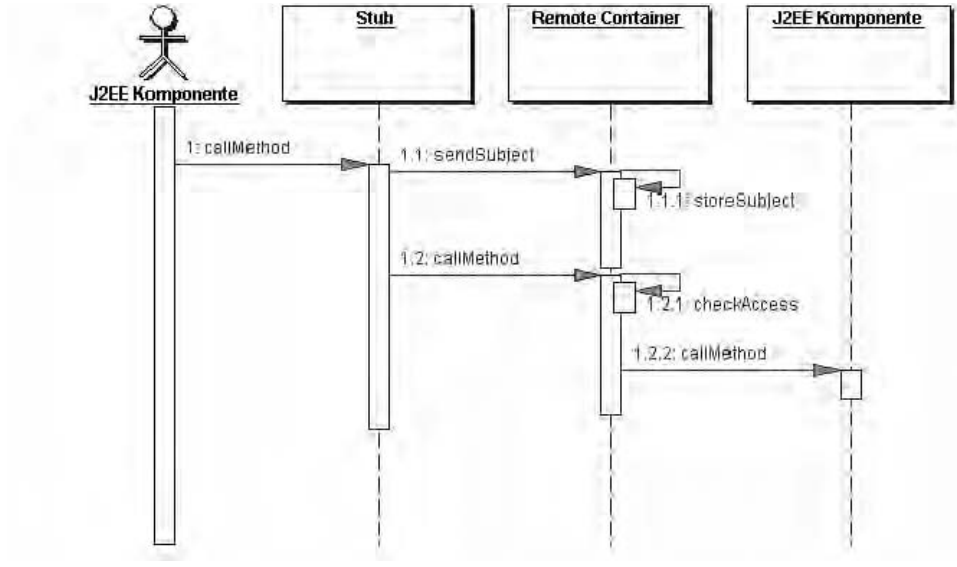


Abbildung 1.5: Prinzip der Übertragung der Anmeldeinformationen vom Client zum Applikationsserver

Damit kann eine Enterprise JavaBean sofort vor dem Zugriff nicht-authentifizierter Benutzer geschützt werden, indem innerhalb des Applikationsservers Rollen definiert und diesen dann reale Benutzer zugeordnet werden. So wird durch die folgende Einstellung die Suchfunktionalität einer Anwendung nur für die Rolle »all« zugelassen; »Gäste« dürfen keine Suche durchführen (siehe Abbildung 1.6).

Der Rolle »all« sind dann verschiedene Benutzer oder Benutzergruppen zugeordnet, hier nur der Name »j2ee« (siehe Abbildung 1.7).

Durch die Verwendung des Application Clients ist es möglich, die Authentifizierung beim Applikationsserver transparent für die Anwendung durchzuführen. Die eigentliche Arbeit übernimmt der JAAS.

Das Konzept des Application Clients wird von Weblogic und JBoss leider nur recht mäßig unterstützt. Geht es hauptsächlich um die automatische Authentifizierung, ist dies ist aber keine sonderlich große Einschränkung. Im nächsten Abschnitt wird ein generischer Applikationsclient vorgestellt, der für alle Applikationsserver funktionieren wird.

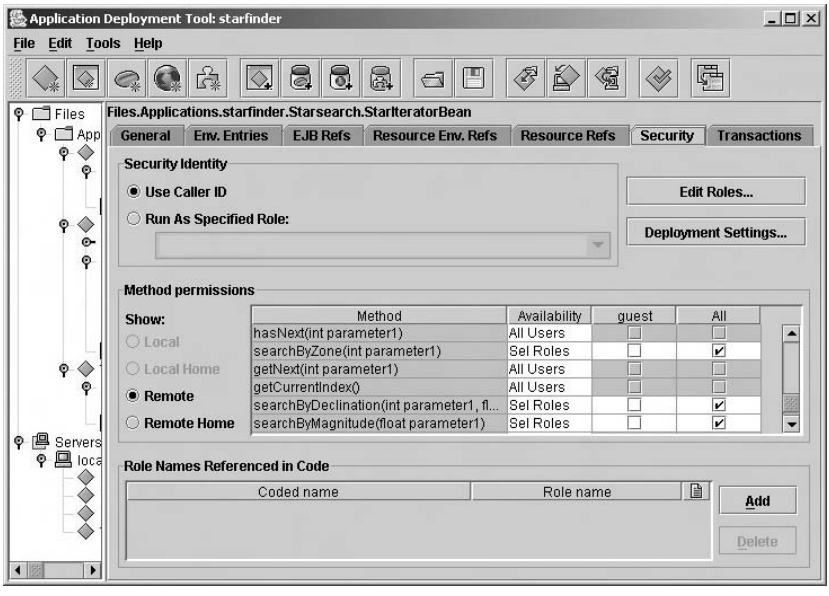


Abbildung 1.6: Deklarative Berechtigungen für eine Client-Anwendung: Nur die Benutzerrolle »all« erhält die Berechtigung, etwa die Such-Methode »searchByZone« aufzurufen

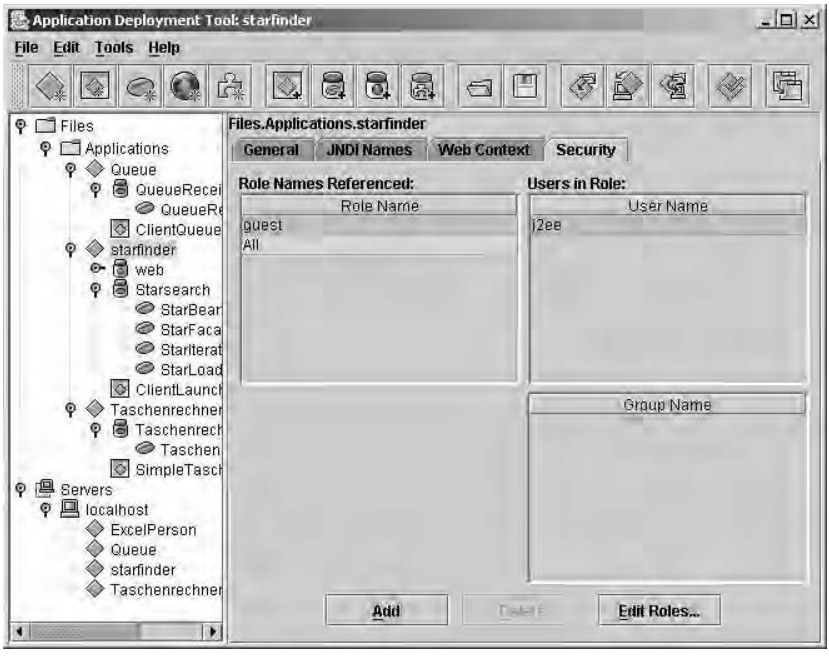


Abbildung 1.7: Zuordnung von Benutzern zu einer Rolle mit dem Deployment Tool der Referenz-Implementierung von Sun

1.2.5 Performance

Als Applikationsserver werden die Referenz-Implementierung von Sun, BEA Weblogic 7.0 und der JBoss 3.0 verwendet. Ziel der Messung ist, einen Eindruck davon zu bekommen, wie die Übertragung der Benutzer-Informationen vom Client zum Applikationsserver sowie die Authentifizierung auf Server-Seite die Methodenaufrufzeiten beeinflussen wird. Dazu dient eine simple Enterprise JavaBean mit den drei Methoden:

```
public void doVoid(){
}
public double doPrimitive(double pDouble){
return pDouble;
}
public Object doObject(Object pObject){
return pObject;
}
```

Der Client ruft 1.000 Mal hintereinander diese Methoden auf, wobei als Objekt ein 30.000 Elemente umfassendes Byte-Array verwendet wird. Client und Applikationsserver befanden sich jeweils auf der gleichen Maschine.

Server	Direkter Aufruf [msec]	Aufruf mit Authentifizierung [msec]	Differenz [msec]	Mehraufwand pro Aufruf [msec]
Referenz-Implementierung 1.3.1	1.482	1.549	67	0,067
	1.532	1.626	94	0,094
	7.694	7.884	190	0,190
BEA Weblogic 7.0	538	554	16	0,016
	581	560	21	0,021
	3.144	3.258	114	0,114
JBoss 3.0	1.592	2.003	411	0,411
	2.003	2.390	387	0,387
	5.849	6.126	277	0,277

Tabelle 1.2: Zusätzlicher Zeitaufwand für die Übermittlung der Authentifizierungs-Informationen für verschiedene Applikationsserver. In der jeweils oberen Reihe der Aufruf der Methode doVoid, darunter doPrimitive und unten doObject. Client und Server befanden sich auf der selben Maschine.

Der Mehraufwand ist zwar durchaus messbar, für einen einzelnen Methodenaufruf jedoch mit unter einer Millisekunde faktisch vernachlässigbar. Auch die Prüfung der Berechtigung auf dem Applikationsserver fällt kaum ins Gewicht. Für die Referenz-Implementierung und den Weblogic ergeben sich Werte von etwa 0,3 msec bzw. 0,2 msec pro Aufruf.

1.3 Authentifizierung beim Applikationsserver mit dem JAAS

1.3.1 Problemstellung

Wie kann sich der Standalone Client beim Applikationsserver authentifizieren und wie erfolgt die Eingabe des Benutzernamens und des Passworts?

1.3.2 Technischer Hintergrund

Zur Anmeldung beim Applikationsserver können zwei unterschiedliche Technologien verwendet werden:

- Anmeldung beim JNDI-Kontext,
- Verwendung des Java Authentication and Authorization Services, JAAS.

Bei der ersten Alternative werden Benutzername und Kennwort bei der Erzeugung des InitialContext in Form einer Hashtable mit den folgenden Schlüsseln mitgegeben:

Statische Konstante in javax.naming.Context	**Beschreibung**	**Zugeordnete Umgebungsvariable**
SECURITY_AUTHENTICATION	Security-Level, »none«, »simple« oder »strong«	java.naming.security.authentication
SECURITY_PRINCIPAL	Principal (User)	java.naming.security.principal
SECURITY_CREDENTIALS	Credential (User und Legitimation)	java.naming.security.credentials
SECURITY_PROTOCOL	Verwendetes Protokoll, z. B. »SSL«	java.naming.security.protocol

Tabelle 1.3: Konstanten zur Anmeldung bei einem JNDI-Provider

Principal und Credential müssen vor der Instanzierung des `InitialContext` in der Anwendung erzeugt werden. Dies kann sich beim Caching von Context-Implementierungen als sicherheitskritisch erweisen.

Der Principal ist eine Abstrahierung einer Person, eines Unternehmens oder einer Benutzerkennung. Modelliert wird der Principal durch das Interface `java.security.Principal` (siehe Abbildung 1.8).

Zur Authentifizierung wird jedoch ein »Credential« benötigt. Was ist dies aber nun genau? Die Antwort ist leider wesentlich allgemeiner als gewünscht: Ein Credential ist die Möglichkeit, eine Identität mit einem Authorisierungs-Werkzeug prüfen zu lassen. Damit scheidet jedoch die Definition eines eigenen Credential-Interfaces aus.

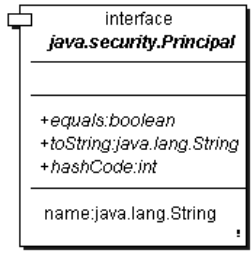


Abbildung 1.8: Das Interface java.security.Principal

Die Bibliotheken eines Applikationsservers enthalten eigene Klassen, die die Rolle des Principals und des Credentials übernehmen können.

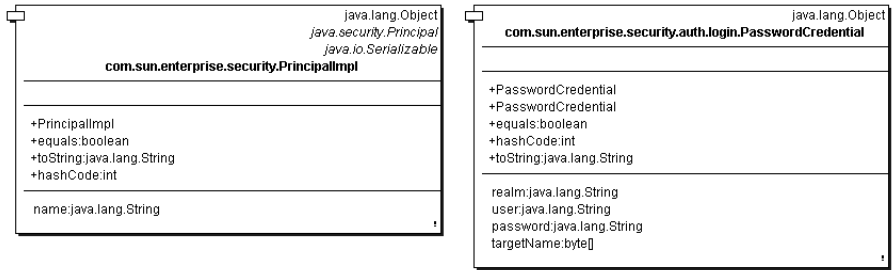


Abbildung 1.9: Die Security-Implementierungen von Principal und Credential der Referenz-Implementierung von Sun

Die Verwendung dieser Klassen direkt in der Client-Anwendung würde zu einer unzulässigen Kopplung mit einem bestimmten Applikationsserver führen. Der JAAS delegiert deshalb die Erzeugung an ein »Login-Modul«, eine Implementierung des Interfaces javax.security.auth.spi.LoginModule. Diese Schnittstelle ist Bestandteil des JAAS-SPI und muss deshalb vom Hersteller implementiert werden (siehe Abbildung 1.10).

Welche Anmelde-Module konkret verwendet werden, wird in einer Konfigurationsdatei definiert, welche als Bestandteil der Java Laufzeitumgebung aufzufassen ist.

Das Halten der Informationen erfolgt vollständig getrennt von der Applikation in der JAAS-Implementierung des Applikationsserver-Herstellers.

Speziell für den Application Client ist das Element namens »`callback-handler`«, mit dem der Anwender die Anmeldeinformationen eingeben kann. `CallbackHandler` ist ein Interface im Paket `javax.security.auth.callback`. Darin ist die Methode `handle(Callback[] callbacks)`deklariert, die während des Login-Vorgangs vom JAAS automatisch aufgerufen wird. `Callback` ist ein reines Marker-Interface. In der einfachsten Form liest ein Callback-Handler Benutzernamen und Kennwort von der Konsole oder aus einem Anmeldedialog aus.

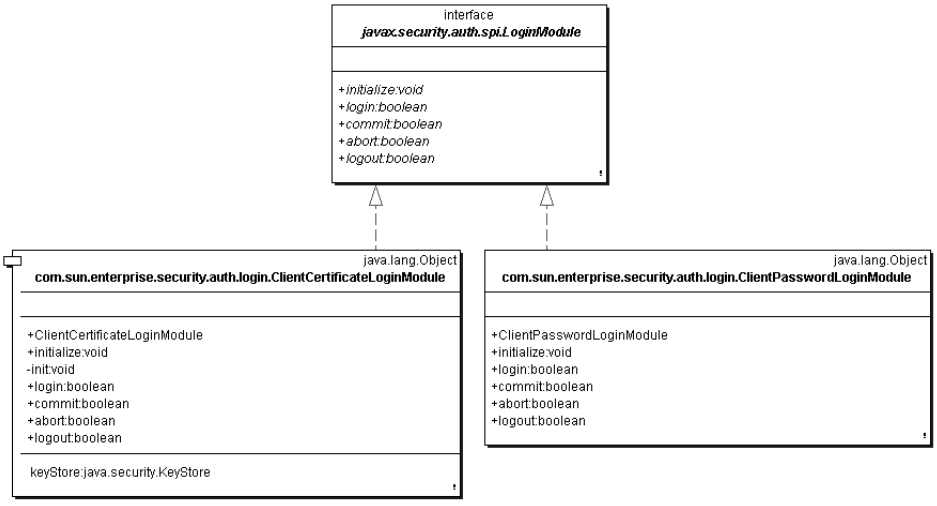


Abbildung 1.10: Implementierung des LoginModule-Interfaces in der Referenz-Implementierung von Sun

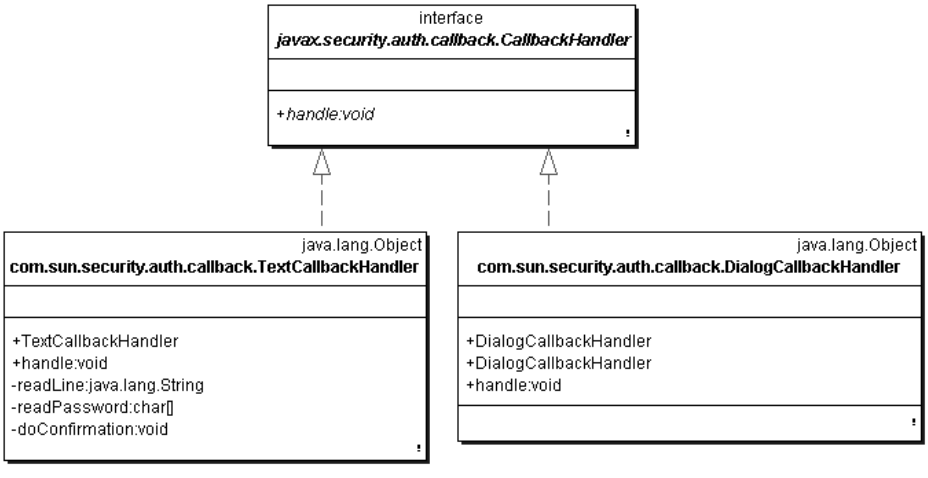


Abbildung 1.11: Callback-Handler der Referenz-Implementierung von Sun

Principal und Credentials werden nicht direkt verwendet, sondern sind in einem `javax.security.auth.Subject` gehalten. Ein Subject kann mehrere Principals und Credentials halten, da ein Benutzer sich gegen verschiedene Systeme mit unterschiedlichen Kennungen (Benutzername, Personalnummer usw.) identifizieren kann.

Welche Login-Module und welche Callback-Handler verwendet werden, wird in einer JAAS-Konfigurationsdatei definiert, deren Lokation mit Hilfe der Umgebungsvariable `java.security.auth.login.config` beim Programmstart angegeben werden kann. Diese

Datei enthält im Wesentlichen mehrere JAAS-Kontexte, in denen dann jeweils ein LoginModul eingetragen wird. Für die Referenz-Implementierung von Sun ist für den Client im Verzeichnis `%J2EE_HOME%/lib/security` die Datei `clientlogin.config`:

```
default {
com.sun.enterprise.security.auth.login.ClientPasswordLoginModule required
debug=false;
};
certificate {
com.sun.enterprise.security.auth.login.ClientCertificateLoginModule required
debug=false;
};
```

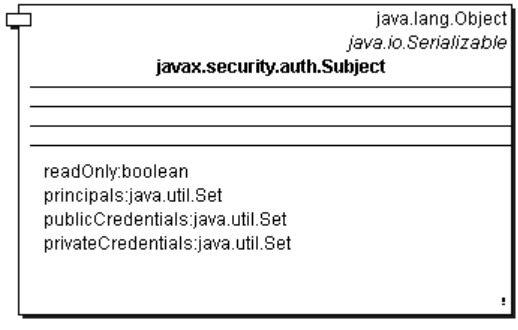


Abbildung 1.12: Die Properties der Subject-Klasse

Das Konzept des Application Clients beruht stark auf dem JAAS, der aber erst ab der J2SE 1.4 Version in die Klassenbibliothek aufgenommen wurde und vorher nur als Java Extension vorhanden war. Deshalb unterstützen erst die neuesten Generationen der Applikationsserver den Application Client, vorher wurde die Anmeldung bei der Instanzierung des `InitialContext` durchgeführt. Die vom Client gelieferten Benutzer-Informationen werden innerhalb des Applikationsservers verwaltet und werden für das deklarative Berechtigungskonzept auf Methodenebene verwendet.

Das Auslesen des Deskriptors und die Initiierung des JAAS-Anmeldevorgangs muss nun aber nicht die Anwendung selbst übernehmen, dies ist Aufgabe des »Application Containers«. Die Implementierung eines funktionsfähigen Application Containers kann natürlich der Entwickler der Client Anwendung selbst übernehmen. In der Regel stellen jedoch bereits die Anbieter von Applikationsservern fertige Lösungen zur Verfügung.

1.3.3 Praxis

Zur Anmeldung beim Applikationsserver wird bei den im Projekt eingesetzten Versionen heute noch oft der JNDI-Kontext verwendet. JBoss 3.0 und Weblogic 7.0 bevorzugen jedoch bereits JAAS.

Die Weblogic Starter-Klasse ist `weblogic.j2eeclient.Main`. Zum Deployment dient die lauffähige Klasse weblogic.ClientDeployer. Die konkreten Schritte zur Erstellung des Application Clients unterscheiden sich nur im Detail vom Vorgehen bei der Referenz-Implementierung, konzeptuell sind die gleichen Abläufe durchzuführen:

- Erstellen eines Java-Archivs, das die Client-Klassen enthält,
- Hinzufügen einer XML-Datei gemäß der DTD des Application Clients,
- Verteilung auf die Client-Maschinen und
- Aufruf mit dem Client Container.

Speziell für den Weblogic ist die eine Runtime-XML-Datei neben dem Client-Archiv zu erzeugen. Diese Datei ist nicht in der JAR-Datei enthalten und muss deshalb anhand einer speziellen Namenskonvention (Name des Client-Archivs + ».runtime.xml«) im gleichen Verzeichnis abgelegt werden. Es ist zu erwarten (und auch wünschenswert), dass in Zukunft der Application Client auch von BEA komfortabler unterstützt wird.

Der Client Container des Weblogic unterstützt die Authentifizierung mit JAAS nicht direkt. Im Folgenden wird deshalb ein einfacher Application Client Container vorgestellt, der allgemein die Anmeldung beim Applikationsserver durchführen kann. Detaillierte Informationen zur Authentifizierung mit JAAS sind in einem folgenden Hotspot gegeben. Diese Anwendung ist kein vollständiger Application Client, da die Informationen aus dem Deskriptor nicht verwendet werden, sondern ergänzt den Client Container des Weblogic.

Der Application Client ist aber plattform-übergreifend definiert, so dass die eigentliche Client-Anwendung keinerlei Bezug zum Applikationsserver benötigen sollte. Beim Wechsel des Applikationsservers darf ausschließlich die Konfiguration geändert werden müssen. Um die JAAS-Funktionalität der Weblogic zu nutzen, sind von BEA eigene Callback-Handler und Login-Module in die Klassenbibliothek aufgenommen worden. Dies ist keine Besonderheit des Weblogic, sondern muss von allen Applikationsservern so oder ähnlich gehandhabt werden, da die Anmeldung hersteller-abhängige Hilfsklassen, die beispielsweise Benutzer und Passwörter halten, verwendet werden. Die Implementierung des JAAS im Weblogic besteht aus den folgenden Komponenten:

- Im Lieferumfang ist die Klasse `weblogic.security.auth.login.UsernamePasswordLoginModule` enthalten, die die benötigten Informationen in die benötigten Weblogic-Implementierungen umsetzt. Als Credential-Klasse wird

  ```
  weblogic.security.auth.login.PasswordCredential
  ```

  verwendet.

- Die Container-Klasse verwendet den `LoginContext` des Paketes `javax.security.auth.login`. Dieser benötigt wiederum einen Callback-Handler. Als Besonderheit sendet das Weblogic-Login-Modul eine eigene Implementierung der `Callback`-Schnittstelle, den URLCallback. Dieser Callback verlangt vom Benutzer die Eingabe der URL, auf der sich der Applikationsserver befindet.

- Damit wird ein eigener Callback-Handler notwendig, der den `URLCallback` ausführt.

Während die erste applikationsserver-spezifische Angabe, nämlich die Auswahl des Login-Moduls, bereits im JAAS durch die JAAS-Konfiguration abgedeckt wird, muss bei der Instanzierung des `javax.security.auth.login.LoginContext` im Client-Programm ein Callback-Handler angegeben werden. Zur Entkopplung wird eine Callback-Handler-Factory eingesetzt. Bereits vorhandene JAAS-konforme Handler können, falls gewünscht, mit der Funktionalität des `URLCallbacks` dekoriert werden:

```
public class WeblogicCallbackHandler implements CallbackHandler {
 CallbackHandler delegate;
 public WeblogicCallbackHandler(CallbackHandler pDelegate){
 delegate = pDelegate;
 }
 public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
 ArrayList list = new ArrayList();
 for (int i = 0; i < callbacks.length; i++){
 if (callbacks[i] instanceof URLCallback){
 Callback[] temp = new Callback[list.size()];
 temp = (Callback[])list.toArray(temp);
 delegate.handle(temp);
 list.clear();
 handleUrlCallback((URLCallback)callbacks[i]);
 }
 else{
 list.add(callbacks[i]);
 }
 }
 }
 private void handleUrlCallback(URLCallback pCallback) throws IOException,
UnsupportedCallbackException {
 System.err.print(pCallback.getPrompt());
 System.err.flush();
```

```
 pCallback.setURL((new BufferedReader(new
InputStreamReader(System.in))).readLine());
 }
}
```

Die URL des Weblogic, nach der Standard-Installation `t3://localhost:7001`, wird hier von der Konsole eingelesen.

Die Implementierung des Client Containers kann damit allgemein gehalten werden. Die Klasse `com.hotspots.javax.security.auth.ApplicationClient` zeigt das typische Vorgehen bei der Anmeldung:

- Setzen des SecurityManagers,
- Ausführen des Login,
- Rückgabe des `javax.security.auth.Subject`, falls sich eine Anwendung für die eingegebenen Benutzer-Informationen interessieren sollte,
- Aufrufen der Main-Methode der übergebenen Starter-Klasse.

```
public class ApplicationClient {
private LoginContext lc;
ApplicationClient(String[] args){
 if (System.getSecurityManager()==null){
 System.setSecurityManager(new SecurityManager());
 }
try {
lc = new LoginContext("default", CallbackHandlerFactory.getInstance().create());
lc.login();
Subject subject = lc.getSubject();
try{
 Subject.doAs(subject, new MainInvoker(args));
}
//Exception-Handling
}
public Subject getSubject(){
 return lc.getSubject();
}
//Deklaraton des MainInvokers, der reflektiv die Main-Methode der Starter-Klasse
aufruft
}
```

Zur Konfiguration des JAAS dient eine Konfigurationsdatei, in der der voll qualifizierte Klassenname eines LoginModules eingetragen wird. Hier enthält die Datei `clientlogin.config` den Verweis auf das Modul von Weblogic:

```
default {
weblogic.security.auth.login.UsernamePasswordLoginModule required debug=false;
};
```

Nachdem der Client in jedem Fall mit aktivem Security Manager startet, muss eine Berechtigungsdatei erstellt werden. Diese Policy-Datei muss zumindest die Einträge

```
grant{
 permission javax.security.auth.AuthPermission "createLoginContext";
 permission javax.security.auth.AuthPermission "modifyPrincipals";
 permission javax.security.auth.AuthPermission "modifyPrivateCredentials";
 permission javax.security.auth.AuthPermission "doAs";
 permission java.net.SocketPermission "*:*", "connect";
 permission java.io.SerializablePermission "enableSubstitution";
 permission java.io.FilePermission "C:${/}bea7${/}weblogic700${/}-", "read,write";
 permission java.lang.RuntimePermission "accessClassInPackage.sun.io";
 permission java.lang.RuntimePermission "createClassLoader";
 permission java.lang.RuntimePermission "getClassLoader";
 permission java.util.PropertyPermission "*", "read,write";
};
```

enthalten. Der Start erfolgt schließlich durch den Aufruf

```
%JAVA_HOME%\bin\java
-Djava.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
-Djava.naming.provider.url=t3://localhost:7001
-classpath c:\bea7\weblogic700\server\lib\weblogic.jar;%APPCLASSES%
-Djava.security.auth.login.config=clientlogin.config
-Djava.security.policy=client.policy
com.hotspots.javax.security.auth.ApplicationClient
<Name der Anwendung Anwendungsparameter>
```

Als Name der Anwendung kann auch die Starter-Klasse des Weblogic verwendet werden. Dann muss in der Berechtigungsdatei aber noch zusätzlich die Leseberechtigung für das Client-Archiv gewährt werden.

Auch der JBoss-Applikationsserver (*www.jboss.org*) verwendet JAAS. Eine geeignete Konfigurations-Datei hat das folgende Aussehen:

```
default {
 org.jboss.security.ClientLoginModule required debug=true;
};
```

Zur Anmeldung an JBoss kann der vorgestellte Client Container ohne Änderung verwendet werden.

1.4 Credentials und Anmeldung bei weiteren Ressourcen

1.4.1 Problemstellung

Wie können im Application Client oder dem Enterprise Bean die eingegebenen Benutzernamen und Passwörter ausgelesen werden, um eine Authentifizierung bei einem anderen System zu ermöglichen?

1.4.2 Technischer Hintergrund

Das J2EE-Programmiermodell propagiert den Principal und das so genannte »Credential« (in der einfachsten Form eine Klasse, die Benutzer und Passwort als String zur Verfügung stellt) automatisch vom Client zum Applikationsserver. Principal und Credential müssen jedoch in irgendeiner Schicht erzeugt und gehalten werden. Als Container dafür dient die Klasse `javax.security.auth.Subject`. Bei einer web-basierten Anwendung übernimmt diese Aufgabe ein Servlet, bei einer Standalone-Applikation muss diese Aufgabe der Endbenutzer-Client erledigen. Im letzten Fall müssen die verwendeten Stubs noch vor jedem Methodenaufruf das Credential zum Applikationsserver übermitteln, um dort die Authentifizierung zu ermöglichen. Nach erfolgreicher Authentifizierung wird die Propagierung von den beteiligten Containern automatisch durchgeführt. Die Art wird deklarativ im Deskriptor festgelegt. Es stehen die beiden Varianten »use caller ID« und »run as« zur Verfügung.

Eine EnterpriseBean im Applikationsserver hat jedoch selber keinerlei direkten Zugriff auf dieses Credential. Durch die Methode `getCallerPrincipal()` des `EJBContext`-Interfaces kann nur der Principal und damit im Wesentlichen die Anmeldekennung ausgelesen werden. Diese Beschränkung ist in der Spezifikation durchaus bewusst gewählt worden: Die Authentifizierung ist alleinige Aufgabe des EJB-Containers, nicht der installierten Komponenten.

Es offenbart sich jedoch dann ein Problem, wenn eine J2EE-Komponente die Information sowohl des Benutzernamens als auch des Credentials verwenden muss. Die automatische Propagierung der Anmelde-Informationen wird vom Container nämlich nur bei Methoden-Aufrufen durchgeführt, nicht bei der Anmeldung an weitere Ressourcen. So kann beispielsweise ein Application Client als Ressource den JMS-Provider anfordern:

```
public class ClientQueueManager implements QueueConnectionFactory{
 private Context cContext;
 public ClientQueueManager(){
 try{
 Context = (Context)(new InitialContext()).lookup("java:comp/env/jms");
 }
 catch(Exception pException){
 pException.printStackTrace();
 }
 }
 public QueueConnection createQueueConnection() throws JMSException{
 try{
 return
((QueueConnectionFactory)cContext.lookup("TheQueueConnectionFactory")).createQueue
Connection();
 }
 catch(Exception pException){
```

```
 throw new JMSException(pException.getMessage());
 }
 }
 public QueueConnection createQueueConnection(java.lang.String userName,
java.lang.String password) throws JMSException{
 try{
 return
((QueueConnectionFactory)cContext.lookup("TheQueueConnectionFactory")).createQueue
Connection(userName, password);
 }
 catch(Exception pException){
 throw new JMSException(pException.getMessage());
 }
 }
 public Queue getQueue() throws JMSException{
 try{
 return (Queue)cContext.lookup("TheQueue");
 }
 catch(Exception pException){
 throw new JMSException(pException.getMessage());
 }
 }
}
```

Sämtliche Lookups beziehen sich auf den »privaten« Kontext `java:comp/env/jms`, so dass eine direkte Verwendung dieses Managers nicht vorgesehen ist. Die folgende Anwendung muss deshalb als Application Client deployed werden:

```
public class ClientQueueSender{
 public static void main(String[] args){
 try{
 ClientQueueManager lManager = new ClientQueueManager();
 QueueConnection lConnection = lManager.createQueueConnection();
 QueueSession lSession = lConnection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
 Queue lQueue = lManager.getQueue();
 QueueSender lSender = lSession.createSender(lQueue);
 Message lMessage = lSession.createMessage();
 lMessage.setStringProperty("Message", "Test");
 lSender.send(lMessage);
 lSender.close();
 lSession.close();
 lConnection.close();
 }
 catch(Exception pException){
 pException.printStackTrace();
 }
 }
}
```

Damit werden Benutzername und Kennwort eingelesen. Doch was passiert nach der Eingabe mit dem eingegebenen Usernamen und dem Passwort? Diese müssen ja irgendwie auf dem Client gehalten werden, da bei jedem Methodenaufruf der EJB-Container den Aufrufenden kennen und prüfen muss. Im obigen Beispiel tritt das Problem sofort dann auf, wenn die Authentifizierung beim JMS-Provider mit den eingegebenen Informationen durchgeführt werden soll: Bei der Konfiguration einer ConnectionFactory kann nur unterschieden werden zwischen der container- oder der application-basierten Authentifizierung:

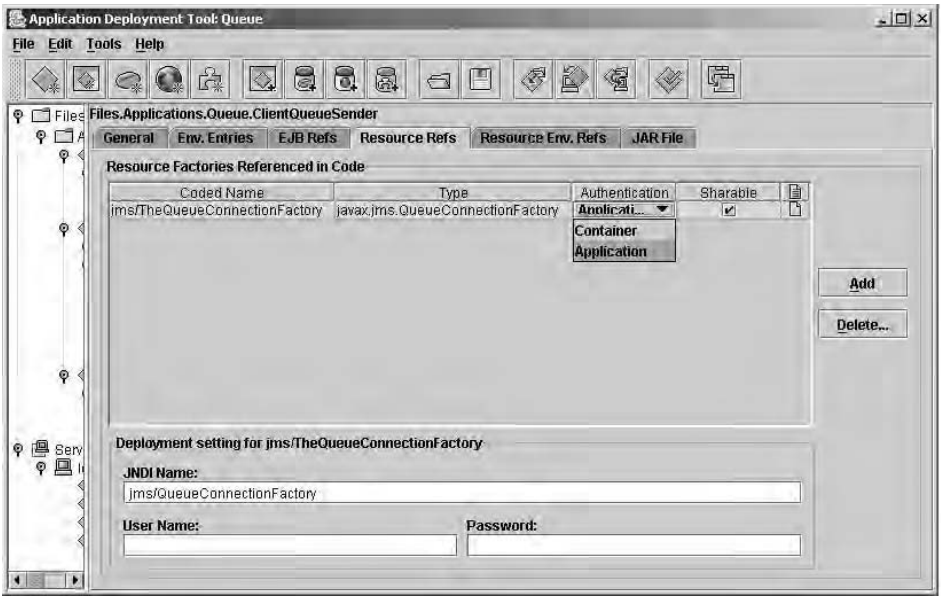


Abbildung 1.13: Authentifizierung bei einer Ressource im Deployment Tool der Referenz-Implementierung

Bei der Container-Variante werden die Benutzernamen und das Passwort direkt im Deskriptor eingegeben. Dies ist beispielsweise dann sinnvoll, wenn die J2EE-Komponente auf einen öffentlichen Bereich einer Datenbank zugreifen möchte, der aber zumindest durch einen symbolischen Namen und ein symbolisches Passwort geschützt ist. Wichtig hier ist zu wissen, dass bei der container-basierten Authentifizierung nicht die Usernamen und Kennwörter des Aufrufenden durchgereicht werden, sondern nochmals im Deskriptor eingetragen werden müssen. Damit ist eine »individuelle« Anmeldung bei externen Systemen so nicht durchzuführen!

Bei der application-basierten Authentifizierung werden jedoch der Benutzername und ein Kennwort benötigt. Doch wie bekommt eine J2EE-Komponente darauf den Zugriff?

1.4.3 Lösung

Diese Frage ist nicht trivial, da das JAAS-API genau dieses Wissen vor der Anwendung kapseln soll: Die Anwendung soll sich um Benutzer und Passwort nicht kümmern. Wird ein Application Client statt einer individuellen Anwendung eingesetzt, ist es nicht spezifiziert, wo und wie Benutzernamen und Kennwort abgelegt sind. Wichtig ist ja nur, dass der Stub diese Informationen lesen und bei entfernten Methodenaufrufen an den Server übermitteln kann. Um zumindest an den Benutzernamen heranzukommen, würde eine EnterpriseBean den EJB-Kontext mit der Methode `getCallerPrincipal()` verwenden, und so zumindest den Benutzernamen abfragen können. Die Spezifikation des Application Clients schreibt nun leider keinen korrespondierenden ClientContext vor, da kein Zwang zur Implementierung eines Interfaces besteht (leider!).

Bei der Verwendung des JAAS wird das Problem durch die Klasse

```
javax.security.auth.Subject
```

gelöst. Dieses bietet die Methoden

```
Set getPrincipals()
Set getPublicCredentials()
Set getPrivateCredentials
```

Das gültige Subject kann durch den Aufruf der statischen Methode `Subject.getSubject(AccessControlContext)` erhalten werden. Dies verlangt jedoch, dass der Codebestandteil, in dem das Subject angefordert wird, innerhalb einer `java.security.PrivilegedAction` ausgeführt wird. Genau dies ist auf dem Client nicht notwendigerweise der Fall, da die Implementierung des Application Clients die Benutzer-Informationen auch anders halten darf.

Ein Paradebeispiel für diese Aussage bietet die Referenz-Implementierung von Sun selbst:

- Als Credential wird bei der Verwendung eines Passworts die Klasse `com.sun.enterprise.security.auth.login.PasswordCredential` verwendet. Diese enthält unter anderem den Benutzernamen und das Passwort als lesbare Property.
- Die Anmelde-Informationen werden in der Klasse `com.sun.enterprise.security.ClientSecurityContext` abgelegt. Diese Klasse stellt eine Reihe von statischen Methoden zum Setzen und Lesen der Sicherheits-Informationen bereit. Insbesondere ist nur durch diese Methoden der Zugriff auf das `javax.security.auth.Subject` und damit auf die als Credentials dienenden Objekte möglich. `Subject.getSubject(…)` liefert hier stets `null`.

- Der Application Client Container der Referenz-Implementierung setzt diesen ClientSecurityContext im Rahmen der Anmeldung. `Subject.doAs(…)` wird hier nicht aufgerufen.
- Die generierte Stub-Klasse liest aus dem `ClientSecurityContext` das Subject aus und sendet das Credential vor jedem Methodenaufruf zur Prüfung zum Applikationsserver.

Wie können auch in so einem Fall die Benutzer-Informationen ausgelesen werden?

In unserer Anwendung benötigen wir auf dem Client ein Credential. Um unabhängig vom eingesetzten Applikationsserver zu bleiben, wird für passwort-basierte Anwendungen das `CredentialIF` definiert:

```
public interface CredentialIF{
 public String getUser();
 public String getPassword();
 public String getRealm();
}
```

Eine einfache konfigurierbare Factory erzeugt Implementierungen

```
public class CredentialFactory{
private static CredentialIF credential;
public static final String ID = "Credential";
public static CredentialIF create(){
 try{
 if (credential == null){
 InitialContext lContext = new InitialContext();
 String lClassName = lContext.lookup("java:comp/env/" + ID).toString();
credential = (CredentialIF) (Class.forName(lClassName).newInstance());
}
}
catch(Exception pException){
 pException.printStackTrace();
 return null;
 }
 }
}
```

Eine Lösung für die Referenz-Implementierung von Sun sieht so aus:

```
import com.sun.enterprise.security.auth.login.PasswordCredential;
import com.sun.enterprise.security.ClientSecurityContext;
class J2EEriCredential implements CredentialIF{
 private PasswordCredential cCredential;
 public J2EEriCredential(){
```

```
 cCredential =
(PasswordCredential)ClientSecurityContext.getCurrent().getSubject().getPrivateCred
entials().iterator().next();
 }
 public String getUser(){
 return cCredential!=null?cCredential.getUser():null;
 }
 public String getPassword(){
 return cCredential!=null?cCredential.getPassword():null;
 }
 public String getRealm(){
 return cCredential!=null?cCredential.getRealm():null;
 }
}
```

Die Anwendung muss die Berechtigung dafür haben, die Credentials aus dem Subject zu lesen. Dies ist für den normalen Application Client nicht notwendig und muss deshalb zusätzlich in der Berechtigungsdatei mit aufgenommen werden. Diese befindet sich bei der Referenz-Implementierung im Verzeichnis /lib/security und wird vom Batchjob »runclient« beim Start des Application Client verwendet. Alternativ dazu, und in der Praxis sicherlich sinnvoller, kann für die Anwendung auch ein angepasster Batchjob erzeugt werden, der eine andere Policy-Datei verwendet. Wichtig ist nur, dass die Anwendung die folgende Berechtigung enthält:

```
permission javax.security.auth.PrivateCredentialPermission
"com.sun.enterprise.security.auth.login.PasswordCredential
com.sun.enterprise.security.PrincipalImpl \"*\"", "read";
```

Damit kann der Queue-Sender nun endlich die Anmeldung beim JMS-Provider durchführen:

```
 public static void main(String[] args){
 try{
 ClientQueueManager lManager = new ClientQueueManager();
 CredentialIF credential = CredentialFactory.create();
 QueueConnection lConnection =
lManager.createQueueConnection(credential.getUser(), credential.getPassword());
QueueSession lSession = lConnection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
Queue lQueue = lManager.getQueue();
//…
```

Damit können Client-Anwendungen durch hochspezialisierte Implementierungen des `CredentialIF`-Interfaces, die duch die Factory im Rahmen eines Frameworks zur Verfügung gestellt werden, unabhängig von der Art der Implementierung der Anmelderoutine auf einheitliche Weise die Benutzer-Informationen auslesen.

Eine ähnliche Aufgabenstellung ist für EJB-Komponenten gegeben, da der EJB-Context ja nur den Principal direkt zur Verfügung stellt. Auch hier wird das Subject in einem eigenen Kontext gehalten (Klasse `com.sun.enterprise.security.SecurityContext`), es wird nur eine andere Implementierung des Interfaces `CredentialIF` benötigt:

```
package com.hotspots.javax.security.auth.login;
import com.sun.enterprise.security.auth.login.PasswordCredential;
import com.sun.enterprise.security.SecurityContext;
class J2EEriEJBCredential implements CredentialIF{
 private PasswordCredential cCredential;
 public J2EEriEJBCredential(){
 cCredential =
(PasswordCredential)SecurityContext.getCurrent().getSubject().getPrivateCredential
s().iterator().next();
 }
 public String getUser(){
 return cCredential!=null?cCredential.getUser():null;
 }
 public String getPassword(){
 return cCredential!=null?cCredential.getPassword():null;
 }
 public String getRealm(){
 return cCredential!=null?cCredential.getRealm():null;
 }
}
```

Zusätzlich dazu muss der Applikationsserver dieser Anwendung dann aber auch das Recht einräumen, die Credential-Informationen zu lesen. Dieser Zugriff ist normalerweise keinem EJB erlaubt und muss in der Server-Konfiguration eingetragen werden. Näheres hierzu im Abschnitt über die Konfiguration des Applikationsservers. Konkret muss für die Referenz-Implementierung von Sun ein Eintrag in die Dateien `server.policy` bzw. `client.policy` im lib/security-Unterverzeichnis hinzugefügt werden:

```
// permissions for default domain
grant {
 …

permission javax.security.auth.PrivateCredentialPermission
"javax.resource.spi.security.PasswordCredential * \"*\"", "read";
permission javax.security.auth.PrivateCredentialPermission
"javax.resource.spi.security.GenericCredential * \"*\"", "read";
//Neu:
permission javax.security.auth.PrivateCredentialPermission
"com.sun.enterprise.security.auth.login.PasswordCredential
com.sun.enterprise.security.PrincipalImpl \"*\"", "read";
};
```

Sind diese Voraussetzungen geschaffen, haben eine EnterpriseBean und auch der Application Client Zugriff auf die aktuellen Credentials und können dieses nutzen, um weitere Authentifizierungen bei Ressourcen durchzuführen.

Die hier präsentierte Lösung ist so natürlich nur für die Referenz-Implementierung zu verwenden und weiterhin sicherheitstechnisch gesehen bedenklich. Sie dient rein zum Aufzeigen der Problematik und als Ansatz zur Lösung.

1.4.4 Praxis

Wird der im vorherigen Hotspot gezeigte eigene Application Client verwendet, sind keinerlei weiteren Aktionen notwendig, da der selbst geschriebene Application Container die main-Methode der eigentlichen Anwendung mit der statischen Methode `Subject.doAs(…)` aufruft. Damit können das Subject bzw. das eigentlich interessierende Credential bei gegebener Berechtigung `javax.security.auth.AuthPermission` mit den Anweisungen

```
Subject subject = Subject.getSubject(AccessController.getContext());
cCredential =
(PasswordCredential)subject.getPrivateCredentials().iterator().next();
```

gelesen werden. `PasswordCredential` ist hier die Weblogic-Implementierung aus dem Paket `weblogic.security.auth.login`. Die Credential-Factory gibt dann nur noch, wie auch bei der Referenz-Implementierung, einen Wrapper um diese Klasse zurück.

Innerhalb des Weblogic Applikationsservers selbst funktioniert die Delegation des Credentials auf andere Art und Weise. Der vorgestellte Ansatz (Auslesen des Credentials innerhalb einer EnterpriseBean-Komponente) ist sicherheitstechnisch nicht unbedenklich: Eine »feindliche« EnterpriseBean könnte auf diese Art und Weise sämtliche Benutzernamen und Passwörter auslesen! In einer professionellen Produktionsumgebung dürfte die `AuthPermission` zum Lesen der privaten Credential-Informationen nur streng restriktiv speziellen Komponenten gegeben werden. Weblogic sieht deshalb den Zugriff auf Credential-Informationen so nicht vor. Stattdessen werden von BEA die so genannten »Credential Maps« eingeführt. Eine Credential Map ist, wie der Name bereits vermuten lässt, eine Möglichkeit, für authentifizierte Benutzer die Credentials zur Anmeldung an eine Ressource auszutauschen.

Weblogics Credential Maps werden in Zusammenhang mit der Java Connector Architecture angewandt. Für einen Weblogic User werden Benutzername und Credential für die Anmeldung an einen Connector hinterlegt. Beim Zugriff auf den Connector werden diese Informationen zur Authentifizierung beim Connector benutzt. Die Standard-Implementierung legt die Benutzernamen und Kennwörter in einen internen LDAP-Server ab. Eine eigene Implementierung kann nun aber selbstverständlich auch dasselbe Credential weiter propagieren.

1.5 Authentifizierung und Anpassung des JAAS

1.5.1 Problemstellung

Wie können anwendungsspezifische Anmelderoutinen definiert werden?

1.5.2 Technischer Hintergrund

Ein Endbenutzer-Client als »Application Client« muss selbst den Principal und das Credential erzeugen und liefern. Dies erfolgt durch das API JAAS. Dieses API bietet zum einen Schnittstellen und Klassen für den Anwendungsprogrammierer auf dem Client, um den eigentlichen Login durchzuführen. Diese sind in den Paketen `java.security, javax.security.auth` und `javax.security.login` enthalten. Daneben wird im Paket `javax.security.login.callback` das Interface `CallbackHandler` definiert, das eine angepasste Eingabe der benötigten Informationen festlegt. Dazu muss jede Implementierung die Methode

```
handle(javax.security.login.callback.Callback[] callbacks)
```

definieren. Callback ist ein reines Marker-Interface, die Klassenbibliothek enthält eine Reihe von Implementierungen, die z.B. die Ausgabe von Informationen (Klasse `TextOutputCallback`) oder die Eingabe des Benutzernamens (NameCallback) ermöglichen.

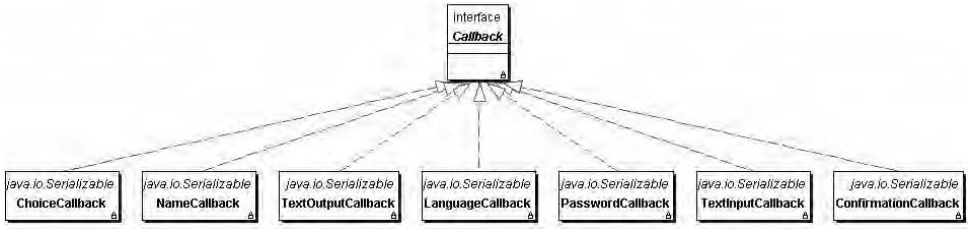


Abbildung 1.14: Die Callback-Implementierungen der Java Klassenbibliothek

Ein einfacher Callback-Handler liest Benutzername und Kennwort von der Konsole:

```
class SimpleCallbackHandler implements CallbackHandler {
public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {

for (int i = 0; i < callbacks.length; i++) {
if (callbacks[i] instanceof TextOutputCallback) {
 TextOutputCallback toc = (TextOutputCallback)callbacks[i];
 switch (toc.getMessageType()) {
 case TextOutputCallback.INFORMATION:
```

```
 System.out.println(toc.getMessage());
 break;
 case TextOutputCallback.ERROR:
 System.out.println("ERROR: " + toc.getMessage());
 break;
 case TextOutputCallback.WARNING:
 System.out.println("WARNING: " + toc.getMessage());
 break;
 default:
 throw new IOException("Unsupported type: " + toc.getMessageType());
}
} else if (callbacks[i] instanceof NameCallback) {
NameCallback nc = (NameCallback)callbacks[i];
 System.err.print(nc.getPrompt());
 System.err.flush();
 nc.setName((new BufferedReader(new InputStreamReader(System.in))).readLine());
 }
 else if (callbacks[i] instanceof PasswordCallback) {
PasswordCallback pc = (PasswordCallback)callbacks[i];
System.err.print(pc.getPrompt());
System.err.flush();
pc.setPassword(readPassword(System.in));
 } else {
throw new UnsupportedCallbackException
(callbacks[i], "Unrecognized Callback");
}
}
 }
```

Ein Callback-Handler wird von einem JAAS-Client zur Verfügung gestellt, die Methoden werden jedoch nicht direkt aufgerufen. Die zentrale Klasse des JAAS ist für den Client der `LoginContext`, dessen Implementierung den Anmeldevorgang durchführen muss. Der LoginContext ist aber selbst wiederum konfigurierbar und delegiert die Anmeldung an eine Implementierung des Interface `javax.security.auth.spi.LoginModule` weiter. Diese Konfigurationsdatei wird durch die Umgebungsvariable

```
java.security.auth.login.config
```

festgelegt. Innerhalb der Konfigurationsdatei werden in separierten Namensräumen ein oder mehrere LoginModule-Implementierungen festgelegt und mit Key-Value-Paaren konfiguriert. Die folgende Konfigurations-Datei legt unter dem Namen »default« ein LoginModule des Weblogic-Applikationsservers fest:

```
bea_default {
weblogic.security.auth.login.UsernamePasswordLoginModule required debug=true;
};
```

Während die Anmeldung innerhalb der Client-Anwendung beinahe eine Trivialität darstellt:

```
lc = new LoginContext("bea_default", new SimpleCallbackHandler());
lc.login();
```

wird transparent für den Anwendungsprogrammierer an das LoginModule delegiert:

- Der LoginContext liest aus einer Konfigurationsdatei unter dem Eintrag »bea_default« den voll qualifizierten Klassennamen einer Implementierung des LoginModule-Interfaces.
- Die Implementierung des LoginModule wird instanziert und initialisiert.
- Der Aufruf der Methode login() wird an das Login Modul delegiert. Darin wird festgelegt, welche Informationen eingegeben werden müssen, indem Callback-Objekte erzeugt werden.
- Diese werden der Methode `handle` des angegebenen Callback-Handlers übergeben.
- Danach kann diese Information geprüft werden, um schließlich als Letztes noch den Principal und Credential zu erzeugen. Diese Prüfung erfolgt in der Regel auf dem Server, kann aber natürlich auch lokal erfolgen. Principal und Credential müssen ebenfalls geeignet implementiert werden.

Ein LoginModule, das Benutzername und Kennwort benötigt, enthält beispielsweise folgende Anweisungen:

```
…
Callback[] callbacks = new Callback[2];
callbacks[0] = new NameCallback("Username: ");
callbacks[1] = new PasswordCallback("Password: ", false);
callbackHandler.handle(callbacks);
username = ((NameCallback)callbacks[0]).getName();
char[] tmpPassword = ((PasswordCallback)callbacks[1]).getPassword();
…
```

Ein einfacher Principal ohne jegliche kryptografische Kodierung hat folgendes Aussehen:

```
public class SimplePrincipal implements Principal, java.io.Serializable {
 private String cName;
 public SimplePrincipal(String pName) {
 if (pName == null){
 throw new NullPointerException("illegal null input");
}
cName = pName;
 }
 public String getName() {
```

```
return cName;
 }
//Geeignete Implementierungen von equals(), hashCode und toString()
```

Die Methoden `equals()`, `hashCode()` und `toString()` des `java.security.Principal`-Interfaces sind zwar bereits in `java.lang.Object` definiert, müssen aber natürlich in einer eigenen Implementierung geeignet überschattet werden, da Principals natürlich über Maschinengrenzen hinweg ausgetauscht werden sollen und folglich die Mimik eines entfernten Parameters aufweisen müssen.

Die folgende Tabelle gibt eine Übersicht der beteiligten Komponenten:

Implementierungen von	Bereit gestellt von	Beschreibung
Callback	JRE-Klassenbibliothek JAAS-Provider	Eingabe von Anmeldeinformationen und Ausgabe von Mitteilungen.
Callback-Handler	Client-Anwendung	Implementierung der Eingabe-Funktionalität, z. B. Konsoleneingabe oder GUI-Dialog.
LoginContext	JRE-Klassenbibliothek	Auslesen der JAAS-Konfigurationsdatei. Halten und Bereitstellen des Security-Subjects.
Principal	JAAS-Provider	Halten des Anmeldenamens.
Credential	JAAS-Provider	Halten aller Informationen, die zur Authentifizierung benötigt werden. Achtung: »Credential« ist nur ein Begriff, kein Interface!
Subject	JRE-Klassenbibliothek	Halten der Informationen des Principals und der Credentials.
LoginModule	JAAS-Provider	Erzeugen der hersteller-abhängigen Principals und Credentials. Authentifizierung beim JAAS-Provider

Tabelle 1.4: Die Komponenten einer JAAS-Anwendung

Ablauf des Login:

- Das Programm wird gestartet. Dabei werden zwei System-Properties gesetzt:
  - `java.security.auth.login.config=auth.config` bestimmt den Namen der Konfigurationsdatei für die Authentifizierung.
  - `java.security.policy=security.policy` bestimmt die Policy-Datei.
- Im Programmcode wird die Anweisung `new LoginContext(String loginContext, CallbackHandler handler)` erreicht:
  - Nun wird in `auth.config` der Eintrag für den angegebenen `loginContext` ausgelesen. Darin werden der voll qualifizierte Klassenname einer `LoginModule`-Implementierung sowie eventuell weitere Konfigurationsparameter gefunden.

  - Das `LoginModule` wird vom `LoginContext` instanziert und durch den Aufruf der `initialize()`-methode initialisiert. Dabei werden die definierten Parameter als Map sowie die Referenz auf den `CallbackHandler` übergeben.
  - Sowohl das `LoginModule` als auch der `CallbackHandler` können in verteilten Architekturen selbstverständlich auch Stubs sein.
- Nachdem mehrere `LoginModules` während der Authentifizierung beteiligt sein können, erfolgt diese in zwei Phasen:
  - Die `login()`-Methode fordert die Anwendung zur Eingabe von Benutzernamen und Kennwörtern auf.
  - Während des Commits werden alle LoginModule-Implementierungen nochmals involviert und entscheiden endgültig über den Erfolg der Authentifizierung.
- `LoginContext.login()` wird erreicht. Das `LoginModule` übernimmt die Kontrolle und versucht, die benötigten Benutzer-Informationen zu erhalten. Dazu werden
  - Instanzen von `Callback`-Implementierungen erzeugt. Diese werden in einem Array zusammengefasst und
  - an den `CallbackHandler` übergeben (Methode `handle(Callback[])`.
- Der `CallbackHandler` analysiert die übergebenen Callback-Referenzen und prüft deren Typ. Je nach Typ des Handlers werden in der `handle`-Methode die Callbacks durch Eingabeaufforderung, Auslesen von Properties oder ähnliches gefüllt.
- Das `LoginModule` erhält die gefüllten `Callback`-Referenzen und wertet die Informationen aus. Je nach Ergebnis wird `true` oder `false` zurückgegeben, das Werfen einer `LoginException` wird nur bei einem echten Fehler durchgeführt.
- Anhand von `auth.config` entscheidet der `LoginContext`, wie weiter vorgegangen wird:
  - `LoginModule.login()` liefert `true` zurück, dann wird in jedem Falle das nächste `LoginModule` angesprochen.
  - `LoginModule.login()` liefert `false` zurück, dann wird geprüft, welche Relevanz das Modul bei der Authentifizierung inne hat. Es sind insgesamt vier Möglichkeiten vorhanden:
    - REQUIRED Das `LoginModule` muss erfolgreich sein. Das nächste LoginModule wird stets aufgerufen.
    - REQUISITE Das `LoginModule` muss erfolgreich sein, sonst werden die anderen nicht mehr geprüft.
    - SUFFICIENT Falls das `LoginModule` erfolgreich ist, ist die gesamte Authentifizierung erfolgreich, sonst wird das nächste Modul ausgeführt.
    - OPTIONAL Es ist gleichgültig, ob das Modul erfolgreich war, oder nicht. Die weiteren Module werden ausgeführt.

- Nun initiiert der `LoginContext` die zweite Phase, das Commit:
  - Für alle Module wird die `commit()`-Methode aufgerufen. Diese gibt `true` oder `false` zurück. `true` signalisiert die endgültig erfolgreiche Authentifizierung, `false` einen Fehlschlag.

Für eine anwendungsspezifische Authentifizierung genügen folglich eigene Implementierungen der `CallbackHandler`-Schnittstelle, ein eigenes `LoginModule` sowie eventuell weitere `Callback`-Implementierungen.

1.5.3 Praxis

Der JBoss Applikationsserver bietet eine sauber strukturierte Security-Architektur. Im Paket `org.jboss.security.auth.spi` sind bereits die gängigsten Authentifizierungsmöglichkeiten durch `LoginModule`-Implementierungen abgedeckt:

Klasse	Beschreibung
`IdentityLoginModule`	Jeder Benutzer bekommt eine konfigurierte Identität und einen festgelegten Satz von Benutzerrollen.
`UsersRolesLoginModule`	User=Passwort-Einträge sind in der Datei users.properties eingetragen. User=Rolle1,Rolle2-Einträge sind in der Datei roles.properties eingetragen. Die beiden Dateien werden im %JBOSS_HOME%\conf-Verzeichnis gesucht.
`LdapLoginModule`	Zugriff auf einen LDAP-Server.
`DatabaseServerLoginModule`	Benutzer und Rollen werden in einer Datenbank gehalten.
`ProxyLoginModule`	Das ProxyLoginModule lädt ein anderes LoginModule von einer beliebigen Lokation.
`ClientLoginModule`	Das LoginModule für den Client setzt das Subject, das vom Client an den JBoss-Applikationsserver übertragen wird.

Tabelle 1.5: Die LoginModule-Implementierungen des JBoss-Applikationsservers

Security-Realms werden in der Konfigurationsdatei `login-config.xml` im `<policy>`-tag eingetragen.

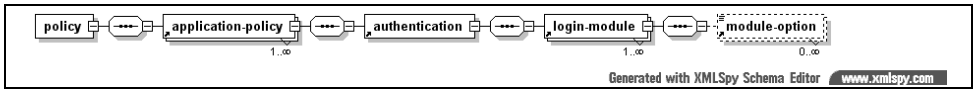


Abbildung 1.15: Die Document Type Definition für die Security-Konfiguration des JBoss-Applikationsservers

Der folgende Eintrag verwendet das `DatabaseServerLoginModule`:

```
<application-policy name = "application_client">
 <authentication>
 <login-module code = "org.jboss.security.auth.spi.DatabaseServerLoginModule"
 flag = "required">
 <module-option name = "dsJndiName">java:/SapdbDS</module-option>
 <module-option name = "debug">true</module-option>
 </login-module>
 </authentication>
</application-policy>
```

Hierbei wird eine DataSource namens »`java:/SapdbDS`« verwendet, darin werden die Standard-Tabellen PRINCIPALS und ROLES abgefragt.

Eigene `LoginModule`-Implementierungen für den JBoss erben am einfachsten von der Klasse `AbstractServerLoginModule` oder `UsernamePasswordLoginModule` und überschreiben die Methoden

- `login()getUsernamePassword`
- `getRoleSets()getIdentity()`

So nimmt das folgende `LoginModule` Kontakt mit einem RMI-Server auf:

```
public class HotspotsLoginModule extends UsernamePasswordLoginModule {
private String rmiLookup;
private RemoteUsersRoles delegate;
protected Group[] getRoleSets() throws LoginException {
SimpleGroup rolesGroup = new SimpleGroup("Roles");
ArrayList groups = new ArrayList();
groups.add(rolesGroup);
try {
String[] roles = delegate.getRoles(getUsername());
for (int i = 0; i < roles.length; i++) {
groups.add(new SimpleGroup(roles[i]));
}
} catch (Exception pException) {
pException.printStackTrace();
}
Group roleSets[] = new Group[groups.size()];
groups.toArray(roleSets);
return roleSets;
}
protected String getUsersPassword() throws LoginException {
try {
return delegate.getUserPassword(getUsername());
} catch (Exception e) {
e.printStackTrace();
return null;
}
}
```

```
public void initialize(
Subject subject,
CallbackHandler handler,
Map shared,
Map options) {
super.initialize(subject, handler, sharedState, options);
rmiLookup = (String) options.get("rmiLookup");
try {
System.out.println("initialize");
delegate = (RemoteUsersRoles) Naming.lookup(rmiLookup);
} catch (Exception pException) {
pException.printStackTrace();
throw new IllegalStateException(pException.getMessage());
}
}
}
```

Das Modul kann dann als zusätzliche Möglichkeit der Authentifizierung eingesetzt werden:

```
<application-policy name = "application_client">
 <authentication>
 <login-module code = "org.jboss.security.auth.spi.UsersRolesLoginModule"
 flag = "sufficient">
 </login-module>
 <login-module code =
"com.hotspots.javax.security.auth.spi.HotspotsLoginModule"
 flag = "sufficient">
 module-option name="rmiLookup">rmi://localhost:1100/UsersRoles
 </module-option>
 </login-module>
 </authentication>
</application-policy>
```

1.6 Single Sign-On

1.6.1 Problemstellung

Wie kann der bereits zur Anmeldung am PC verwendete Benutzername automatisch zur Authentifizierung beim Applikationsserver verwendet werden?

1.6.2 Technischer Hintergrund

Die bisher vorgestellte Vorgehensweise führte bis jetzt immer dazu, dass beim Start der J2EE-Applikation ein Anmeldedialog erscheint. Für ein »Single Sign-On« ist dies natürlich nicht geeignet. Stattdessen soll der bereits beim Betriebssystem angemeldete Benutzer als Principal verwendet werden.

1.6.3 Lösung

Es genügt hier, eine Implementierung des LoginModules zu verwenden, die direkt den Benutzernamen ausliest und diese Information an den Applikationsserver sendet. Dies kann auf einfachste Art und Weise dadurch realisiert werden, dass der Anwender beim Aufruf der Client-Applikation das Recht einräumt, die System-Property »user.name« zu lesen. Dies erfolgt bei aktiviertem Security Manager durch Vergabe der Permission

```
java.util.PropertyPermission "user.name", "read";
```

in einer Policy-Datei. Die Alternative des Anwendungsstarts ohne Security Manager ist bei verteilten Anwendungen natürlich nicht zu empfehlen. Wird die Anwendung als Application Client gestartet, wird der Security Manager automatisch gesetzt.

Zur Prüfung des Benutzers muss das `Subject`-Objekt noch die Möglichkeit liefern, den Benutzernamen zu prüfen. Dafür wird das Credential verwendet, in einfachen Anwendungen kann dafür ein String dienen.

Das Passwort ist nach erfolgreicher Authentifizierung beim Betriebssystem aus Sicherheitsgründen nicht mehr direkt abfragbar. Um eine nochmalige Abfrage eines Passworts bei der Kontaktaufnahme mit dem Applikationsserver unnötig zu machen, kann entweder auf eine nochmalige Prüfung des Passwortes verzichtet werden, oder aber der bereits angemeldete Benutzer authentifiziert sich beim Applikationsserver mit einem »Default-Passwort«.

1.6.4 Praxis

Die Implementierung des LoginModule ist sehr einfach:

```
public class UsernameLoginModule implements LoginModule {
 private Subject cSubject;
 private Map cSharedState;
 private Map cOptions;
 private boolean debug = false;
 private Principal cUserPrincipal;

 public void initialize(Subject pSubject, CallbackHandler pCallbackHandler, Map
pSharedState, Map pOptions) {cSubject = pSubject;
 cSharedState = pSharedState;
 cOptions = pOptions;
debug = "true".equalsIgnoreCase((String)cOptions.get("debug"));
 }
 public boolean login() throws LoginException {
 return true;
 }
 public boolean commit() throws LoginException {
 cUserPrincipal = PrincipalFactory.newInstance();
```

```
 if (!cSubject.getPrincipals().contains(cUserPrincipal))
 cSubject.getPrincipals().add(cUserPrincipal);
 cSubject.getPrivateCredentials().add(Credential.newInstance());
 if (debug) {
System.out.println("Added Principal to Subject");
}
 return true;
 }
 public boolean abort() throws LoginException {
return true;
 }
 public boolean logout() throws LoginException {
cSubject.getPrincipals().remove(cUserPrincipal);
return true;
 }
}
```

Nachdem die Principal-Implementierungen und das Credential jedoch für jeden Hersteller differieren, wird zur Entkopplung für beide Klassen eine Factory dazwischen geschaltet. Die Referenz-Implementierung von Sun verwendet als Credential die Klasse

```
com.sun.enterprise.security.auth.login.PasswordCredential,
```

die als Container für Benutzername, Passwort und Realm-Name dient. Die Implementierung der Credential- und Principal-Hilfsklassen lautet damit:

```
import java.security.Principal;
public class PrincipalFactory{
 private PrincipalFactory(){}
 public static Principal newInstance(){
 return new
com.sun.enterprise.security.PrincipalImpl(System.getProperty("user.name"));
 }
}
public class Credential{
 private Credential(){}
 public static Object newInstance(){
 return new
com.sun.enterprise.security.auth.login.PasswordCredential(System.getProperty("user
.name"), "user", "default");
 }
}
```

Um die Anmeldung beim Applikationsserver zu ermöglichen, genügt es, die Benutzer in der Referenzimplementierung mit einem Default-Passwort anzulegen, da die Authentifizierung ja bereits mit der Anmeldung beim Betriebssystem erfolgt ist. In obigem Beispiel wird als Passwort »user« verwendet. Der dritte Parameter ist der Name des so genannten »realms«, in dem die Benutzer- und Passwort-Informationen abge-

legt sind. Ein Applikationsserver unterstützt in der Regel zumindest zwei verschiedene Realms, eines für die einfache Verwaltung von Anmeldenamen und Passwort, das andere über Schlüssel oder Zertifikate.

Die Authentifizierung erfolgt durch Starten des Application Clients, der eine Starterklasse verwendet, sowie die benötigten Konfigurationsdateien ausliest. Soll das oben entwickelte LoginModule verwendet werden, müssen diese Konfigurationsdateien und der Batchjob angepasst werden:

```
run_userclient.bat
@echo off
%JAVA_HOME%\bin\java -Djava.security.policy=user_client.policy
-Djava.security.auth.login.config=user_clientlogin.config
-Dcom.sun.enterprise.home=%J2EE_HOME%
-classpath %CPATH%;%APPCPATH% com.sun.enterprise.appclient.Main %*
user_clientlogin.config
default {
com.hotspots.javax.security.auth.spi.UsernameLoginModule required debug=true;
};
user_client.policy
grant codeBase "file:${com.sun.enterprise.home}/lib/system/-" {
permission java.security.AllPermission;
};
grant codeBase "file:${com.sun.enterprise.home}/lib/j2ee.jar" {
permission java.security.AllPermission;
};
grant {
 permission java.util.PropertyPermission "user.name", "read";
 permission javax.security.auth.AuthPermission "modifyPrincipals";
 permission javax.security.auth.AuthPermission "modifyPrivateCredentials";
permission java.net.SocketPermission "*", "connect,accept,resolve";
permission java.net.SocketPermission "localhost:1024-", "accept,listen";
};
```

Mit diesen Einstellungen ist eine direkte Authentifizierung beim Applikationsserver ohne weitere Aufforderung zur Eingabe von Benutzername und Kennwort möglich.

Noch einfacher ist die Implementierung eines eigenen Callback-Handlers, der auf Anfrage die Informationen ohne Benutzer-Eingaben generiert:

```
class SingleSignOnCallbackHandler implements CallbackHandler {
public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
 for (int i = 0; i < callbacks.length; i++){
 System.out.println(callbacks[i]);
}
for (int i = 0; i < callbacks.length; i++) {
 if (callbacks[i] instanceof TextOutputCallback) {
 TextOutputCallback toc = (TextOutputCallback)callbacks[i];
 //Print TextOutputs
```

```
}
 } else if (callbacks[i] instanceof NameCallback) {
 NameCallback nc = (NameCallback)callbacks[i];
 nc.setName(System.getProperty("user.name"));
 }
 else if (callbacks[i] instanceof PasswordCallback) {
 PasswordCallback pc = (PasswordCallback)callbacks[i];
 pc.setPassword("useruser".toCharArray());
 } } } }
```

Eine komfortablere Möglichkeit, die bereits vorhandenen Benutzerinformationen aus dem System auszulesen, bietet das als separate Extension (jre 1.3) vorhandene bzw. seit J2SE 1.4 in die Laufzeitumgebung integrierte Paket `com.sun.security.auth.module`. Darin sind vordefinierte LoginModule-Implementierungen für das beim Download gewählte Betriebssystem (Solaris, Linux, Windows) vorhanden. Diese Klassen sind zwar nicht Teil der öffentlichen Java-APIs, werden aber ausführlich in einer separaten Dokumentation erläutert, die im guide/security/jaas/spec-Verzeichnis der J2SE-Installation zu finden ist.

Diese LoginModule-Implementierungen lesen die zur Verfügung stehenden Informationen aus. Wird beispielsweise das NTLoginModule verwendet, stehen die folgenden Informationen in der Klasse NTPrincipal zur Verfügung:

```
[NTLoginModule] succeeded importing info:
userID = User
domain = HOTSPOT_DOMAIN
domainSID = null
userSID = S-1-5-21-776746741-2136417557-3852222622-1005
primary group ID = S-1-5-21-776746741-2136417557-3852222622-513
group ID = S-1-5-21-776746741-2136417557-3852222622-513
…
impersonationToken = 1080
```

1.7 Applets als J2EE-Komponenten

1.7.1 Problemstellung

Können Applets direkt auf EJBs zugreifen?

Ist die Applet-Technologie »deprecated«?

1.7.2 Technischer Hintergrund

Als 1996 die Programmiersprache Java im Internet auftauchte, wurde sie auch auf Grund der Applet-Technologie sehr positiv aufgenommen. Die Möglichkeit, plattformunabhängige Software mittels http komfortabel und ohne zusätzliche Installationen

auf dem Client ausführen zu lassen, ist natürlich brillant einfach. Nachdem ein Java-Applet stets mit aktiviertem Security Manager läuft, ist das Risiko potenzieller Hacker-Attacken minimal. Damit ist ein Applet natürlich auch für J2EE-Anwendungen interessant. Im Komponentenmodell sind Applets auch als eigenständige Client-Komponenten spezifiziert.

In der Praxis traten jedoch sehr schnell Probleme auf:

- Die Hersteller von Browsern haben häufig Schwierigkeiten, eine korrekte und moderne Java Laufzeitumgebung zu integrieren bzw. zeigten kein Interesse an Java. So haben die Produktreihen von Netscape und insbesondere Microsoft immer wieder enorme Probleme, die aktuellen Java-Versionen vollständig zu unterstützen. Der Internet Explorer des Betriebssystems »Windows XP« wird mittlerweile in der Regel sogar komplett ohne Java Laufzeitumgebung zur Verfügung gestellt.
- Auf dem Client-PC wird häufig für alle Applet-Anwendungen eine einzige Virtuelle Maschine gestartet. Ein Applet läuft damit in einem Thread, nicht in einem eigenen Prozess. Dies kann die Größe und Komplexität der Anwendung beschränken. Insbesondere führt ein unsauberer Programmierstil selbst eines einzigen Applets zu Einbußen in der Performance des Gesamtsystems.
- Das Laden der Anwendung erfolgt über http und dauert deshalb bei größeren Anwendungen lange.
- Die Caching-Strategie der vom Web Server geladenen Klassen ist einfach: Die .class-Dateien werden im normalen Cache des Browsers gehalten. Es findet keine Versionskontrolle statt.
- Die strengen Sicherheits-Restriktionen führten schnell dazu, dass komplexere Anwendungen nur mit signierten Applets durchgeführt werden konnten. Das Signieren von Applets ist aber erheblicher Aufwand und wird vom Benutzer nicht immer akzeptiert.
- Die Java-Unterstützung kann in den Browser-Einstellungen abgeschaltet werden. Ein vorsichtiger (und häufig auch falsch informierter) Internet-Benutzer deaktiviert Java aus Sicherheitsbedenken komplett.

1.7.3 Lösung

Um diese Problematik zumindest teilweise zu entschärfen, bietet Sun die Java Laufzeitumgebung als kostenlose Plug-In-Komponente an. Die Installation des Java Plug-Ins erfolgt entweder administrativ im Rahmen des Rechner-Setups, kann aber auch beim Laden einer HTML-Seite, die ein Applet enthält, automatisch initiiert werden. Das Plug-In ist seit der Java-Sprachversion 1.3 in das normale JRE integriert, der Download des Plug-Ins installiert somit die komplette Java Laufzeitumgebung mit knapp 10 Megabyte.

Nach erfolgreicher Installation kann das Plug-In konfiguriert werden:

- Die Java Laufzeitumgebung mit Aufrufparametern für die Virtuelle Maschine,
- Proxy-Server für http, ftp, Gopher usw.,
- Cache für geladene Klassenbibliotheken,
- Zertifikate und Zertifizierungsstellen.

Damit ist die notwendige Vorbedingung geschaffen, um überhaupt an ein J2EE-Applet denken zu können. Ein simples Applet kann versuchen, eine Zeichenkette im Applikationsserver (hier die Referenz-Implementierung) zu binden und wieder auszulesen:

```
public class SimpleApplet extends JApplet{
 private JTextArea cResult = new JTextArea();
 public SimpleApplet(){
 cResult.setLineWrap(true);
 getContentPane().add(cResult);
 }
 public void start(){
 try{
 Hashtable lTable = new Hashtable(2);
 lTable.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.enterprise.naming.SerialInitContextFactory");
InitialContext lContext = new InitialContext(lTable); lContext.rebind("Test",
"Wert");
 cResult.setText(lContext.lookup("Test").toString());
 }
 catch(Exception pException){
 pException.printStackTrace();
 cResult.setText(pException.getMessage());
 }
 }
}
```

Der Versuch, den Applikationsserver aus dem Applet heraus anzusprechen, wird jedoch sofort kläglich scheitern, da die benötigten Klassen nicht gefunden werden und aus Sicherheitsgründen Aktionen wie der Aufbau beliebiger Socket-Verbindungen nicht erlaubt sind.

Dies ist natürlich nicht negativ zu bewerten! Ganz im Gegenteil zeigt sich hier die ausgeklügelte Sicherheitsarchitektur der Java Laufzeitumgebung in voller Aktion: Es wäre sicherheitstechnisch höchst bedenklich, wenn dieses Applet sofort laufen würde.

Als Erstes wird die Context-Factory-Klasse nicht gefunden. Der Klassenlader des Plug-In ignoriert die CLASSPATH-Variable komplett. Es gibt nun eine ganze Reihe von Möglichkeiten, diese Klassen zu finden:

- Dynamisches Laden vom Web Server. Das Applet kann in einem indizierten Java Archiv gehalten werden, das auch die J2EE-Klassenbibliothek enthält.
- Automatische Installation des j2ee.jar als Java Extension. Dazu muss im Manifest des Applet-Archivs die URL, von der die Enterprise Edition geladen werden kann, angegeben werden.
- Konfiguration des Plug-Ins, um eine lokal vorhandene J2EE-Installation zu benutzen.

Im Folgenden wird der Schwerpunkt auf die Konfiguration des Plug-Ins gelegt, in der der Klassenpfad ergänzt werden kann:

Abbildung 1.16: Erweiterung des Klassenpfades der Java Plug-In-Komponente

Beim Neustart des Applets wird nun zwar die Context Factory gefunden, nun greift aber die Java Sicherheitsarchitektur erneut: Die Kommunikation mit dem Applikationsserver benötigt zusätzliche Rechte. Um diese Rechte einzuräumen, sind wiederum mehrere Alternativen denkbar:

- Signieren des Applets. Wird die Signatur erkannt, bekommt das Applet die `java.security.AllPermission`. Dieses Verhalten eines signierten Applets hat natürlich weitreichende Konsequenzen: Vertraut der Benutzer einer Signatur oder einem Zertifikat, hat das Applet Vollzugriff auf das System, was sicherlich nur in Ausnahmefällen gewünscht ist.
- Signierte Applets können deaktiviert werden, indem in der Java Konfigurationsdatei die RuntimePermission »usePermission« eingetragen wird. Dann bekommen auch signierte Applets nur die Rechte eingeräumt, die in der Security-Policy eingeräumt werden. Dieses Verfahren ist administrativ zwar aufwändiger als die erste

Variante, da eine Policy-Datei gepflegt werden muss, andererseits kommt dann das feingranulierte Berechtigungs-Konzept in voller Stärke zum Tragen. Der Eintrag muss in der korrekten Laufzeitumgebung stattfinden: Das Plug-In verwendet bei der Standard-Installation unter Windows die Security-Konfigurationsdateien in c:\programme\java\j2re1.4.0\lib\security. Sinnvoller als das Editieren dieser Dateien ist natürlich auch eine Benutzer-Policy im User-Home-Verzeichnis.

- Nachdem es die Plug-In-Konfiguration ermöglicht, Laufzeitparameter zu definieren, kann auch eine spezielle Plug-In-Policy-Datei angegeben werden. Im Intranet kann diese Datei auch auf dem Web Server liegen.

Die beiden letzten Varianten sind zumindest für eine unternehmensweite Intranet-Umgebung zu empfehlen. Die erste ist durch das Einräumen der AllPermission eher als kritisch zu erachten und nur für nicht-versierte Internet-Benutzer in Betracht zu ziehen, denen eine Konfiguration des Plug-Ins nicht zuzumuten ist.

Die Konfigurations-Einstellungen für das Plug-In liegen übrigens in der einfachen editierbaren Datei `properties140` im Home\.java-Verzeichnis. Damit ist eine zentrale Verwaltung für den Administrator einer Intranet-Umgebung sehr komfortabel möglich.

Für das obige Beispiel wird eine Policy-Datei angegeben:

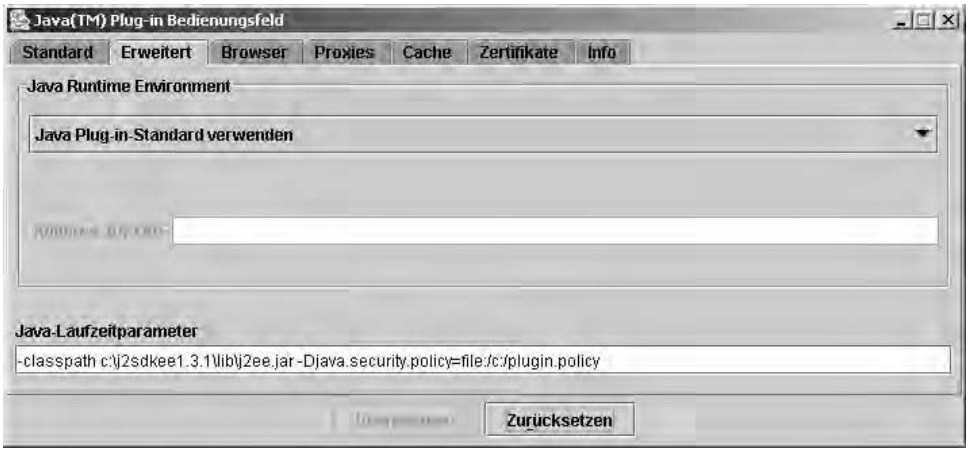


Abbildung 1.17: Angabe einer Policy-Datei für das Plug-In

Für diese Demonstrations-Anwendung genügt es, in der Policy-Datei plugin.policy die `java.security.AllPermission` zu vergeben, um das Applet im Internet-Explorer laufen zu lassen:

Abbildung 1.18: Das Applet im Internet-Explorer unter Windows XP

Bei Änderungen der Plug-In-Komponente ist darauf zu achten, dass zur Übernahme der neuen Einstellungen alle Explorer-Fenster geschlossen werden müssen. Die Virtuelle Maschine des Plug-Ins wird, genau wie bei den Office-Produkten, nur einmal pro Explorer-Anwendung gestartet, so dass keine automatische Neu-Initialisierung erfolgen kann.

Neben diesem direkten Zugriff auf den Applikationsserver können Applets natürlich auch mit einem vorgeschalteten Servlet kommunizieren. Dazu ist es nur notwendig, dass das Applet die Berechtigung hat, mit dem Web Server Kontakt aufzunehmen, auf dem sich das Servlet befindet. Dies ist dann automatisch der Fall, wenn sich Applet und Servlet auf dem gleichen Server befinden. Nach dem Öffnen der URL können dann Daten, so auch serialisierte Objekte, über Streams ausgetauscht werden. Diese Art der Kommunikation verlangt auf dem Client in der Regel keine aufwändige Konfiguration der Laufzeitumgebung, ist aber wesentlich unkomfortabler und für den Programmierer fehleranfälliger als die interface-basierte Kopplung über EJB-Stubs.

1.7.4 Praxis

Soll eine Applet als J2EE-Client verwendet werden, müssen, wie eben gezeigt wurde, mit Hilfe der Plug-In-Komponente durch Konfiguration der Java Laufzeitumgebung der Anwendung alle benötigten Rechte eingeräumt werden. Dies bedeutet in der Praxis jedoch keinen größeren Aufwand als bei der Installation einer Standalone-Anwendung, da auch hier notwendigerweise Policy-Dateien erstellt werden sollten.

Was ist nun genau zu tun, um ein Applet im J2EE-Umfeld zu nutzen? Für die Konfiguration soll von folgenden Nebenbedingungen ausgegangen werden:

- Der Web Server ist die Referenz-Implementierung von Sun auf dem Port 8000. Darauf liegt eine einfache HTML-Seite, die ein Applet enthält.

- Als anzusprechender Applikationsserver dient Weblogic auf dem Port 7001. Darin installiert ist eine EnterpriseBean, auf das das Applet zugreifen soll.

Um das Applet komfortabel konfigurieren zu können, wird eine Hilfsklasse verwendet, die die Kommunikation mit dem Applikationsserver übernimmt. Insbesondere soll hier natürlich die Anmeldung durchgeführt werden. Dies übernimmt die Klasse AppletLogin, die über die Parameter der Applet-tag-Umgebung konfiguriert wird:

```
public class AppletLogin{
//Attribute und Konstanten
 private void init(){
 try{
 String lFactory = cApplet.getParameter(JNDI_CONTEXT_FACTORY);
 if (lFactory != null){
 System.setProperty(Context.INITIAL_CONTEXT_FACTORY, lFactory);
 }
 String lUrl = cApplet.getParameter(JNDI_CONTEXT_URL);
 if (lUrl != null){
 System.setProperty(Context.PROVIDER_URL, lUrl);
 }
 System.setProperty("java.security.auth.login.config", cApplet.getCodeBase()+"/" +
cApplet.getParameter("jaas_config_file"));
LoginContext lLogin = new LoginContext(cApplet.getParameter(JAAS_CONTEXT),
CallbackHandlerFactory.getInstance().create(getCallbackHandlerClass(),
getCallbackHandlerWrapperClasses()));
 lLogin.login();
 Properties lTable = new Properties();
 lTable.put(Context.APPLET, cApplet);
 cContext = new InitialContext(lTable);
 }
//…Fehlerbehandlung
}
public Object getObject(Class pClass){
 try{
 return
javax.rmi.PortableRemoteObject.narrow(cContext.lookup(cApplet.getParameter(JNDI_LO
OKUP)), pClass);
 }
//…Fehlerbehandlung und Auslesen der Callback-Handler
}
```

Ein beliebiges Applet instanziert `AppletLogin` und kann sich dann durch den Aufruf der Methode `getObject()` eine Referenz auf das Home-Interface holen. Die zugehörige HTML-Datei definiert die Parameter, die zum Aufbau der Kommunikation gebraucht werden:

```
<applet code=com.hotspots.taschenrechner.TaschenrechnerApplet height=300
width=300>
<param name=jaas_config_file value= jaas.config>
<param name=jaas_context value= applet_bea>
```

```
<param name=jndi_context_factory value=weblogic.jndi.WLInitialContextFactory>
<param name=jndi_context_url value=t3://localhost:7001>
<param name=jndi_lookup value=Taschenrechner>
<param name=jaas_callback_handler_class
value=com.hotspots.javax.security.auth.callback.WeblogicGuiCallbackHandler>
```

Die folgenden Vorbedingungen müssen zum Start des Applets noch erfüllt sein:

- Die unten dargestellte Berechtigungs-Datei wird in diesem Beispiel zentral auf dem Web Server gehalten:

```
grant {
 permission java.lang.RuntimePermission "usePolicy";
 permission java.lang.RuntimePermission "createClassLoader";
 permission java.lang.RuntimePermission "getClassLoader";
 permission java.lang.RuntimePermission "modifyThreadGroup";
 permission java.lang.RuntimePermission "modifyThread";
 permission java.io.SerializablePermission "enableSubstitution";
 permission java.net.SocketPermission "localhost:7001", "connect, resolve";
 permission javax.security.auth.AuthPermission "createLoginContext";
 permission javax.security.auth.AuthPermission "modifyPrincipals";
 permission javax.security.auth.AuthPermission "modifyPrivateCredentials";
 permission java.util.PropertyPermission "abc.*", "read";
 permission java.util.PropertyPermission "weblogic.*", "read";
 permission java.util.PropertyPermission "java.*", "read, write";
};
```

Die hervorgehobenen Einstellungen sind für die automatische Generierung der Weblogic-Stub-Klassen notwendig und deshalb nicht für alle Applikationsserver repräsentativ.

- Das Java Plug-In muss konfiguriert werden, um mit dem Weblogic kommunizieren zu können. Als Laufzeitparameter muss angegeben werden:

```
-classpath c:\bea7\weblogic700\server\lib\weblogic.jar
-Djava.security.policy=
http://localhost:8000/taschenrechner/taschenrechner.policy
```

- Auf dem Web Server muss die JAAS-Konfigurationsdatei abgelegt werden. Die Datei heißt, wie den obigen Parametern entnommen werden kann, jaas_config und definiert darin den Kontext »applet_bea«.

```
applet_ri {
com.hotspots.javax.security.auth.login.RIClientPasswordLoginModule required
debug=true;
};
applet_bea {
weblogic.security.auth.login.UsernamePasswordLoginModule required debug=true;
};
```

- Spezielle Callback-Handler-Klassen werden ebenfalls auf dem Web Server abgelegt. In diesem Beispiel soll die Anmeldung über einen einfachen Dialog erfolgen.
- Erweiterung des Namensraums um die Lokation des weblogic.jar, das entweder lokal installiert ist oder über ftp/http geladen werden kann. Dies wird, wie bereits gezeigt, im Java Plug-In konfiguriert.

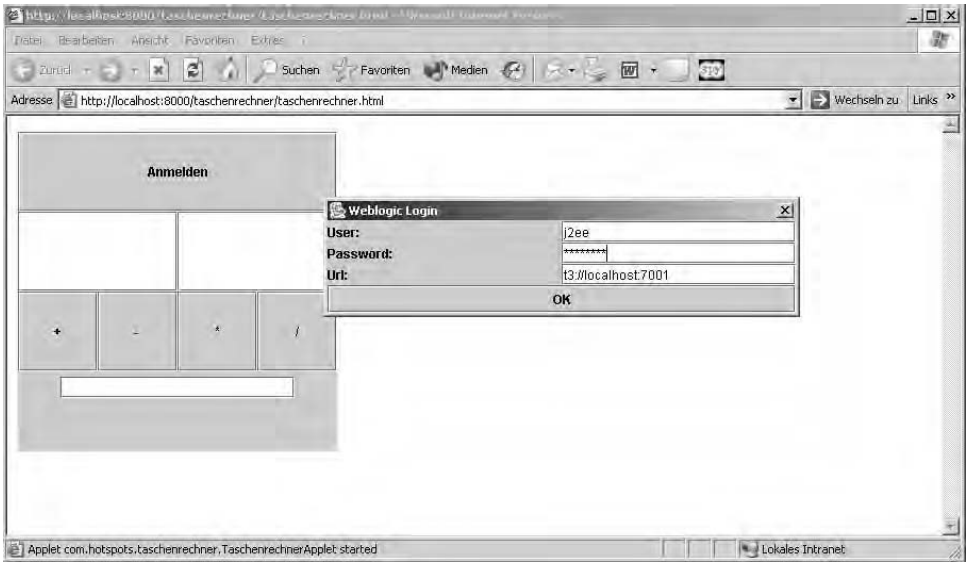


Abbildung 1.19: Das J2EE-Applet mit aktivem Anmeldedialog in Aktion

1.8 Deployment von J2EE-Clients mit Java Web Start

1.8.1 Problemstellung

Wie kann eine J2EE-Applikation auf dem Endbenutzer-System installiert werden?

Was bringt die Java WebStart-Technologie?

1.8.2 Technischer Hintergrund

Die ursprüngliche Idee der Verteilung von Java-Anwendungen über das Internet in Form von Applets hat sich durch die schlechte Umsetzung von Virtuellen Maschinen in den geläufigsten Browser-Typen nicht durchgesetzt. Erst mit der Einführung des Java Plug-Ins und der dadurch zu erwartenden höheren Konsistenz ist zu erwarten, dass die Applet-Technologie eine Renaissance erleben wird.

Es können jedoch nicht alle Verteilungsprobleme auf dem Client-Rechner mit Applets abgedeckt werden. So sollte beispielsweise ein über das Internet geladenes Textverarbeitungsprogramm auch »offline« arbeiten können. Erst für einen notwendigen oder gewünschten Update der Software sollte dann eine erneute Netzwerkverbindung aufgebaut werden müssen.

Diese Aufgabe kann natürlich durch den Download und die Ausführung eines Installationsprogramms ohne weitere Technologien durchgeführt werden, doch bringt das eine Reihe von eklatanten Nachteilen mit sich:

- Das Laden und die Installation der Anwendung ist ein nicht-trivialer Vorgang, der im Bereich des Internets manchen Benutzer überfordert.
- Die Anwendung muss im Wesentlichen selbst den Update organisieren und durchführen. Dies führt dann zu bekannten Menüpunkten wie »Nach neuen Versionen suchen …«.
- Die Installation irgendwelcher Software aus potenziell unsicheren Quellen wird vom Benutzer eher skeptisch beurteilt, weil ein Sicherheitsrisiko vermutet wird. Diese mangelnde Akzeptanz ist leider auch für Java-Programme der Fall, obwohl die Java-Sicherheitsarchitektur einen Hacker-Angriff zumindest äußerst unwahrscheinlich macht. Den wenigsten Internet-Nutzern ist bekannt, dass zum Starten einer Java-Applikation in einer kontrollierten Sandbox einfach die Option »`java.security.manager`« angegeben werden muss. Andere Nutzer vergessen diesen Aufrufparameter aus Nachlässigkeit. Es wäre wahrscheinlich wesentlich besser gewesen, wenn wie bei Applets auch eine »normale« Java-Anwendung stets mit eingeschaltetem Security Manager gestartet werden würde. Dass sich die Aufmerksamkeit im Bereich Sicherheit trotz offensichtlicher Sicherheitslücken in ganz anderen Bereichen der Computer-Software immer noch stark auf Java konzentriert, lässt sich nur durch grundlegend falsche Informationen begründen: Ein Internet-Surfer, der brav den diversen Tipps aus Computer-Zeitschriften folgend in der Konfiguration seines Browsers JavaScript und nebenbei gleich auch noch Java deaktiviert hat, installiert häufig bedenkenlos bei Bedarf Spiele, MP3-Player, Video-Betrachter und sonstige Software-Pakete.
- Die verschiedenen Installationen sind häufig unüberschaubar, da keine gemeinsame Management-Instanz existiert. Installierte Anwendungen landen »irgendwo« auf dem Rechner, der im Laufe der normalen Surfer-Aktivität im Laufe der Zeit von installierten Anwendungen geradezu überhäuft wird.

Im Unternehmen wurde diese Problematik häufig durch eine Kombination aus einem eigenen Klassenlader und einem »Bytecode-Server« gelöst. So liegen beispielsweise die Java-Archive der Anwendung in einem geschützten Bereich des Großrechners und werden über ein spezielles Netzwerkprotokoll geladen. Liegen die Archive auf einem

File oder Web Server, enthält die Klassenbibliothek bereits den `java.net.URLClassLoader`, der die Klassen aus einem URL-Array lesen kann. Soll die Anwendung auf Client-Seite nun jedoch beginnen, geladene Klassen in einem Cache zu verwalten, ist der Entwicklungsaufwand nicht unerheblich.

1.8.3 Lösung

Sun hat nun in seine Java Laufzeitumgebung das Produkt »Web Start« integriert:

Abbildung 1.20: Das »Web Start«-Produkt der Java Laufzeitumgebung

Dieses Programm enthält Features, die die oben gestellten Probleme lösen sollen:

- Dynamischer Download und Installation von Programmen entweder direkt von einem http-Server oder durch Link aus einer HTML-Seite heraus.
- Die Programme werden auf dem Client-PC zentral gecached und stehen somit auch offline zur Verfügung.
- Das Produkt kontrolliert alle geladenen Produkte, so dass auch nach längerer Zeit und größeren Mengen von installierten Web Start-Anwendungen die Übersicht gewahrt bleibt. Auch eine zentrale Update-Verwaltung ist integriert.
- Der Start einer installierten Anwendung erfolgt stets mit aktiviertem Security Manager, ein »Vergessen« dieser Option ist nicht möglich.

Mit dieser Technologie ist natürlich auch eine Installation von J2EE-Clients auf die Endbenutzer-PCs möglich. Der Ansatz ist wie bei Applets web-zentriert: Die Klassen stehen zum Download auf einem Web Server bereit. Das Laden kann entweder durch Starten der Web-Start-Konsole oder durch Klicken auf einen in einer HTML-Seite integrierten Link erfolgen.

Einstiegspunkt in eine Web-Start-Anwendung ist ein Deskriptor, der im Java Network Launching Protocol spezifiziert ist.

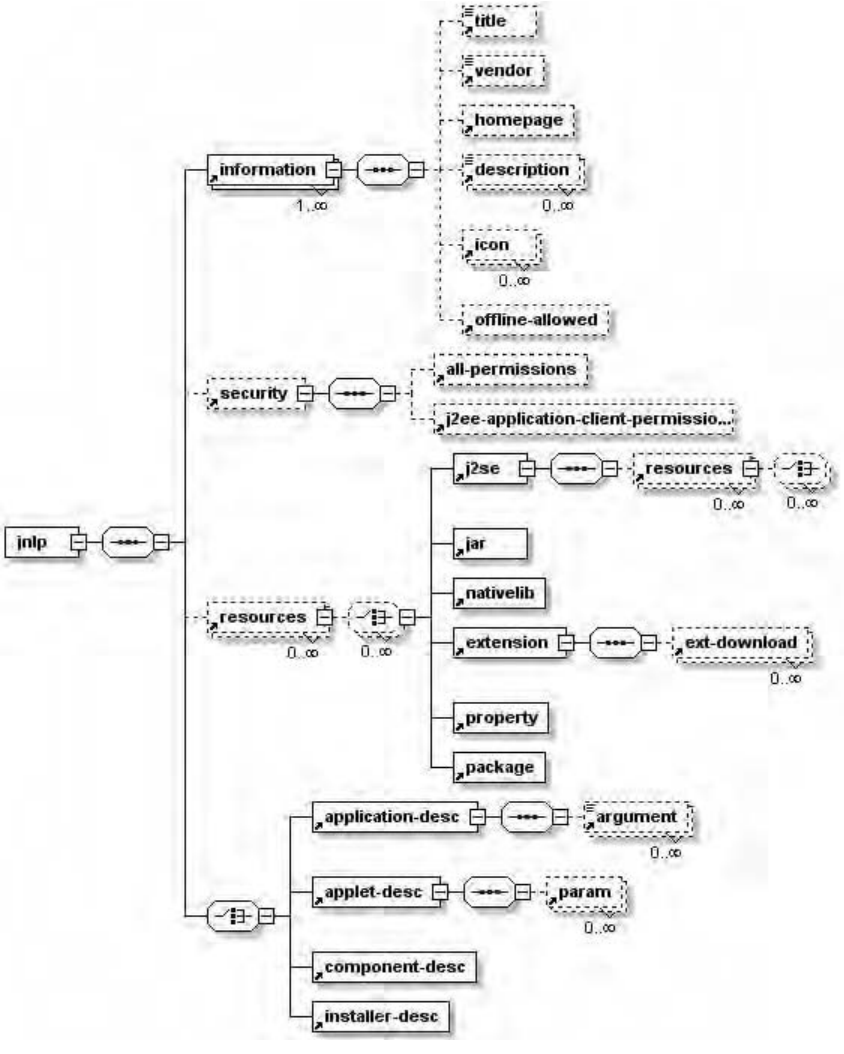


Abbildung 1.21: Die Document Type Definition für eine Web-Star- Anwendung

Um eine Java Applikation über JNLP zu verteilen, müssen die folgenden Bedingungen erfüllt sein:

- Der Web Server muss den MIME-Typ `application/x-java-jnlp-file` mit der Endung `.jnlp` verknüpfen.
- Die Applikation wird in ein Java-Archiv gepackt.
- Falls die Anwendung auf dem Client Berechtigungen benötigt, muss das Archiv signiert werden. Dann kann im `security`-tag des Deskriptors entweder der Wert `all-permissions` oder `j2ee-application-client-permission` eingetragen werden. Dieses doch recht grob granulierte Sicherheitskonzept soll in Zukunft dosierter umgesetzt werden.
- Die Anwendung wird durch einen JNLP-Deskriptor beschrieben.

1.8.4 Praxis

Der JNLP-Deskriptor der Starfinder-Clientanwendung lautet wie folgt:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- JNLP File for Starfinder Demo Application -->
<jnlp spec="1.0+" codebase="http://localhost:8000/jws">
 <information>
 <title>Starfinder Demo Application</title>
 <offline-allowed/>
 </information>
 <security> <all-permissions/></security>
 <resources>
 <j2se version="1.4+"/>
 <jar href="starfinder/starfinderClient.jar"/>
 <property name="com.hotspot.java.naming.factory.initial"
value="com.hotspots.javax.naming.InitialHashContextFactory"/>
 </resources>
 <application-desc
 main-class="com.hotspots.javax.cjb.spi.launcher.ClientLauncher">
 <argument>/ClientLauncher.properties</argument>
 <application-desc/>
</jnlp>
```

Beim ersten Aufruf des Programms erfolgt ein Sicherheitshinweis, da das Programm die `<all-permissions>`-Berechtigung anfordert (siehe Abbildung 1.22).

Nachdem der Signierer (hier: »J2EE Architekt«) seinen Schlüssel nicht von einer allgemein anerkannten Zertifizierungs-Institution gegenprüfen ließ, lautet die Empfehlung »nicht installieren«.

Abbildung 1.22: Der Sicherheitshinweis des Web Start Programms

Wird dieser Dialog trotzdem durch Betätigen der Schaltfläche »Starten« bestätigt, erfolgt der Download der Anwendungs-Klassenbibliothek in den Webstart-Cache. Anschließend wird das Programm ausgeführt.

Das Web-Start-Produkt der J2SE wird im Übrigen mit der Datei `javaws.cfg` im Installationsverzeichnis konfiguriert. Soll das Startverhalten noch speziell angepasst werden, kann darin der Eintrag

```
javaws.cfg.jre.0.path=C\:\\Programme\\Java\\j2re1.4.0\\bin\\javaw.exe
```

z.B. auf einen angepassten Batchjob deuten, in dem noch weitere Umgebunsvariablen gesetzt werden.

1.9 Ein Business Delegate pro Client

1.9.1 Problemstellung

Wie kann eine Client-Applikation ähnlich einfach wie eine HTML-Seite verwaltet werden?

1.9.2 Technischer Hintergrund

Eine rein web-basierte Anwendung benötigt einen Internet-Browser zur Darstellung der übertragenen HTML-Dateien. Eine Java Client-Anwendung benötigt die installierte Java-Laufzeitumgebung in Form der Java 2 Standard Edition. Wird nun auf die-

ser Maschine ein Programm mit Zugriff auf ein Enterprise Java Bean installiert, so muss eine Vielzahl weiterer Klassen im Namensraum der Applikation gefunden werden. Diese Klassen sind nur teilweise in der J2EE enthalten und werden benötigt, um die eigentliche Kommunikation zwischen den client-seitigen Stubs und EJB-Container aufbauen zu können. Diese Klassen sind ausschließlich typisch und verwendbar für einen bestimmten Applikationsserver und sind bei einem Wechsel des Herstellers auf allen Clients, die diese Klassen lokal verwenden müssen, zu installieren. Laufende Anwendungen müssen gestoppt werden.

Natürlich müssen die vom EJB-Komponentenentwickler definierten Home- und Komponenten-Interfaces auf den Client transferiert werden, aber damit auch

- EJBObject und EJBHome,
- die CreateException und gegebenenfalls die FinderException.

Damit muss bereits zumindest dieser Teil der J2EE installiert bzw. dynamisch geladen werden.

Problematischer sind jedoch andere Klassen, nämlich die konkreten Implementierungen des Collection-APIs, falls der Client den Zugriff beispielsweise auf die Ergebnisse von Select- und Finder-Methoden bekommt. Die gleiche Situation zeigt sich für die J2EE-Interfaces HomeHandle, Handle und EJBMetaData.

Der JNDI-Kontext des Applikationsservers wird über eine konkrete InitialContextFactory-Implementierung angesprochen, die ebenfalls geladen werden muss. Diese InitialContextFactory lädt dann dynamisch eine große Menge von weiteren Klassen nach.

Zum Ansprechen eines simplen EJB, installiert in der J2EE-Referenz-Implementierung eines Applikationsservers von Sun, werden neben den Komponenten-Interfaces mehr als 400 weitere Klassen mit einer Gesamtgröße von über einem Megabyte nachgeladen. Die absolute Majorität dieser Klassen wird von der CORBA-Implementierung gestellt. Ein ähnliches Verhalten zeigt sich natürlich auch bei BEAs Weblogic: Auch hier lädt die Implementierung der InitialContextFactory eine Menge benötigter Hilfsklassen nach, die ebenfalls im Klassenpfad des Clients gefunden werden müssen.

Dies ist ein unbefriedigender Zustand: Benötigt eine Client-Anwendung den direkten Zugriff auf den Applikationsserver, so muss der Endbenutzer-Rechner dafür aufwändig konfiguriert werden:

- Installation der applikationsserver-abhängigen Klassenbibliotheken,
- alternativ: Zugriff auf einen Bytecode-Server,
- Anpassen der Aufrufparameter: Erweiterung des Klassenpfades, Angaben zur Factory etc.,
- eventuell Einstellungen für Proxy-Server.

Weiterhin ist dies natürlich Speicher-Verschwendung: Bereits im ersten Hotspot wurde gezeigt, dass alleine die geladenen Klassenobjekte einige Megabyte an Speicher verbrauchen.

Im Gegensatz dazu ist der web-zentrierte Ansatz zwar von der Administration her brillant einfach (Eintippen einer URL, Verwaltung in Favoriten ...), jedoch beschränkt sich dann die Implementierung von Programmlogik auf dem Client auf unperformante und umständliche Skriptsprachen bzw. auf Applets oder Plug-In-Komponenten. Diese Einfachheit ist möglich, weil zum einen die plattformübergreifenden Datenformate HTML und XML definiert und zum anderen das einheitliche Netzwerkprotokoll http eingesetzt wird.

1.9.3 Lösung

Die Java-Laufzeitumgebung bietet aber durchaus genügend Services, um auch einer Client-Anwendung eine Umgebung zu bieten, die ein ähnlich einfaches Arbeiten wie in einem Browser ermöglicht. Diese Einfachheit bedeutet:

- keinerlei Änderung der Client-Aufrufparameter bei einem Wechsel oder Update des Applikationsservers,
- zentrale Verwaltung und Konfiguration der Verbindung.

Das folgende Beispiel zeigt die Implementierung eines lokalen JNDI-Kontextes, der zur einfacheren Verwendung auch als Dienst gestartet werden kann. Die Idee besteht darin, vollkommen transparent für den Client statt des direkten Zugriffs auf den JNDI-Namensraum des Applikationsservers auf eine lokale JNDI-Implementierung zu verweisen. Diese nimmt Kontakt mit einem »Remote-Context« auf, der in einer separaten virtuellen Maschine laufen soll, um damit die Entkopplung der Namensräume der Client-Applikation und der Kommunikationsklassen zum Applikationsserver zu erreichen. Als Protokoll zwischen lokaler und entfernter JNDI-Implementierung bietet sich das schlanke Java RMI an.

Die Vorteile dieses Ansatzes sind:

- Die applikationsserver-spezifischen Klassen müssen nur ein einziges Mal pro Rechner geladen werden. Damit ist ein anwendungsübergreifender Klassencache realisiert, der nach Bedarf auch einfach persistent ausgelegt werden kann.
- Geöffnete Verbindungen werden wieder verwendet, Authentifizierung und Authorisierung kann einmalig erfolgen.
- Die Konfigurationsinformationen werden ebenfalls nur ein einziges Mal abgelegt.
- Der lokale JNDI-Kontext kann als applikationsübergreifendes Clipboard genutzt werden.

- Die Schnittstelle zwischen dem ServiceLocator und der Client-Anwendung kann allgemein formuliert werden. Dazu dienen entweder serialisierte Stubs in Verbindung mit Dynamic Proxies oder aber XML-Dateien, auch in Kombination mit dem in der J2SE 1.4 eingeführten standardisierten Persistenzmechanismus für Java Beans.

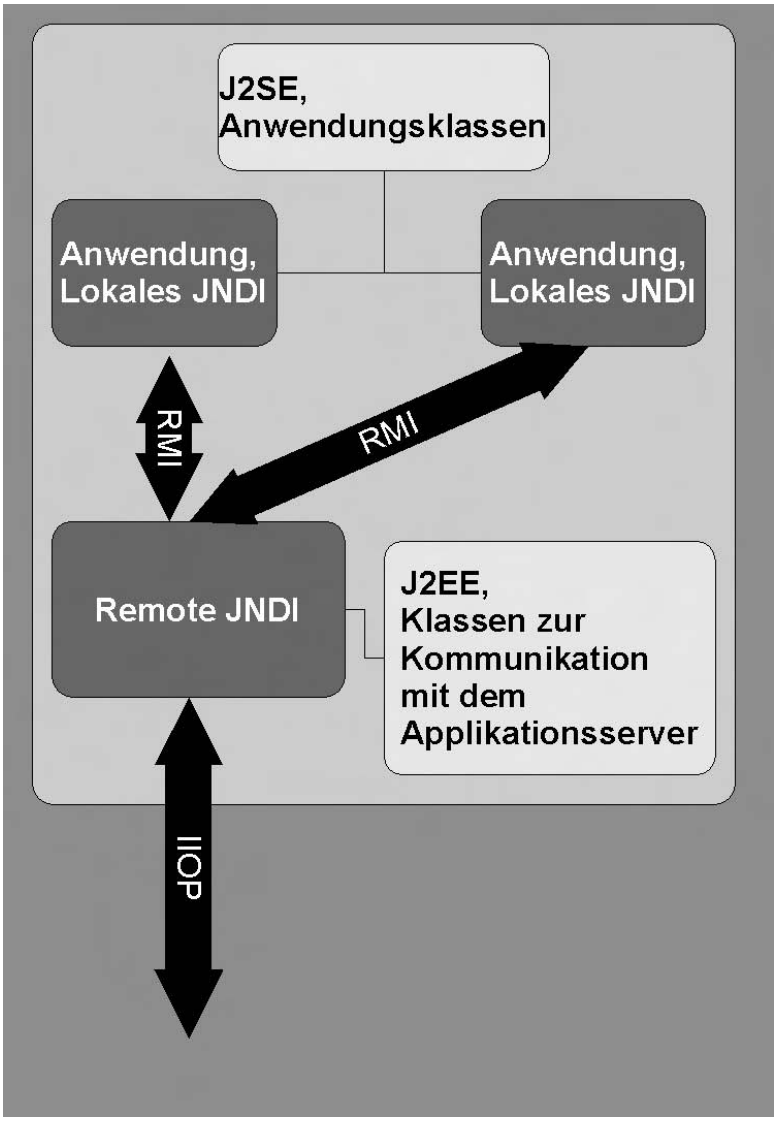


Abbildung 1.23: Zentrale Haltung aller applikationsserver-spezifischen Klassen in einem von mehreren Anwendungen gemeinsam genutzten JNDI-Kontext

Die Implementierung des Remote-Kontextes ist nicht nur ein Anbieten der Methoden des Context-Interfaces mit Hilfe von Java RMI, sondern dient gleichzeitig als Adapter zwischen den Klassen der J2SE und der J2EE bzw. den Klassenbibliotheken des Applikationsservers. Diese Anpassung ist an zumindest drei Stellen notwendig:

- Die vom Applikationsserver gesendeten Stubs dürfen natürlich nicht direkt an die Anwendung übertragen werden, da sonst nichts gewonnen wäre: Gerade die Stubs verwenden ja die speziellen Kommunikationsklassen. Bei der Referenzimplementierung von Sun wird der javax.rmi.CORBA.Stub verwendet, der intern an einen javax.rmi.CORBA.StubDelegate delegiert. Dieser Stub-Delegate wird aber vom ORBManager auf den `com.sun.corba.ee.internal.javax.rmi.CORBA.StubDelegateImpl` gesetzt und kann so nicht direkt in einer virtuellen Maschine verwendet werden, die nur die J2SE kennt.
- Die Implementierungen der EJBMetaData-, HomeHandle-, Handle-Interfaces sowie die Implementierungen des Collection-APIs sind laut EJB-Spezifikation vom Hersteller des Applikationsservers zu implementieren. Hier ist eine Entkopplung vorzusehen.
- Konsequenterweise darf der Client auch den Typ des EJBHome- und EJBObject-Interfaces nicht kennen. Diese Trennung wird in der Praxis aber nur dann möglich sein, wenn der Client die EJB-Technologie nicht verwendet und nur mit den dann vorhandenen Business-Interfaces, nicht jedoch mit den Home- und Komponenten-Interfaces arbeitet. Dies ist jedoch nicht wahrscheinlich, da beispielsweise bei der Verwendung eines SessionBeans irgendwo im Client-Code der Aufruf einer remove()-Methode erfolgen wird, die in EJBHome oder in EJBObject deklariert sind. So sind einige Interfaces und Exception-Klassen der J2EE konzeptionell der J2SE zuzuordnen und somit auf dem Client-Rechner zu installieren.

Besteht noch die Möglichkeit, den Remote-Context auf einer separaten Maschine auslagern zu können, muss auf dem Client ausschließlich ein einfaches JRE ohne jeglichen Bezug zum Applikationsserver intalliert werden. Der ServiceLocater kann dann sämtliche EJB-Stubs in einfache RMI-Stubs umsetzen, sowie eventuell aufgetretene Exceptions in Standard- oder Applikationsexceptions umwandeln. Dies ist auch für die Verwendung eines Kleingerätes in Verbindung mit der J2ME interessant, weil damit eine Anpassung der J2EE-Anwendung im Applikationsserver entfallen kann.

1.9.4 Praxis

Der RemoteContext muss aufgrund des notwendigen Werfens einer java.rmi.RemoteException nochmals nachdefiniert werden. Dies ist zwar lästig, lässt sich aber leider nicht umgehen, falls alle Methoden des JNDI-Context-Interfaces unterstützt werden müssen. Der lokale JNDI-Kontext, hier der JNDIDispatcher, referenziert den Remote-

Context über einen RMI-Stub und delegiert die Methodenaufrufe an diesen. Der RemoteContext benötigt schließlich die Referenz auf den JNDI-Kontext des Applikationsservers.

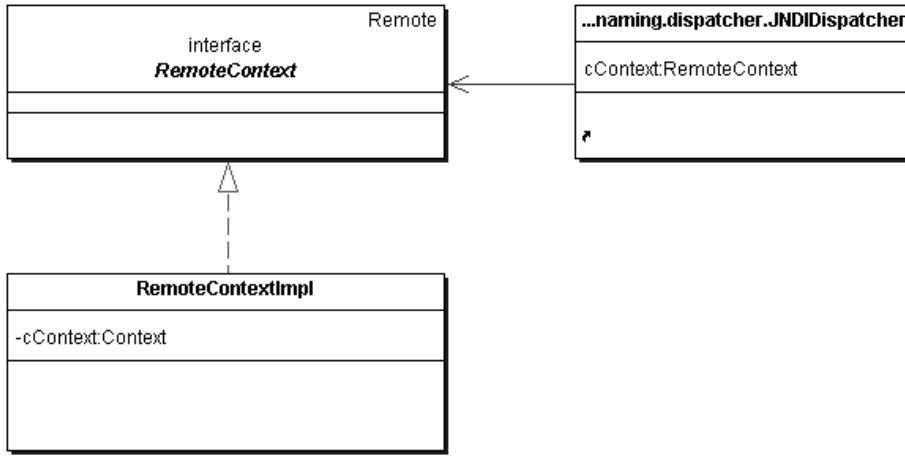


Abbildung 1.24: Lokaler und entfernter JNDI-Kontext

Nach einem Lookup liefert der Applikationsserver einen Stub, der dynamisch eine ganze Bibliothek von Kommunikationsklassen lädt. Dieser Stub darf nun keinesfalls direkt an die Client-VM geschickt werden. Stattdessen wird ein Dynamic Proxy verwendet, der jetzt aber als einfacher RMI-Stub ausgebildet werden kann. Somit liegen alle benötigten Klassen in der Bibliothek der J2SE!

Der zugehörige InvocationHandler, hier der StubInvocationHandler, kann allgemein definiert werden, so dass der Client nur einen einmaligen Satz von wenigen Klassen installieren muss. Dazu gehört als letzte Klasse noch der RemoteInvocationHandler: Der StubInvocationHandler delegiert die Methodenaufrufe einfach an den Stub dieser Klassen weiter.

Die vom JNDI-Service gelieferten Stubs sowie die Rückgabetypen sämtlicher Methodenaufrufe müssen in »ungefährliche« Objekte umgewandelt werden. Dazu dient ein StubAdapter:

```
RemoteInvocationHandler lRemoteHandler = new RemoteInvocationHandlerImpl(pObject);
Remote lRemote = UnicastRemoteObject.exportObject(lRemoteHandler);
StubInvocationHandler lHandler = new
StubInvocationHandler((RemoteInvocationHandler)lRemote);
Object lProxy = Proxy.newProxyInstance(StubAdapter.class.getClassLoader(),
pObject.getClass().getInterfaces(), lHandler);
return lProxy;
```

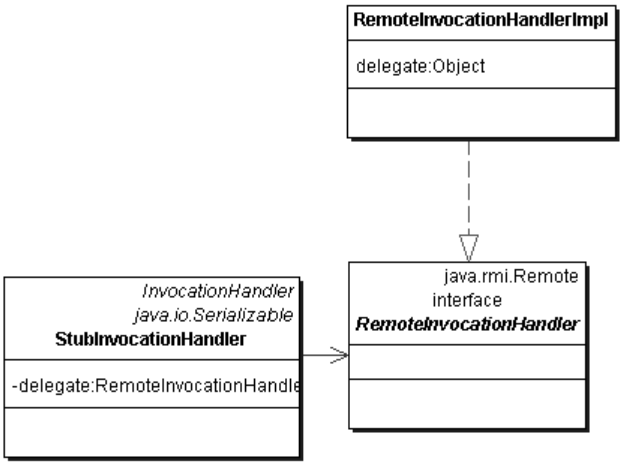


Abbildung 1.25: Die beteiligten InvocationHandler

Diese Umwandlung ist aber nicht immer notwendig bzw. korrekt, es ist zu unterscheiden:

- Normale serialisierbare Klassen. Diese werden ohne weitere Umwandlung zurückgegeben. Dies verlangt natürlich, dass eine eventuell eigens definierte Klasse keinerlei applikationsserver-typischen Klassen verwendet.
- java.rmi.Remote: Dies ist der Fall wenn ein Methodenaufruf wie create() wiederum einen Stub zurückliefert. Genau hier muss der der Wrapper eingreifen, einen RemoteInvocationHandler instanzieren und einen Dynamic Proxy generieren. Zurückgegeben wird in diesem Fall ein serialisierter StubInvocationHandler.
- Arrays: Diese können ohne Umwandlung zurückgegeben werden, falls der Typ des Arrays nicht erneut java.rmi.Remote ist. In letzterem Fall muss ein Array von StubInvocationHandlern erzeugt und zurückgegeben werden.
- Implementierungen der EJBMetaData, HomeHandle und Home-Interfaces: Diese Interfaces des Paketes javax.ejb werden zwar vom Aplikationsserver-Hersteller implementiert, erweitern jedoch nicht `java.rmi.Remote`, sondern sind serialisierbar. Trotzdem enthalten diese Klassen in jedem Falle konkrete Bezüge zum Applikationsserver und dürfen deshalb nicht zum Client transferiert werden. Während die EJBMetaData-Implementierung eine nur flüchtige Information darstellt, können die persistenten Handles in einem Repository des RMI-Servers abgelegt werden. Dies ist bei dem hier vorgestellten transienten RMI-Service nicht der Fall, Handles sind nach dem Stoppen der RMI-Registry ungültig.
- Als Letztes müssen noch die Implementierungen des Collection-APIs untersucht werden. Finder-Methoden liefern als Rückgabetyp eine Applikationsserver-abhän-

gige Implementierung des Collection- oder Set-Interfaces. Die Implementierung für die Referenz-Implementierung von Sun enthält intern eine einfache ArrayList, die die Stubs auf die EnterpriseBeans hält.

Nach diesen Umformungen ist das Ziel erreicht: Eine Client-Anwendung muss nur noch wenige standardisierte Klassen einladen, um mit einem beliebigen Applikationsserver kommunizieren zu können. Die benötigten Klassen sind im Einzelnen:

```
com.hotspots.javax.naming.dispatcher.JNDIDispatcher,
com.hotspots.javax.naming.dispatcher.JNDIDispatcherFactory,
com.hotspots.javax.naming.remote.RemoteContext,
com.hotspots.javax.naming.remote.RemoteContext_Stub,
com.hotspots.javax.naming.remote.RemoteInvocationHandler,
com.hotspots.javax.naming.remote.RemoteInvocationHandler_Stub,
com.hotspots.javax.naming.remote.StubInvocationHandler.
```

Dazu kommen dann in der konkreten Anwendung nur noch die rein koppelnden Erweiterungen von EJBObject und EJBHome. Die vom Applikationsserver generierten Stub-Klassen und damit die gesamten Kommunikationsklassen werden nicht mehr für jede Anwendung benötigt, sondern werden nur noch einmalig beim Starten des Remote-Dienstes geladen.

1.10 J2EE als Vorbild für ein Client-API

1.10.1 Problemstellung

Sind JavaBeans eigentlich »ClientBeans«?

Wie kann das deklarative J2EE-Programmiermodell bei der Erstellung von GUI-Anwendungen eingesetzt werden?

1.10.2 Technischer Hintergrund

Das J2EE-API definiert ein durchgehendes Komponentenmodell für Geschäftsobjekte. Die einzelnen Schichten (Client, Präsentation, Geschäftslogik und Integration) sind über die Schnittstellen des JNDI-API lose gekoppelt. Die Kommunikation der die jeweilige Aufgabe repräsentierenden Komponenten (Servlet, EJB, Connector) erfolgt nicht direkt, Anfragen werden stets von einem Container kontrolliert an den Empfänger delegiert. Die Kopplung erfolgt entweder über ein standardisiertes API der Java-Klassenbibliothek (Servlets, JSP, Connector) oder über die Home- und Component-Interfaces des Enterprise JavaBeans-Komponentenmodells.

Die J2EE endet aber auf einem java-basierten Client in einer grob gegliederten Einheit: Der J2EE Application Client ist eine komplette Anwendung ausgeprägt als Applet

oder Standalone-Anwendung und damit im Vergleich zu Web- oder EJB-Komponenten, in denen sauber standardisierte Schnittstellen definiert sind, äußerst grob granuliert. Der Application Container definiert im Wesentlichen nur die Anbindung an den Applikationsserver mit Anmeldung (JAAS) sowie den konfigurierbaren Zugriff auf das Enterprise JavaBean und Umgebungseinträgen.

Eine, häufig grafisch-orientierte, Benutzeroberfläche wird mit Hilfe der Bibliotheken der J2SE bzw. J2ME entwickelt. Die Virtuelle Maschine und die Klassenbibliothek definieren außer der mächtigen Sicherheitsarchitektur keinerlei standardisierte Infrastruktur, die mit einem Applikationsserver vergleichbar ist. Es existiert kein »Client-Container« mit standardisiert nutzbaren Services. Die J2SE enthält zwar mit dem Paket `java.beans.beancontext` sowie der `javax.infobus`-Erweiterung bereits eine Container-Component-Architektur für JavaBeans. Diese basiert jedoch schwergewichtig auf der JavaBeans-Spezifikation, und kann deshalb beispielsweise nicht einfach auf andere Systeme wie Kleingeräte und Embedded Systems portiert werden. Weiterhin fokussiert die Infobus-Architektur auf den dynamischen Austausch von Daten zwischen JavaBeans und weniger auf die hier interessierende Kopplung von funktionellen Objekten.

Bei der Entwicklung einer GUI-Anwendung zeigt sich für den Programmierer, hauptsächlich bedingt durch die historische Entwicklung der Programmiersprache Java, ein deutlicher Wildwuchs an Möglichkeiten und Programmierstrategien. So ist beispielsweise die Konfiguration einer Anwendung möglich mit Umgebungsvariablen, Programm-Startparametern, java.util.Properties, java.util.prefs.Preferences in Kombinationen mit eigenen Strategien. Die Austauschbarkeit der Komponenten bleibt somit beschränkt, eine Änderung der verwendeten Technologie bedingt in der Regel eine Anpassung und Neukompilierung der Anwendung.

Eine J2EE-Komponente wird dagegen in einer wohldefinierten Deskriptor-Datei konfiguriert und ist in einen Container installiert. Der Zugriff auf diese Informationen erfolgt stets gekapselt über JNDI-Lookups.

Die logische Kopplung einer grafischen Oberfläche an den Controller und das Datenmodell ist ebenfalls nicht standardisiert. Das MVC-Modell wird in verschiedenen Projekten unterschiedlich realisiert, so dass eine Wiederverwendung faktisch ausgeschlossen werden kann.

Es ist natürlich nicht sinnvoll, obwohl prinzipiell möglich, die Bestandteile einer grafischen Oberfläche wie z. B. Schaltflächen als eine Enterprise JavaBean in einen Applikationsserver zu installieren. Grafische Anwendungen werden meistens mit mausgesteuerten Drag&Drop-Werkzeugen erstellt. Im generierten Quellcode werden die verwendeten visuellen JavaBeans in der Regel direkt mit »new« instanziert, so dass eine Umstellung auf Home-Interfaces und Factories mit aufwändiger Codebearbeitung per Hand erfolgen muss. Die Umgebungen eines Endbenutzer-Systems und eines Servers

unterscheiden sich weiterhin so gravierend in Bezug auf Speicherverwaltung, Antwortverhalten etc., dass ein blindes Übertragen von Programmierstrategien zu unbefriedigenden Ergebnissen führen wird. Das Paket javax.ejb führt die Typen EnterpriseBean, EJBHome und EJBObject ein. Die Delegation vom EJBObject (dem Container) zur Enterprise JavaBean (dem eigentlichen Geschäftsobjekt) ermöglicht dem Applikationsserver-Hersteller die Implementierung von Services wie der Transaktionssteuerung und der Autorisierung. Diese Schicht mit zusätzlicher Funktionalität ist nicht für alle Client-Anwendungen notwendig. Ist eine Umhüllung einer Client-Komponente notwendig, kann bei Bedarf ein Wrapper zur Verfügung gestellt werden.

1.10.3 Lösung

Damit stellt sich nun die Frage, welche Ideen des J2EE-Programmiermodells auch für den Client-Programmierer sinnvoll übernommen werden können. Die folgende Aufstellung dient als erster Ansatz zur Definition eines Modells für client-seitige Komponenten.

Für diese wird im Folgenden der neu eingeführte Begriff »ClientBean« verwendet. Eine ClientBean ist nicht mit einer JavaBean zu verwechseln, sondern ist eher mit den Bestandteilen des traditionellen MVC-Patterns zu identifizieren.

- Die Meilensteine im Lebenszyklus jeder Komponente werden im Interface »ClientBean« deklariert.
- Die Erzeugung der client-seitigen Komponenten erfolgt stets indirekt über eine Factory, die wie in der J2EE als Home-Interface bezeichnet wird.
- Komponenten-Interfaces müssen nicht für jede Anwendung eigens definiert werden. Ein Client-Programm kann als Zusammenspiel von Destinations und Producern hoch abstrahiert werden.
- Die Komponenten einer Client-Anwendung werden über einen Client Container kontrolliert, erzeugt und verwaltet. Der Container dient damit als Factory.
- Eine Umhüllung eines ClientBeans mit Wrappern kann deklarativ erfolgen und wird ebenfalls vom Client Container durchgeführt.
- Jede Anwendung besteht aus Komponenten, die in einen Client Container installiert werden müssen. Der Container kontrolliert den Lebenszyklus der Komponenten und stellt einen ClientContext zur Verfügung.
- Der Client Container stellt einen JNDI-konformen `javax.naming.Context` zur Verfügung. Darin sind Konfigurationseinstellungen, ResourceBundles usw. abgelegt. Der Client kann den Kontext aber auch als Zwischenspeicher nutzen. Insbesondere

werden die Referenzen auf die Home-Interfaces in diesem Namensraum abgelegt. Wie im Applikationsserver werden Komponenten ausschließlich durch diesen Namensraum konfiguriert.

- Die Kopplung von ClientBeans erfolgt ausschließlich unter Verwendung von Home-Interfaces.

Begriffe und Strategien der J2EE werden so auf die Client-Umgebung umgesetzt. »Client-Umgebung« bedeutet, dass der Fokus auf das standardisierte Umsetzen von Events nach Aktionen sowie die allgemeine Definition der beteiligten Komponenten liegt. Während die J2EE in der Version 1.3 als Enterprise Bean-Typen Sessions, Entities und Message-Listener definiert, wird auf Client-Seite exemplarisch der Typ ClientBean definiert.

Auf der Client-Seite können Anwendungen sehr häufig anhand des MVC-Patterns extrem allgemein formuliert werden: Ein Controller empfängt ein Aktionskommando und Parameter und initiiert eine Zustandsänderung in einem Datenmodell. Das Datenmodell wiederum stellt den geänderten Zustand einer View zur Verfügung. Das bedeutet aber, dass bei der Adaptierung des J2EE-Modells auf dem Client die Notwendigkeit für die Definition eigener Komponenten-Interfaces wegfällt: Ein ParameterProducer ist in der Lage, auf Anfrage Parameter zu liefern. Diese Parameter werden der ParameterDestination zur Verarbeitung übermittelt. Ein ResultProducer erzeugt Ergebnisse, die von einer ResultDestination empfangen und verwendet werden können.

Im Gegensatz zu den hoch-spezialisierten Zustandsänderungen der EnterpriseBeans, die durch eigens definierte Komponenten-Interfaces definiert werden müssen, kann eine MVC-Anwendung allgemein gehalten werden.

Alle aufgeführten Interfaces werden als ClientBeans aufgefasst, um in einem ClientServer installiert werden zu können. Diese vier Typen definieren die Schnittstellen zwischen dem Controller und dem Datenmodell. Der Controller selbst wird unten als weiterer ClientBean-Typ eingeführt.

Wie in der J2EE-Spezifikation definieren diese Interfaces den Lebenszyklus der implementierenden Bean. Ein ClientBean wird ausschließlich innerhalb eines Containers genutzt und implementiert folglich weder java.io.Serializable noch java.rmi.Remote. Auch auf die Implementierung des javax.naming.Referencable-Schnittstelle wird bewusst verzichtet, da das ClientBean-API möglichst schlank bleiben und umgebungsunabhängig eingesetzt werden soll. So kennt beispielsweise das populäre Mobile Information Device Profile der Java 2 Micro Edition keine der eben aufgezählten Schnittstellen. Die Implementierung des Context-Providers muss diese Besonderheit berücksichtigen.

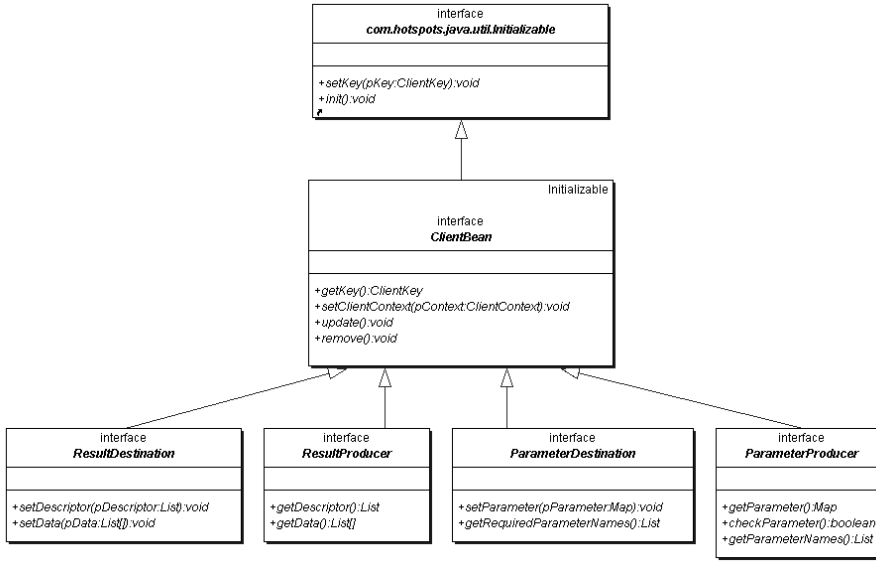


Abbildung 1.26: Die Architektur der ClientBeans

Im Rahmen des Lebenszyklus wird für jede ClientBean der ClientContext gesetzt:

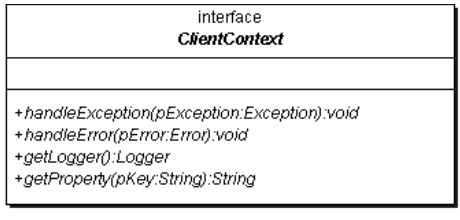


Abbildung 1.27: Der ClientContext

Dieser übernimmt hier zentrale Aufgaben, z.B. die Fehlerbehandlung und stellt einen Logger sowie den Zugriff auf Konfigurationsdaten zur Verfügung. Die Implementierung dieses Interfaces übernimmt der Hersteller des Client Containers.

ClientBeans können identifiziert werden. Dazu muss der Client Container im Rahmen der Instanzierung der Bean eine Instanz der Klasse ClientKey übergeben.

Der ClientKey besitzt zwei Konstruktoren: Der erste definiert einen Root-Eintrag, der zweite übernimmt sowohl einen Eintrag, als auch einen weiteren ClientKey. Damit kann eine verkettete Liste von ClientKeys aufgebaut werden, der Root-Key ist allen Teilen einer Hierarchie gemeinsam. Auf diese Art und Weise sollen logisch zusammengehörende Komponenten gruppiert werden können. Dies ist bei ClientBeans-Anwendungen notwendig, da pro Applikation exakt ein ClientContainer gestartet werden

soll, der dann mehrere Anwendungszweige gemeinsam kontrolliert. Mit dem ClientKey ist es möglich, dass ein Controller einen ParameterProducer (z. B. ein Eingabefeld) mit der zugehörigen ResultDestination (z. B. einer Tabelle) verknüpft.

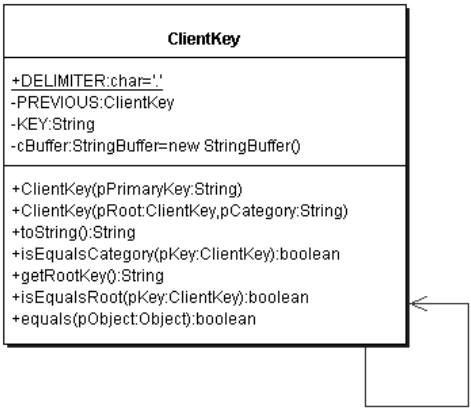


Abbildung 1.28: Der ClientKey

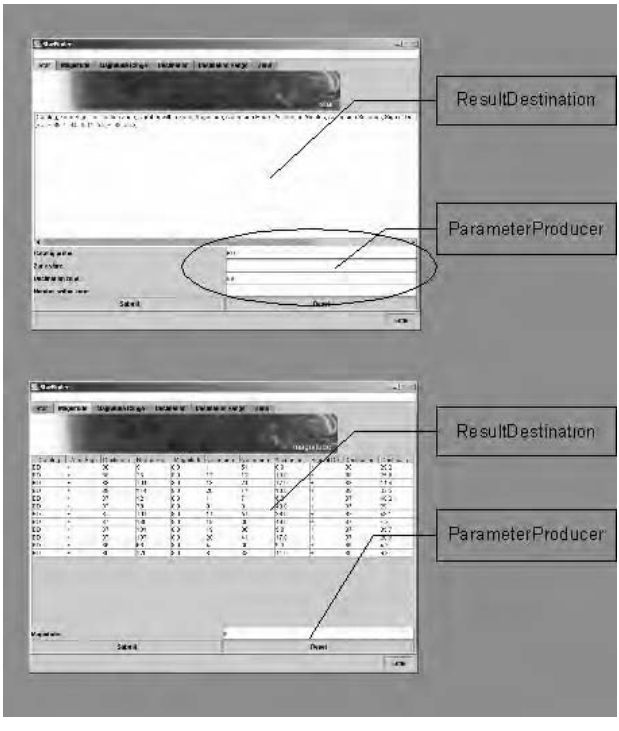


Abbildung 1.29: Kopplung von verschiedenen ParameterProducern und ResultDestinations über eine ClientKey-Hierarchie

Die Home-Interfaces der ClientBeans sind wie in der J2EE-Spezifikation nicht allgemein formulierbar, sondern können nur spezifiziert werden. Ein CJBHome-Interface muss eine Methode

```
create(ClientKey) throws CreateException
```

und/oder

```
findByKey(ClientKey) throws FinderException
```

deklarieren.

Controller und visuelle Komponenten sind ebenfalls ClientBeans. Während ein Controller noch allgemein formuliert werden kann,

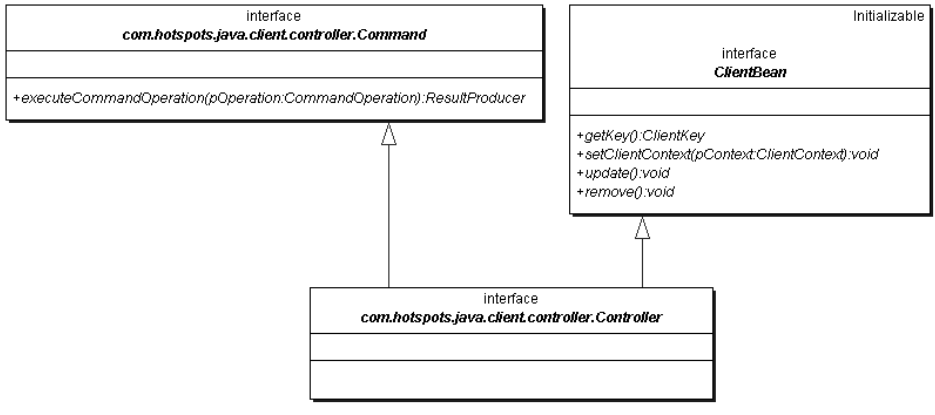


Abbildung 1.30: Der Controller

sind die verwendeten visuellen Komponenten so speziell, dass hierfür nur die Implementierung des ClientBean-Interfaces gefordert wird. So muss beispielsweise ein Panel mit Eingabefeldern das Interface `com.hotspots.javax.cjb.ParameterProducer` implementieren, eine JTable wird zur com.hotspots.javax.cjb.ResultDestination.

Die vom Controller benötigte CommandOperation kann von der visuellen Komponente nicht direkt geliefert werden. Die Umsetzung von Events nach CommandOperations übernimmt ein Event-Adapter, die Delegation zum eigentlichen Controller führt der Dispatcher durch (siehe Abbildung 1.31).

Die Controller der ClientBean-Architektur mit dem Command-Interface übernehmen die Rolle der Servlets innerhalb einer J2EE-Anwendung. Historisch korrekt ist zwar die andere Reihenfolge (Das J2EE-Servlet ist aus dem MVC-Controller entstanden.), es geht hier aber um die Idee des Komponentenmodells. Ein Servlet liest aus dem der

`service()`-Methode übergebenen Parameter vom Typ HttpServletRequest über http übermittelten Parameter und ruft die verarbeitende Geschäftslogik auf. Im ClientBean-Modell kann eine Referenz vom Typ CommandOperation übergeben werden, so dass der zugehörige ParameterProducer direkt ausgelesen werden kann. (Methode `getParameterProducer()`). Weiterhin wird ein String als Kommando-Argument erwartet. Damit kann der Controller die aufzurufende Geschäftslogik auswählen. Dies entspricht dem J2EE-Pattern des Front Controllers.

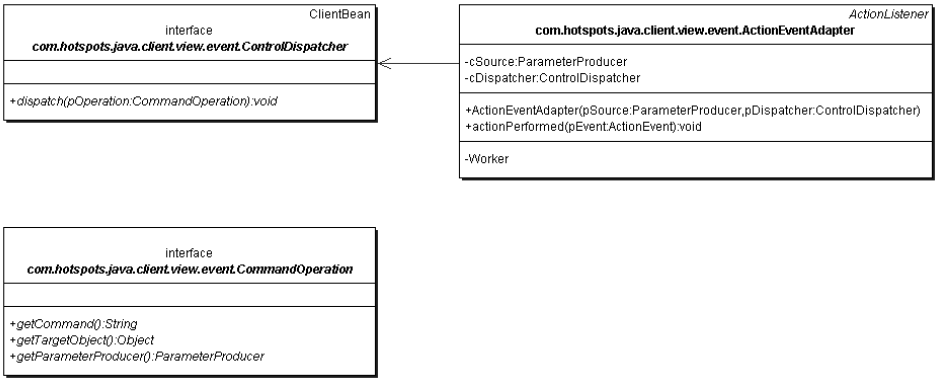


Abbildung 1.31: ActionEventAdapter und ControlDispatcher

Der Controller »kennt« die darstellende Komponente nicht direkt, sondern analysiert nur den enthaltenen ClientKey. Die darin enthaltene Information wird dazu verwendet, die zugehörige ResultDestination zu finden, die den übergebenen ResultProducer ausliest. Dieses Auslesen ist wiederum durch direkte Methodenaufrufe möglich. Somit muss kein Ausgabestrom definiert werden, der dem HttpServletResponse-Objekt entspricht. Die ResultDestination entspricht einer JSP, mit dem ResultProducer ist ein einfaches Interface für den View Helper definiert.

Auch die ClientBean-Architektur schreibt zwingend einen Container vor. Dieser übernimmt die Erzeugung der ClientBeans und kontrolliert deren Lebenszyklus. Genau so wie in der J2EE soll die Instanzierung nicht direkt erfolgen, sondern über Home-Interfaces erfolgen. Das ClientBean-API definiert als zu implementierende Signatur das Interface CJBHomeDelegate, das zur eigentlichen Objekterzeugung verwendet werden kann:

Wie ein EnterpriseBean besteht auch ein konkretes ClientBean aus mehreren Teilen:

- einem Home-Interface,
- einem Komponenten-Interface,
- der das ClientBean-Interface implementierenden Bean-Klasse und
- einem Deskriptor.

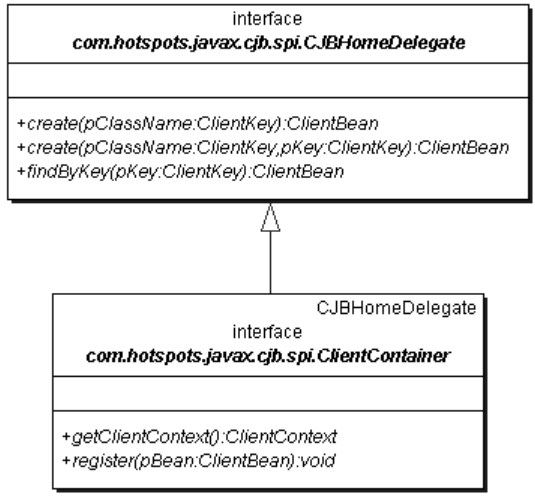


Abbildung 1.32: Das Service Provider Interface der ClientBeans

Einfache Anwendungen werden mit den standardisierten ClientBeans ParameterProducer, ParameterDestination, ResultProducer und ResultDestination auskommen. Ein zusätzliches Komponenten-Interface wird im Unterschied zur J2EE-Spezifikation direkt vom ClientBean implementiert, da auf dem Client keine Services wie Security und Transaktionssteuerung implementiert werden müssen. Die Spezifikation schreibt jedoch analog den Servlet-Filtern vor, dass jedes ClientBean von einer beliebigen Menge von Wrappern umhüllt werden kann. Dies ist durch die Interface-Definition mit einfachen Wrapper-Definitionen möglich.

1.10.4 Praxis

Nachdem keine kommerziellen CJB-konformen ClientContainer angeboten werden, ist für eine Referenz-Implementierung des Client Containers zu sorgen. Dies ist jedoch recht einfach: Der Container liest eine Properties-Datei aus, um den übergebenen Schlüssel als Klassennamen zu interpretieren und damit reflektiv Objekte erzeugen zu können.

Der Default Client Container ist selbst wiederum konfigurierbar, damit vorhandene Factories und PropertyManager wieder verwendet werden können.

Der Client Container schließlich registriert ClientBean-Komponenten und kontrolliert den Lebenszyklus. Beim Hochfahren des Containers werden noch die Konfigurationsdateien ausgelesen und damit beispielsweise der JNDI-Kontext mit Referenzen auf die Home-Implementierungen abgelegt.

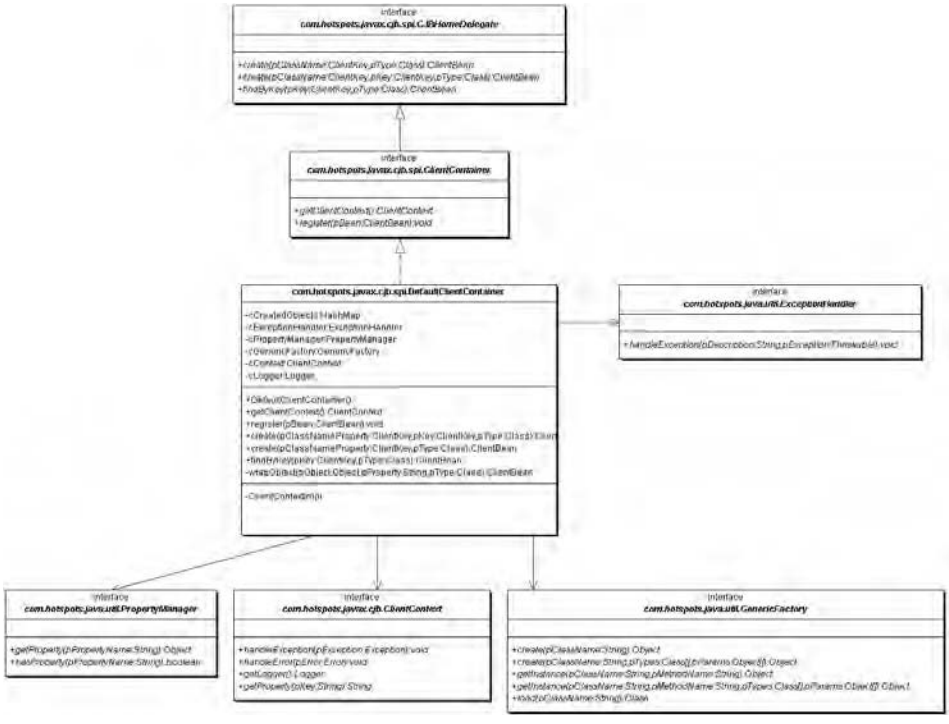


Abbildung 1.33: Der Default Client Container

Der Client Container kann alle erzeugten ClientBeanKomponenten mit beliebigen Wrappern dynamisch umhüllen. Die Definition der Wrapper erfolgt ausschließlich deskriptiv.

Ebenso liegt es in der Veranwortung des Client Container-Herstellers, eine Implementierung des Client Context als auch den JNDI-konformen Namensraum anzubieten. Auf dem Client genügt hier im Wesentlichen eine einfache Hashtable, die über einen Proxy als Kontext angeboten wird (siehe Abbildung 1.34).

Die Erzeugung einer Home-Interface Implementierung kann bei bekanntem Container automatisiert erfolgen. Hier erfolgt die Erzeugung per Hand. Die Java-Klassen eines ClientBeans sehen z.B. für ein Datenmodell, das den lesenden Zugriff auf eine Datei ermöglicht, folgendermaßen aus (siehe Abbildung 1.31).

Die Implementierung des Home-Interfaces liefert noch konfigurierbar eine Instanz des FileViewerModels zurück.

Zur Vollständigkeit definiert das SPI noch Implementierungen für eine eigene ThreadGroup sowie den Exception-Handler des `java.awt.Event-Dispatcher-Threads`, damit der Client Container die volle Kontrolle über auftretende Ausnahmen bekommt.

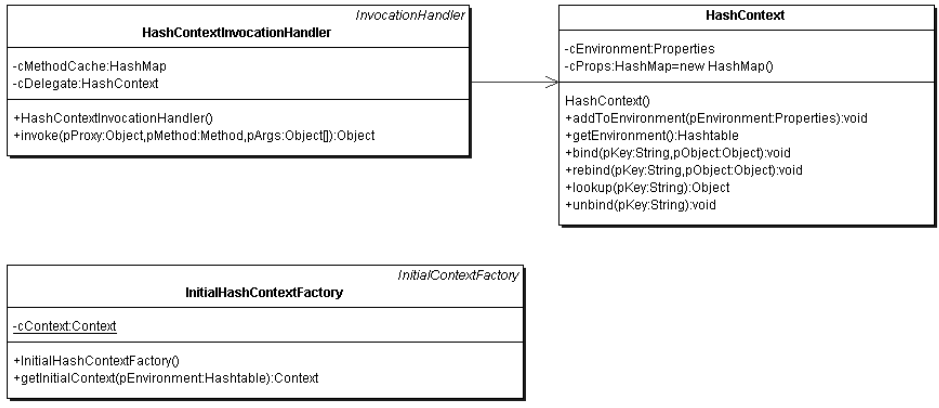


Abbildung 1.34: Eine Hashtable als einfacher JNDI-Kontext

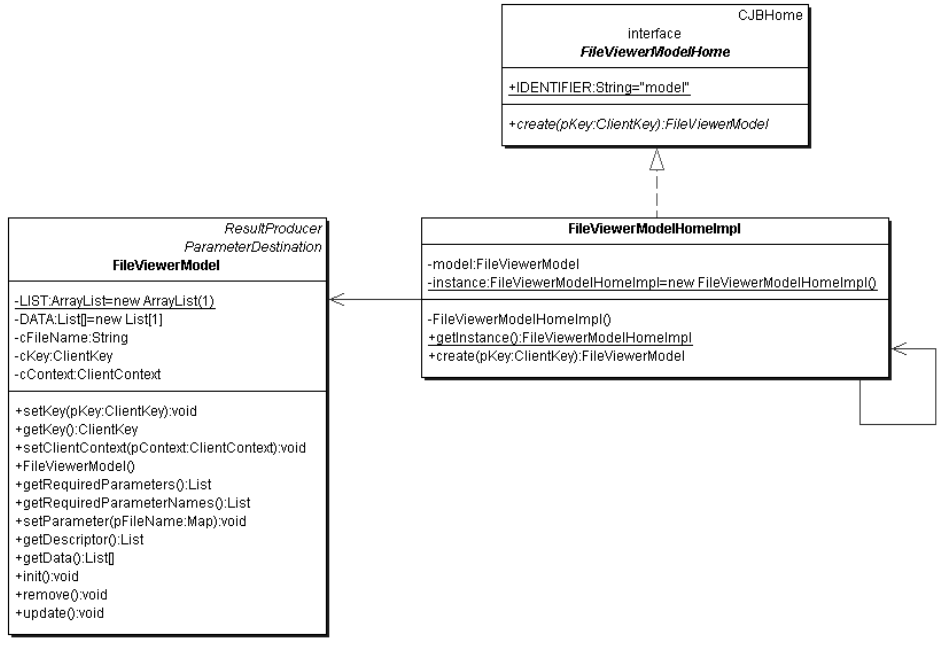


Abbildung 1.35: Ein Beispiel für ein ClientBean: Ein FileViewerModel

Die Konfigurations-Datei ist in diesem Beispiel noch als Properties-Datei ausgebildet. Eine Umstellung auf einen ClientBean-Deskriptor mit einer an die J2EE-Spezifikation angelehnten DTD ist aber problemlos möglich.

Der Arbeitsablauf bei der Erstellung einer Client-Anwendung ist damit analog einer J2EE-Applikation, wenn auch einfacher:

- Der Programmierer schreibt Home-Interfaces.
- Hilfswerkzeuge eines Herstellers einer konkreten Implementierung eines Client-Containers erzeugen daraus standardisiert Implementierungen. In unserem Fall existieren diese Werkzeuge noch nicht, deshalb müssen die Klassen relativ aufwändig selbst implementiert werden. Ebenso müssen hier händisch große Konfigurationsdateien geschrieben werden. Auch dies erfolgt aber bei einem kommerziellen Client Container automatisch und transparent für den Anwendungsprogrammierer!
- Der Programmierer koppelt seine Anwendung durch Lookups auf die Home-Interfaces im JNDI-Kontext. Dies erfolgt vollkommen analog zum J2EE-Programmiermodell.

Es bleibt dem Hersteller des Client-Containers hier noch freigestellt, komfortable Dienste zur Verfügung zu stellen. Das Konzept des »Client Side Containers« wird jedoch aktuell als Java Specification Request im Rahmen des Java Community Processes als neues Java-API definiert.

1.11 Visual Basic und Java

1.11.1 Problemstellung

Wie kann ich aus einem Visual Basic-Programm heraus Java anbinden?

1.11.2 Technischer Hintergrund

Eine direkte Interoperabilität zwischen Java und dem Betriebssystem Windows von Microsoft ist nicht vorgesehen. Eine Kommunikation der Sprachwelten ist jedoch selbstverständlich auf die unterschiedlichsten Arten möglich. Das Spektrum reicht von kommerziellen CORBA-Brokern über einfache Socket-Kommunikation bis hin zur quasi-direkten Kommunikation über das Java Native Interface. Die .NET-Technologie wird das bis vor kurzem propagierte COM-Modell weiterhin unterstützen. Der Zugriff wird jedoch in Zukunft bevorzugt XML-basiert über Web Services (SOAP-Protokoll) erfolgen. Ein Beispiel für die Anbindung einer .NET-Anwendung an die J2EE-Platform ist im weiteren Verlauf gegeben. Es ist jedoch nicht zu erwarten, dass kurzfristig alle VB-Projekte auf die .NET-Plattform portiert werden.

Sun bietet zwei Produkte an, mit denen ein COM-Objekt transparent JavaBeans ansprechen kann:

- Die Umsetzung des DCOM- in das Java RMI-Protokoll wird durch den als Erweiterung erhältlichen Client Access Service realisiert.
- Die Java-Beans als ActiveX-Bridge war bis zur Version 1.3.1 im JDK enthalten. Ab dem JDK 1.4 ist diese Unterstützung aber weggefallen. Durch den Client Access Service der J2EE ist aber weiterhin die direkte Einbindung von Java-Komponenten möglich.

Grundlage der Anbindung an das Betriebssystem ist das JNI, welches maschinenunabhängig Interface-Funktionen definiert, die indirekt den Speicher und damit die Java-Objekte innerhalb der Virtuellen Maschine ansprechbar machen. Fester Bestandteil des JNI ist das »Invocation-API«, das die VM selbst kontrollierbar macht. So kann ein C-Programm mit wenigen Zeilen eine Java-Anwendung starten und darin dynamisch eine Java-Klasse laden und eine Methode (hier die statische main-Methode) ausführen:

```
#include <jni.h>
#include <stdio.h>
#include <stdlib.h>
#define _JNI_IMPLEMENTATION_ IMPL
JavaVM *jvm; /* Pointer auf die Java VM */
JNIEnv *env; /* Pointer auf das Nativ Method Interface */
JavaVMInitArgs vm_args; /* VM Initialisierungs-Argumente */JavaVMOption
options[1];
void main(int argc, char **argv, char **envp){
 jobject StringObj;
 char str[] = "Greetings from C++!";
 options[0].optionString= "-Djava.class.path=c:\\classes";
 vm_args.version = JNI_VERSION_1_2;
 vm_args.options = options;
 vm_args.nOptions = 1;
 vm_args.ignoreUnrecognized = JNI_FALSE;
 printf("C++: Creating Java VM \n");
 JNI_CreateJavaVM(&jvm, (void **) &env, (void *)&vm_args);
 /* den char* in ein StringObject umwandeln */
 StringObj = env->NewStringUTF(str);
 printf("Searching for JavaClass JMain...\n");
 jclass cls = env->FindClass("JMain");
 if (cls == NULL){
 printf("Class JMain not found! \n Exit");
 exit(-1);
 }
 else{
 printf("Found Class JMain\n");
 }
 printf("Calling JMain.start()...\n");
 jmethodID mid = env->GetStaticMethodID(cls, "start", "(Ljava/lang/String;)V");
 if (mid == NULL){
```

```
 printf("method start not found");
 exit(-1);
}
else{
 printf("found method start\n");
}
 env->CallStaticVoidMethod(cls, mid, StringObj);
 printf("Destroying Java VM\n");
 jvm->DestroyJavaVM();
} // main()
```

Die Programmierung des JNI ist aber aufgrund der Komplexität der Funktionsaufrufe und der Vielzahl an möglichen Fehlern auf der nativen C/C++-Seite sicherlich kein Bestandteil der normalen Anwendungsprogrammierung. Stattdessen wird das JNI für konkrete Anforderungen selber nochmals umhüllt und damit dem Anwendungsprogrammierer transparent und komfortabel zur Verfügung gestellt. Dieses Aufgabengebiet wird häufig von kommerziellen Anbietern übernommen. Sun selbst stellt für die Microsoft-COM-Umgebung die »Java Client Access Services Bridge« zur Verfügung, der als »Early Access«-Version die J2EE ergänzt. Andere Anbieter vertreiben aber auch fertige Lösungen wie beispielsweise die Anbindung an COBOL-Bibliotheken. Auch Anbieter von Applikationsservern bieten Produkte an, so z. B. BEA mit Weblogic jCOM.

1.11.3 Lösung

Die Java CAS-COM-Bridge registriert bei ihrer Installation Bibliotheken, die sofort innerhalb des Microsoft Developer-Studios eingebunden werden können. Darin sind in der »Early Access« Version 1.0 sowohl Komponenten zum Ansprechen der Virtuellen Maschine als auch Bibliotheken zur direkten Kommunikation mit dem Applikationsserver enthalten. Diese Komponenten stehen als COM-Objekte zur Verfügung. Die CAS-COM-Bridge besteht in der bisher vorliegenden Version aus einem dreischichtigen Ansatz:

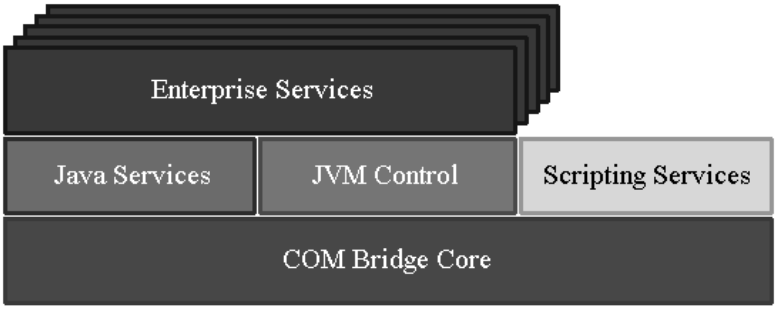


Abbildung 1.36: Die Schichten der CAS-COM-Bridge, Quelle: Dokumentation von Sun Microsystems

Die unterste Schicht ist nichts anderes als eine COM-konforme Umhüllung der JNI-Funktionen. JVM Control kontrolliert den Aufruf der Virtuellen Maschine, die Java Services stellen Funktionen zum Laden von Klassen und Ausführen von Methoden zur Verfügung. Darauf setzen die Enterprise Services auf, die schließlich den Applikationsserver ansprechen. Im Installationsumfang sind die Enterprise Services für die Referenz-Implementierung, für BEA Weblogic, Websphere, Silverstream und iPlanet enthalten.

1.11.4 Praxis

Die Installation der CAS-Bridge gestaltet sich sehr einfach, die Einbindung beispielsweise in das Microsofts Developer Studio ist sofort möglich. Die parallele Entwicklung innerhalb verschiedener Entwicklungsumgebungen gestaltet sich jedoch recht umständlich, da pro Prozess nur eine einzige Virtuelle Maschine gestartet und dann nicht mehr gestoppt werden kann. Als »Prozess« dient hierbei leider die Entwicklungsumgebung selbst, so dass eine Änderung der beteiligten Java-Klassen einen Neustart des Developer Studios erzwingt, um die neue Versionen nutzen zu können. Genauso beendet ein »System.exit(0);« auf Java-Seite sofort die Entwicklungsumgebung.

Um eine Java-Komponente aus Visual Basic heraus ansprechen zu können, gibt es zwei Möglichkeiten:

- Direkte Verwendung der COM-Bridge. Hierzu müssen zwei Standard-Bibliotheken eingebunden werden:

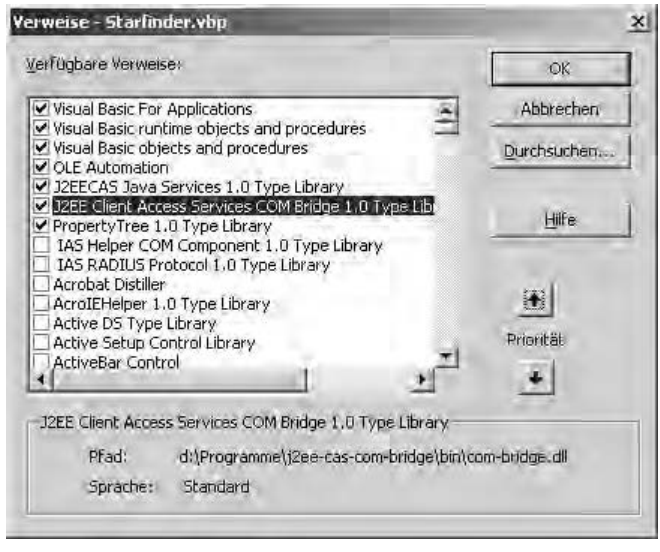


Abbildung 1.37: Die einzubindenden Bibliotheken der Client Access Services Bridge

Innerhalb von Visual Basic kann dann java-ähnlich kodiert werden, jedoch ohne einen typsicheren Kompiler:

```
Private Sub Form_Load()
 Set JVM = New JvmStarter
 JVM.StartJvm
 ShowJavaConsole
 Dim System As Object
 Set System = GetJavaStaticsFor("java.lang.System")
 System.setProperty "com.hotspot.java.naming.factory.initial",
"com.hotspots.javax.naming.InitialHashContextFactory"
 Dim Args(2) As String
 Args(0) = "c:\ \config\VBStarfinder.properties"
 Args(1) = "starfinder"
 Dim Launcher As Object
 Set Launcher =
GetJavaStaticsFor("com.hotspots.javax.cjb.spi.launcher.ClientLauncher")
 Launcher.main Args
 Set StarfinderBeanWrapper =
JavaNew("com.hotspots.starfinder.controller.bean.StarfinderBeanWrapper")
MsgBox "Created : " & StarfinderBeanWrapper.getClass().toString()
End Sub
```

- Erzeugen einer Type-Library durch das im CAS enthaltene Tool »GenTypeLib«. Damit wird aus der Java-Klasse eine COM-Komponente generiert, so dass das Developer Studio dann anhand des Types die vorhandenen Eigenschaften und Methoden prüfen kann. Die erzeugte Typbibliothek muss auf allen Clients installiert werden.

Die zweite Möglichkeit ist sicherlich zu bevorzugen, da die Programmierung ansonsten sehr fehleranfällig ist. Es ist weiterhin empfehlenswert, die Schnittstelle zwischen den Programmiersprachen möglichst einfach zu halten. In den Beispielen wird als koppelndes Interface das VBBridgeIF definiert. Ein VB-Programm setzt durch den Aufruf der Methode addParameter zu übergebende Paramter und startet die Java-Anweisung durch die action-Methode. Die Ergebnisse werden als Liste von beschreibenden Elementen (`getDescriptor()`) sowie die eigentlichen Daten als Listen-Array (`getResult()`) ausgelesen. Aufgetretene Exception-Meldungen sind durch `getExceptionMessage()` zugreifbar.

Aus der `VBBridge`-Klasse wird eine Type-Library generiert und in die VB-Anwendung integriert:

```
gentypelib
-javaclass com.hotspots.java.combridge.VBBridge
-lib VBBridge.tlb
```

und im Unterverzeichnis »output« abgelegt.

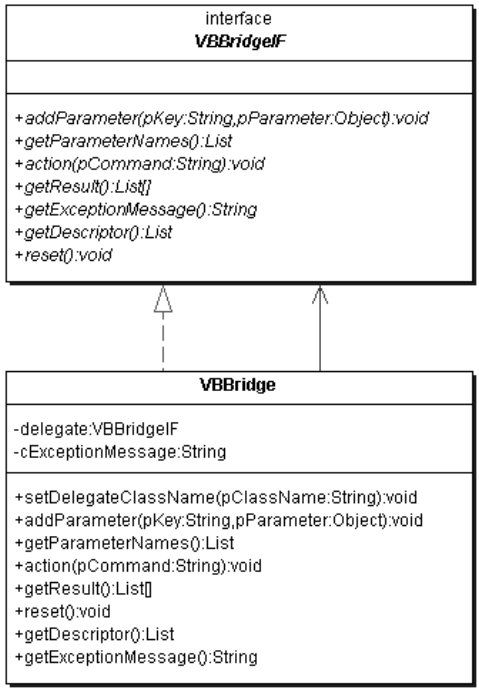


Abbildung 1.38: Das koppelnde Interface zwischen Visual Basic und Java

Abbildung 1.39: Die eingebundene VB-Bridge

Die Implementierung des VBBridgeIF-Interfaces lädt die eigentliche funktionelle VB-Bridge reflektiv nach. Eine Visual-Basic-Anwendung kann somit jede Java-Anwendung durch diese Bridge nutzen:

```
Set TheVBBridge = JavaNew("com.hotspots.java.combridge.VBBridge")
TheVBBridge.setDelegateClassName
"com.hotspots.starfinder.controller.bean.StarfinderBeanWrapper"
```

Methoden werden durch die standardisierte Schnittstelle aufgerufen:

```
TheVBBridge.Reset
For i = 1 To UBound(Param)
TheVBBridge.addParameter Param(i).ParameterName, Param(i).ParameterWert
Next
TheVBBridge.Action (ActionCommand)
Set ResultDescriptor = TheVBBridge.getDescriptor
...
```

Das Ansprechen von EnterpriseBeans wird durch eine weitere Schicht, den Enterprise Services, weiter vereinfacht. Diese Schicht benutzt intern die bisher direkt verwendeten Komponenten, fügt aber eine Reihe weiterer Funktionen hinzu. Dazu werden weitere applikationsserver-abhängige DLLs eingebunden.

Abbildung 1.40: Einbinden der Library für die J2EE-Referenzimplementierung

Damit können Referenzen auf die Home- und Remote-Interfaces des EJBs mit einfachen Aufrufen erhalten werden:

```
Private Function getAppServer() As J2eeRiServices
 If Not AppServerInitialized Then
 Set AppServer = New J2eeRiServices
 AppServer.ProviderURL = "iiop://localhost:1050"
 AppServer.SecurityPrincipal = "j2ee"
 AppServer.SecurityCredentials = "j2ee"
 End If
 Set getAppServer = AppServer
End Function

Private Function getStarIterator() As Object
 If Not StarIteratorInitialized Then
 Dim StarIteratorHome As Object
 Set StarIteratorHome =getAppServer().LookupEjbHome("StarIteratorBean",
"com.abien.j2ee.starsearch.iterator.StarIteratorHome")
 Set StarIterator = StarIteratorHome.Create
 End If
 Set getStarIterator = StarIterator
End Function
```

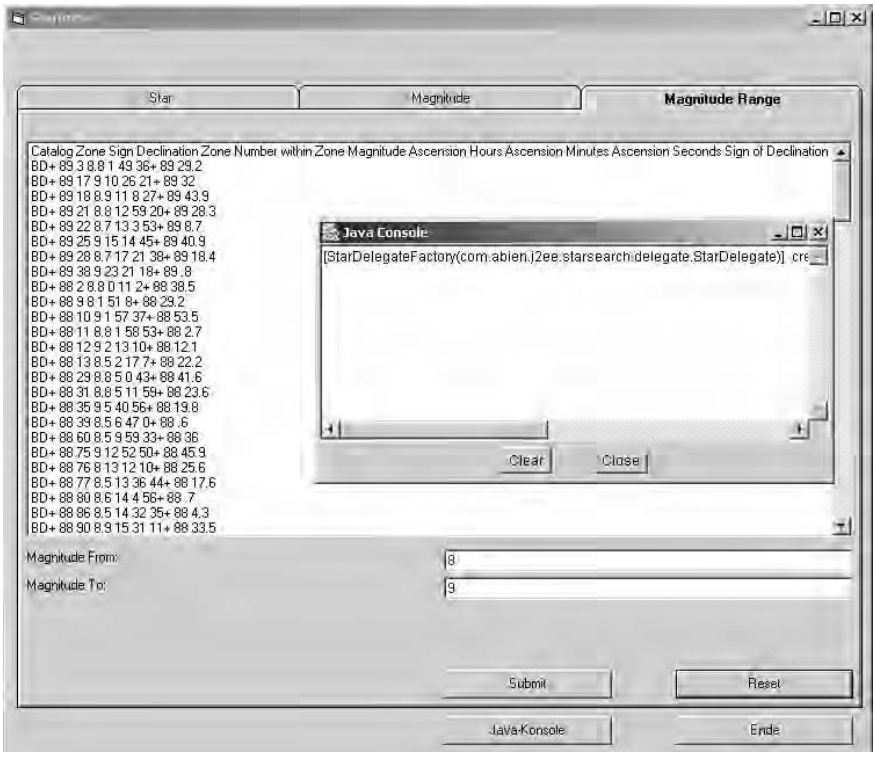


Abbildung 1.41: Eine Starfinder-GUI realisiert mit Visual Basic 6.0

Die angegebene Provider-URL verwendet das Standard-Protokoll für die Referenz-Implementierung der J2EE von Sun.

Eine kommerzielle Alternative hierzu ist beispielsweise für den Weblogic Applikationsserver erhältlich. BEAs Weblogic Applikationsserver enthält eine eigene Variante der Integration von COM namens »jCOM«. Das jcom-Paket ist in der Version 7.0 enthalten und bietet eine bidirektionale Verbindung zwischen den Welten von COM und Weblogic: Jede COM-Komponente kann nach Installation der benötigten nativen Bibliotheken den Applikationsserver ansprechen und umgekehrt. Genauso wie Enterprise Client Access Services basiert die Technologie auf dem JNI. BEA sieht insgesamt drei unterschiedliche Alternativen zum Ansprechen des Applikationsservers von einem Windwos-COM-Client aus vor:

- Ansprechen der Virtuellen Maschine mit DCOM über TCP/IP oder
- über COM auf eine auf lokal laufende VM.
- Verwendung eines »Monikers«.

Die Client Access Bridge von Sun verwendet zwar auch Moniker, die jedoch nicht zum Zugriff auf den Applikationsserver verwendet werden können. Deshalb wird im Folgenden der BEA-Moniker näher erläutert:

Ein Moniker (auf deutsch: Spitzname) ist eine eindeutig einem Namensraum zugeordnete ID in Form einer Zeichenkette, die ein COM-Objekt verwenden kann, um ein anderes, auch entfernt liegendes Objekt anzusprechen. Moniker sind Bestandteile der Windows-Plattform, so dass zum Ansprechen des Applikationsservers neben der Grund-Installation keinerlei zusätzliche Komponenten auf einem Windows-Client installiert werden müssen.

Im Weblogic muss die jCOM-Unterstützung aktiviert werden:

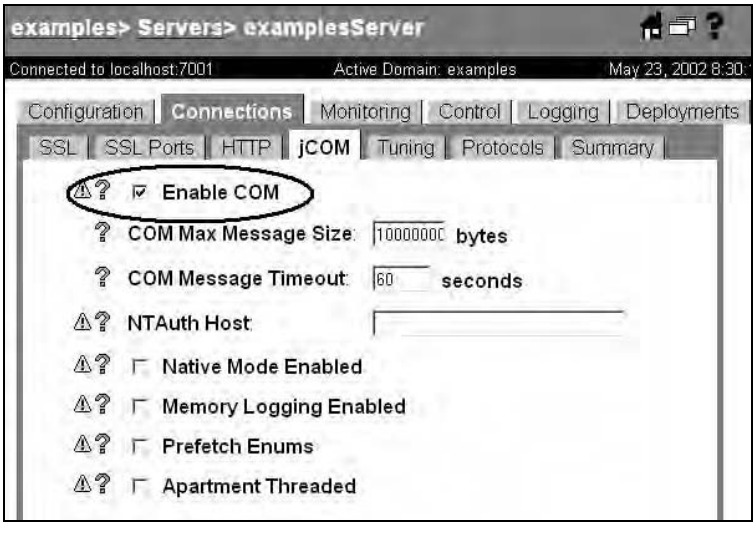


Abbildung 1.42: Aktivierung des jCOM-Features in der Administrationskonsole des BEA Weblogic 7.0

Den Moniker auf einen auch entfernt laufenden Server erhält man einfach über das intern installierte Servlet *http://host:port/bea_wls_internal/com*, das die ID als Zeichenkette zurückgibt. Das Servlet kann direkt aus dem Programm heraus angesprochen werden. Nachdem der Moniker eindeutig ist, kann die Zeichenkette auch z.B. im Browser angezeigt, kopiert und im COM-Programm hart kodiert verwendet werden:

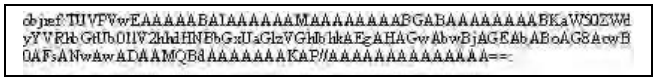


Abbildung 1.43: Der Moniker des Weblogic für »localhost://7001«

Ein Visual Basic-Client verwendet die Methode »GetObject« und kann für das angesprochene Objekt Methoden aufrufen, falls er die notwendigen Berechtigungen besitzt.

```
Dim javaObject AS Object
Set javaObject = GetObject(Moniker)
```

Aus Sicherheitsgründen hat das jCOM-Modul keinerlei Berechtigung, auf die Klassen der installierten J2EE-Komponenten zuzugreifen. Diese Berechtigungen müssen für alle Java-Klassen, die der COM-Client benutzen will, eingeräumt werden, selbst für die Klassen der JRE-Bibliotheken. Dies muss in der Service-Konfiguration für jCOM erledigt werden:

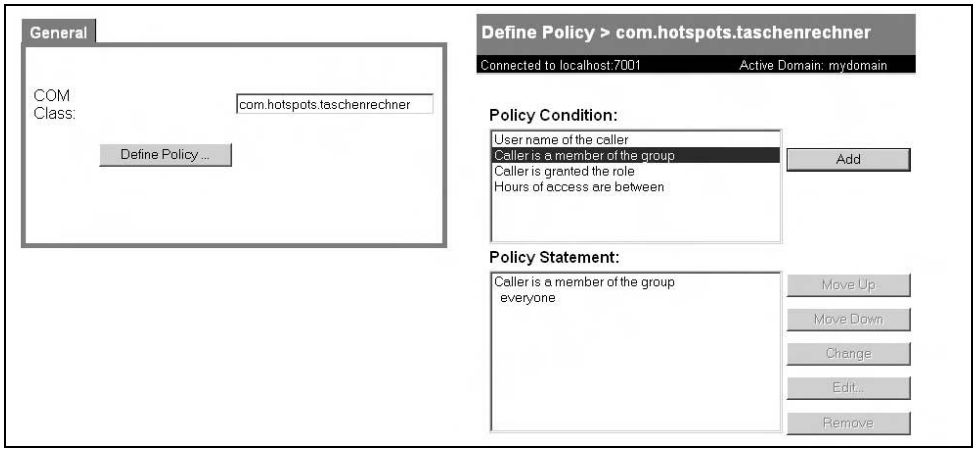


Abbildung 1.44: Definition von Berechtigungen für jCOM-Clients

Das VB-Programm ist einfach, bietet aber natürlich keinerlei Typsicherheit:

```
Private taschenrechner As Object
Public Sub Bind()
Dim objTemp As Object
Dim home As Object
On Error GoTo ErrOut
Set objTemp =
GetObject("objref:TUVPVwEAAAAABAIAAAAAAMAAAAAAAABGABAAAAAAAABKaW5OZWdyYVRhbGtUbOll
V2hhdHNBbGxUaGlzVGhlbhkAEgAHAGwAbwBjAGEAbABoAG8AcwBOAFsANwAwADAAMQBdAAAAAAAKAP//
AAAAAAAAAAAAA==:")
Set home = objTemp.get("myserver:jndi:Taschenrechner")
Set taschenrechner = home.Create()
Exit Sub
ErrOut: MsgBox "Bind Error", vbInformation, "Error Binding Taschenrechner"
End Sub
Public Function doPlus(Zahl1 As Double, Zahl2 As Double)
 doPlus = taschenrechner.addieren(Zahl1, Zahl2)
End Function
Public Function doMinus(Zahl1 As Double, Zahl2 As Double)
 doMinus = taschenrechner.subtrahieren(Zahl1, Zahl2)
End Function
```

1.12 Microsoft Office und Java

1.12.1 Problemstellung

Wie kann aus den Programmen von Microsoft Office auf den Applikationsserver zugegriffen werden?

1.12.2 Technischer Hintergrund

Die Verwendung des applikationsspezifischen MIME-Typs ist eine einfache Möglichkeit, den Browser zum Start des zugehörigen Programms zu veranlassen. Die vom Webserver übermittelten Daten werden dann direkt in dieser Anwendung dargestellt.

Die meisten Office-Anwendungen bieten zur komfortablen Automatisierung und Integration zusätzliche Skript-Sprachen an. Es ist natürlich wünschenswert, diese Skripte zu nutzen, um auf Informationen eines Applikationsservers zugreifen zu können. Das Spektrum der Anwendungen reicht von einfachen Serienbriefen über die Aufbereitung von Daten in Diagrammen bis hin zu personalisierten Angebotsschreiben mit Zugriff auf ein Content Management System.

1.12.3 Lösung

Die Kopplung von MIME-Typen an eine Applikation erfolgt bereits auf Betriebssystemebene und ist damit ebenso wie die Installation der Anwendung die Aufgabe des System-Administrators.

Skript-Sprachen können Bibliotheken einbinden. So kann die CAS-Bridge auch in der Programmiersprache Visual Basic for Applications (VBA) verwendet werden. Damit ist das Problem des Zugriffs auf den Applikationsserver bereits gelöst. Die oben beschriebenen Anforderungen reduzieren sich auf die Implementierung in VBA.

1.12.4 Praxis

Mit der folgenden JSP-Seite kann der Anwender im Browser-Fenster Excel starten:

```
<%@ page contentType="application/vnd.ms-excel" %>
Tabelle
    Jan    Feb    Mar
A    11    12    13
B    21    22    23
C    31    32    33
```

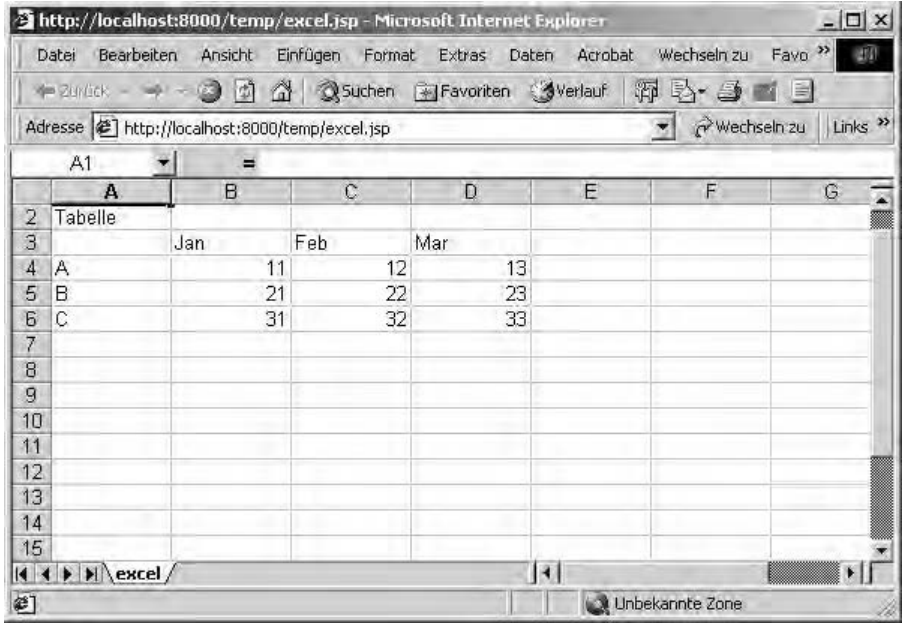


Abbildung 1.45: Im Browser geöffnete Excel-Anwendung

Diese Lösung mag in einem Intranet praktisch sein, scheidet aber in einer Internet-Umgebung aus, da die Beschränkung auf proprietäre Dateiformate und konkrete Anwendungen nicht akzeptabel ist. Weiterhin können diese Informationen ausschließlich als Vorlage verwendet oder rein statisch betrachtet werden. Für letztere Anwendung ist aber schon aufgrund der quasi-plattformunabhängigkeit eine PDF-Datei zu bervorzugen. Weiterhin ist diese Lösung weder vom Design (der Server entscheidet, welcher Client-Typ verwendet werden muss) noch vom momentanen Stand der Sicherheit (Stichwort: Makroviren!) befriedigend. Als Datenaustauschformat wird in Zukunft zwar das plattform-unabhängige XML verwendet werden, die aktuellen Office-Pakete unterstützen dieses Format jedoch bisher nicht. Weiterhin ist zu erwarten, dass die unter Mitarbeit von Microsoft entwickelten »offenen« Standards für Geschäftsdokumente schnell wieder proprietäre Elemente enthalten werden.

Einfacher ist auch hier die Benutzung der in den Office-Produkten integrierten, auf Visual Basic basierenden Sprache »Visual Basic for Applications«. Die Integration der Virtuellen Maschine ist genau so einfach wie in einer reinen VB Applikation. Zu beachten ist jedoch der nicht unerhebliche Speicherbedarf der Java Unterstützung, die für den Client eine nicht zu vernachlässigende Belastung bedeutet. Die folgende einfache Microsoft Excel 2000-Tabelle liest eine Sternendatenbank unter Verwendung des StarfinderBeanWrapers aus und stellt die Helligkeiten in einem Diagramm dar.

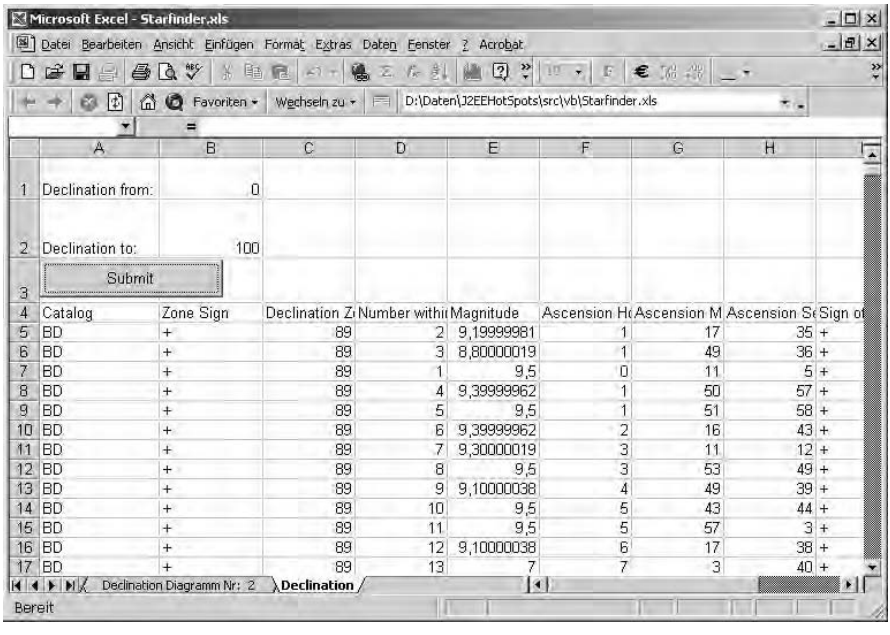


Abbildung 1.46: Auswertung der Starfinder-Datenbank mit dem Microsoft Office-Programm Excel und einem Java-Applikationsserver

```
Set StarfinderBeanWrapper =
JavaNew("com.hotspots.starfinder.controller.bean.StarfinderBeanWrapper")
 StarfinderBeanWrapper.addParameter "Declination from",
Tabelle1.Range("A2").Formula
 StarfinderBeanWrapper.addParameter "Declination to", Tabelle1.Range("B2").Formula
 Tabelle1.Cells.Clear
 Tabelle1.Range("A1").Formula = "Declination from:"
 Tabelle1.Range("B1").Formula = "Declination to:"
 StarfinderBeanWrapper.action ("Declination_Range")
```

Als reine Tabelle benötigt das Programm knapp 11 Megabyte Hauptspeicher. Mit aktivierter Java-Unterstützung werden 21 Megabyte verbraucht, jeweils gemessen mit dem NT-Taskmanager. Der Zusatzbedarf wird nur ein einziges Mal benötigt, da die COM-Access Bridge pro Prozess exakt einmal gestartet wird. Die Virtuelle Maschine wird erst beim Beenden der Office-Anwendung ebenfalls gestoppt.

Auch die Kommunikation zwischen Microsoft Word und Java gestaltet sich völlig problemlos. Ein Java-Programm, das aus einem LDAP-Directory-Server (beispielsweise iPlanet) liest

```
public List getEntries(String pEntry, String pFilter){
try{
ArrayList lRootList = new ArrayList();
DirContext lRoot = (DirContext)cContext.lookup(pEntry);
NamingEnumeration lEnum = lRoot.search("", pFilter, null);
while(lEnum.hasMore()){
HashMap lMap = new HashMap();
SearchResult lResult = (SearchResult)lEnum.next();
Attributes lAttributes = lResult.getAttributes();
NamingEnumeration lAttrEnum = lAttributes.getAll();
while(lAttrEnum.hasMore()){
Attribute lAttribute = (Attribute)lAttrEnum.next();
StringBuffer lBuffer = new StringBuffer();
for (int i = 0; i < lAttribute.size(); i++){
if (i != 0){
lBuffer.append(',');
}
lBuffer.append(lAttribute.get(i));
}
lMap.put(lAttribute.getID(), lBuffer.toString());
}
lRootList.add(lMap);
}
return lRootList;
}
catch(NamingException pException){
System.out.println("Exception:" + pException);
pException.printStackTrace();
return null;
}
}
```

wird von einem einfachen VBA-Makro angesprochen, um einen Serienbrief zu generieren:

```
Public Sub GetLDAP()
 Dim UIDFilter As String
 Dim SearchLdapClass As Object
 Dim SearchLdap As Object
 Dim ResultList As Object
 Set SearchLdapClass = GetJavaStaticsFor("com.hotspots.documents.ldap.SearchLdap")
 Set SearchLdap = SearchLdapClass.getInstance
 UIDFilter = InputBox("Bitte UID-Filter eingeben:")
 Set ResultList = SearchLdap.getEntries("ou=People,dc=siroe,dc=com", "uid=" &
UIDFilter)
 Dim Iterator As Object
 Set Iterator = ResultList.Iterator
 Dim Hashtable As Object
 Dim LdapResult As Object
 While (Iterator.hasNext)
 Set Hashtable = Iterator.Next
 ActiveDocument.Variables("Mail").Value = Hashtable.get("mail")
 ActiveDocument.Variables("Name").Value = Hashtable.get("cn")
 ActiveDocument.Variables("Telephon").Value = Hashtable.get("telephoneNumber")
 Dim fieldCounter As Integer
 Dim fieldCount As Integer
 fieldCount = ActiveDocument.Fields.Count
 For fieldCounter = 1 To fieldCount
 ActiveDocument.Fields(fieldCounter).Update
 Next
 ActiveDocument.PrintPreview
 MsgBox "Weiter?"
 ActiveDocument.ClosePrintPreview
 Wend
End Sub
```

1.13 Java und Microsoft Office

1.13.1 Problemstellung

Wie können meine Microsoft-Office-Datenbestände über einen J2EE-Applikationsserver verfügbar gemacht werden?

Können ODBC-konforme Datenquellen auf dem Endbenutzer-PC zumindest als Zwischenspeicher genutzt werden?

1.13.2 Technischer Hintergrund

Die massive Präsenz von Microsoft-Produkten in der IT-Branche ist ein Fakt, das der J2EE-Entwickler nicht achtlos ignorieren kann. Gewaltige Datenmengen liegen auch in großen Unternehmen häufig in den Dateiformaten des Office-Paketes vor, insbesondere als Excel-Arbeitsblätter und Word-Dokumente. Selbst für die Datenhaltung wird, besonders in klein- und mittelständischen Unternehmen, bei beschränktem Datenvolumen und geringer Benutzer-Zahl durchaus die Access-Datenbank eingesetzt.

Diese Informationen einfach über das Internet zugreifbar zu machen war bisher ein Hauptargument für die Verwendung der Microsoft Active ServerPages-Technologie: Ein in das Betriebssystem integrierter Web Server kann einfach verwendet werden, um beispielsweise Access Datenbanken im Web zu präsentieren. Das von Microsoft entwickelte .NET-Framework integriert diese Formate ebenso nahtlos in seine Architektur, so dass die oben dargestellte Anforderung ohne großen Entwicklungs-Aufwand realisiert werden kann. Dies kann bei der Wahl der Entwicklungsplattform die Entscheidung zugunsten von .NET beeinflussen.

Eine ähnliche Argumentation wird auch für den Arbeitsplatz-PC durchgeführt: Ist bereits mit einer nicht unerheblichen Investition das Microsoft-Office-Paket für den Client lizenziert und installiert, sollen diese Programme natürlich auch verwendet werden. Muss ein J2EE-Client beispielsweise Diagramme oder Reports generieren, ist es sinnvoller, die vorhandene Software zu nutzen, als weitere Werkzeuge zu installieren. Dies ist insbesondere dann gültig, wenn die Daten noch weiter bearbeitet und aufbereitet werden sollen.

Soll stattdessen ein Applikationsserver eingesetzt werden, tritt damit sofort das Problem auf, wie der Server die Daten lesen kann. Eine Konvertierung in eine JDBC-Datenbank oder in XML-Dateien wird vom Anwender nicht immer begeistert unterstützt.

1.13.3 Lösung

Zur Lösung dieses Problems können verschiedene Ansätze verfolgt werden:

- Die in den vorherigen Abschnitten dargestellte Möglichkeit, innerhalb einer laufenden Office-Anwendung eine Virtuelle Maschine zu starten, ermöglicht z.B. ein Ansprechen eines RMI-Server-Objektes, das den Inhalt der Office-Datei entgegennimmt. Damit ist zumindest ein lesender Zugriff möglich.
- Auf der Windows-Plattform ist die »Open Database Connectivity« verfügbar, mit der konforme Datenbanken angesprochen werden können. Aus dem Microsoft-Office-Paket sind dies das Datenbankprogramm »Access« sowie das Tabellenkalkulationsprogramm »Excel«. Damit können unter bestimmten Nebenbedingungen

Daten zumindest gelesen werden. Diese Lösung hat den Vorteil der Portierbarkeit: Soll die Datenhaltung in einem anderen Format erfolgen, genügt der Austausch der JDBC-URL.

- Zum direkten Lesen kann das Apache POI (der »Poor Obfuscation Implementation«, eine kaum verklausulierte Kritik der Autoren über das eigentlich gemeinte OLE2-Dateiformat) genutzt werden. In dieser Bibliothek ist der Datenzugriff auf Excel-Datenbanken bereits veröffentlicht, das Doc-Format des Textverarbeitungsprogramms Word ist in Arbeit, das Präsentationsprogramm »PowerPoint« bereits angedacht.

1.13.4 Praxis

Ziel der folgenden Aktionen ist aufzuzeigen, dass Daten, die in Microsoft Office Dokumenten enthalten sind, auch über den J2EE-Applikationsserver ohne großen Aufwand verwendet werden können.

Um eine Excel-Tabelle mit Hilfe der ODBC-Schnittstelle anzusprechen, wird die altehrwürdige sun.jdbc.odbc.JdbcOdbcBridge verwendet. Dazu kann eine ODBC-Connection mit Hilfe des ODBC-Administrators definiert werden:

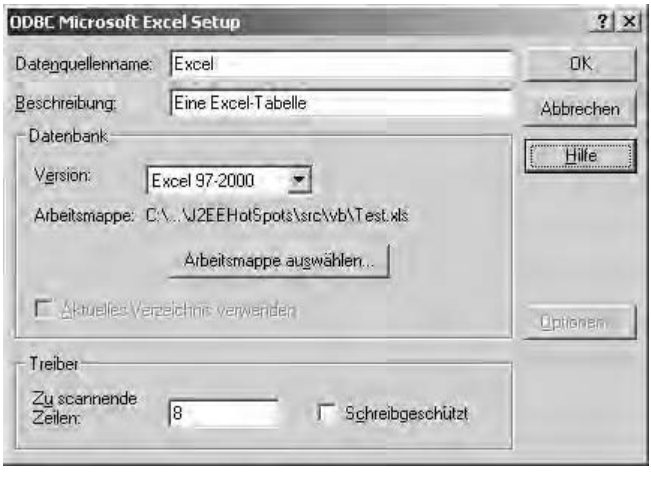
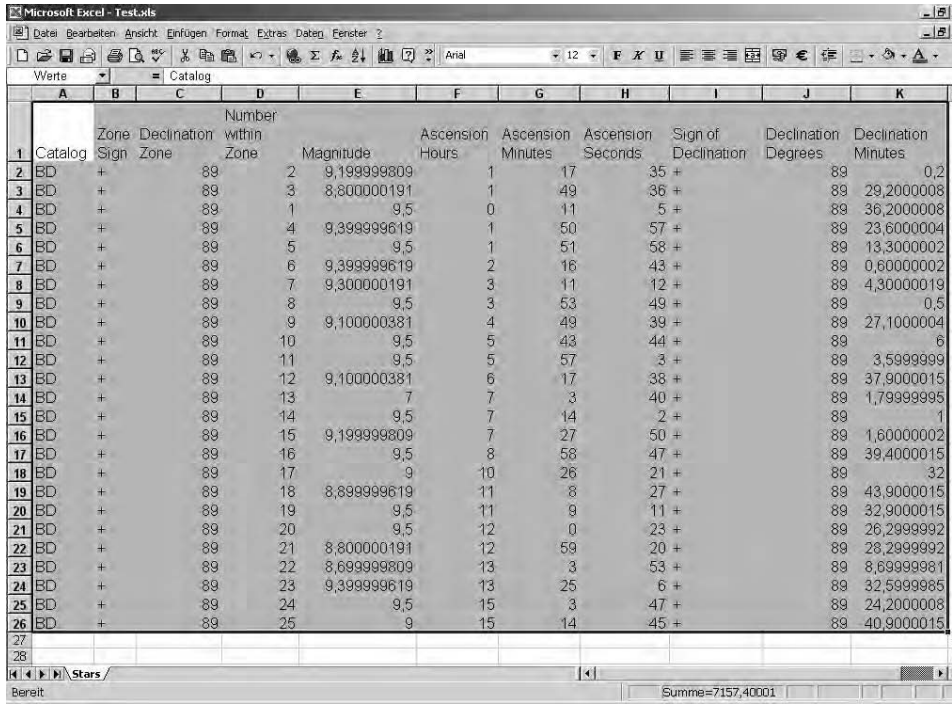


Abbildung 1.47: Konfiguration einer Excel-Datenquelle mit Hilfe des ODBC-Administrators

In der Excel-Tabelle wird ein Name für einen Arbeitsbereich vergeben, in diesem Beispiel »Werte«:



Microsoft Excel - Test.xls

Datei Bearbeiten Ansicht Einfügen Format Extras Daten Fenster ?

Werte = Catalog

	A	B	C	D	E	F	G	H	I	J	K
1	Catalog	Zone Sign	Declination Zone	Number within Zone	Magnitude	Ascension Hours	Ascension Minutes	Ascension Seconds	Sign of Declination	Declination Degrees	Declination Minutes
2	BD	+	89	2	9,199999809	1	17	35	+	89	0,2
3	BD	+	89	3	8,800000191	1	49	36	+	89	29,2000008
4	BD	+	89	1	9,5	0	11	5	+	89	36,2000008
5	BD	+	89	4	9,399999619	1	50	57	+	89	23,6000004
6	BD	+	89	5	9,5	1	51	58	+	89	13,3000002
7	BD	+	89	6	9,399999619	2	16	43	+	89	0,60000002
8	BD	+	89	7	9,300000191	3	11	12	+	89	4,30000019
9	BD	+	89	8	9,5	3	53	49	+	89	0,5
10	BD	+	89	9	9,100000381	4	49	39	+	89	27,1000004
11	BD	+	89	10	9,5	5	43	44	+	89	6
12	BD	+	89	11	9,5	5	57	3	+	89	3,5999999
13	BD	+	89	12	9,100000381	6	17	38	+	89	37,9000015
14	BD	+	89	13	7	7	3	40	+	89	1,79999995
15	BD	+	89	14	9,5	7	14	2	+	89	1
16	BD	+	89	15	9,199999809	7	27	50	+	89	1,60000002
17	BD	+	89	16	9,5	8	58	47	+	89	39,4000015
18	BD	+	89	17	9	10	26	21	+	89	32
19	BD	+	89	18	8,899999619	11	8	27	+	89	43,9000015
20	BD	+	89	19	9,5	11	9	11	+	89	32,9000015
21	BD	+	89	20	9,5	12	0	23	+	89	26,2999992
22	BD	+	89	21	8,800000191	12	59	20	+	89	28,2999992
23	BD	+	89	22	8,699999809	13	3	53	+	89	8,69999981
24	BD	+	89	23	9,399999619	13	25	6	+	89	32,5999985
25	BD	+	89	24	9,5	15	3	47	+	89	24,2000008
26	BD	+	89	25	9	15	14	45	+	89	40,9000015
27											
28											

Stars

Bereit

Summe=7157,40001

Abbildung 1.48: Vergabe eines Namens für den Datenbereich der Excel- Tabelle

Jedes Java-Programm kann dann über einfachstes JDBC auf diesen Bereich zugreifen:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection lCon = DriverManager.getConnection("jdbc:odbc:Excel");
Statement stmt = lCon.createStatement();
ResultSet rs = stmt.executeQuery("Select * from Werte where Magnitude > 9.3");
ResultSetMetaData rsmd = rs.getMetaData();
while(rs.next()){
 …
```

Ein Erweitern des Bereiches, also das Absetzen eines INSERT-Befehls, ist für eine Excel-Tabelle leider so nicht möglich, ein Update dagegen problemlos. Diese Einschränkung ist für eine Microsoft Access-Datenbank nicht gültig, das Programm präsentiert sich in diesem Zusammenhang als vollwertige Datenbank.

Auch die Anbindung einer CMP-EntityBean ist möglich, wenn auch nicht direkt mit dem `JdbcOdbcDriver`. Dieser ODBC-Treiber unterstützt nämlich keine Transaktionen: Versucht der Applikationsserver einen Connection Pool für JDBC-ODBC-Driver aufzubauen, wird die Methode `setAutoCommit(false)` aufgerufen. Die zugehörige native Bibliothek wirft dann sofort eine SQLException, so dass der Treiber nicht direkt innerhalb des Servers eingesetzt werden kann.

Eine eigene Treiber-Implementierung oder das Wrappen der originalen JDBC-ODBC-Connection umgehen dieses Problem. Als Subprotokoll wird die Zeichenkette »j2eeodbc« definiert:

```
public class J2EEJdbcOdbcDriver extends JdbcOdbcDriver{
 static{
 if(JdbcOdbcObject.isTracing())
 JdbcOdbcObject.trace("J2EEJdbcOdbcDriver class loaded");
 J2EEJdbcOdbcDriver jdbcodbcdriver = new J2EEJdbcOdbcDriver();
 try{
 DriverManager.registerDriver(jdbcodbcdriver);
 }
 catch(SQLException sqlexception){
 if(JdbcOdbcObject.isTracing())
 JdbcOdbcObject.trace("Unable to register driver");
 }
 }
 public synchronized Connection connect(String s, Properties properties) throws
SQLException{
 Connection lConnection = super.connect(s, properties);
 if (lConnection != null){
 return new J2EEJdbcConnection(lConnection);
 }
 else{
 return null;
 }
 }
 public boolean acceptsURL(String s) throws SQLException{
 return knownJ2EEURL(s);
 }
 private boolean knownJ2EEURL(String s){
 String s1 = getProtocol(s);
 if(!s1.equalsIgnoreCase("jdbc"))
 return false;
 s1 = getSubProtocol(s);
 return s1.equalsIgnoreCase("j2eeodbc");
 }
private static class J2EEJdbcConnection implements Connection{
 private Connection cConnection;
 J2EEJdbcConnection(Connection pConnection){
 cConnection = pConnection;
 }
…
 public void setAutoCommit(boolean autoCommit) throws SQLException{
 try{
 cConnection.setAutoCommit(autoCommit);
 }
 catch(SQLException pException){

…
}
```

Die J2EE-JDBC-Connection delegiert an die originale Connection weiter, fängt aber in der Methode `setAutoCommit` die vom nativen Treiber geworfene SQLException.

Somit kann der Connection Pool aufgebaut werden, der von einer beliebigen Entity-Bean verwendet werden kann. Es ist jedoch zu beachten, dass die Methodenaufrufe eines EntityBean stets direkt von der Connection bestätigt werden und somit unabhängig vom eingestellten Transaktionsattribut stets als »REQUIRES NEW« aufzufassen sind.

In Anbetracht der weiten Verbreitung von Excel-Tabellen und auch Access-Datenbeständen ist diese Anbindung der Daten an den Applikationsserver zumindest eine nicht zu vernachlässigende Option. Ebenso interessant ist die Möglichkeit, durch die ODBC-Schnittstelle einfache Dateien zeilenweise auslesen zu können. Auch dieser Treiber ist bereits vorhanden und kann durch den ODBC-Administrator relativ einfach konfiguriert werden, so dass auf eine eigene Implementierung verzichtet werden kann.

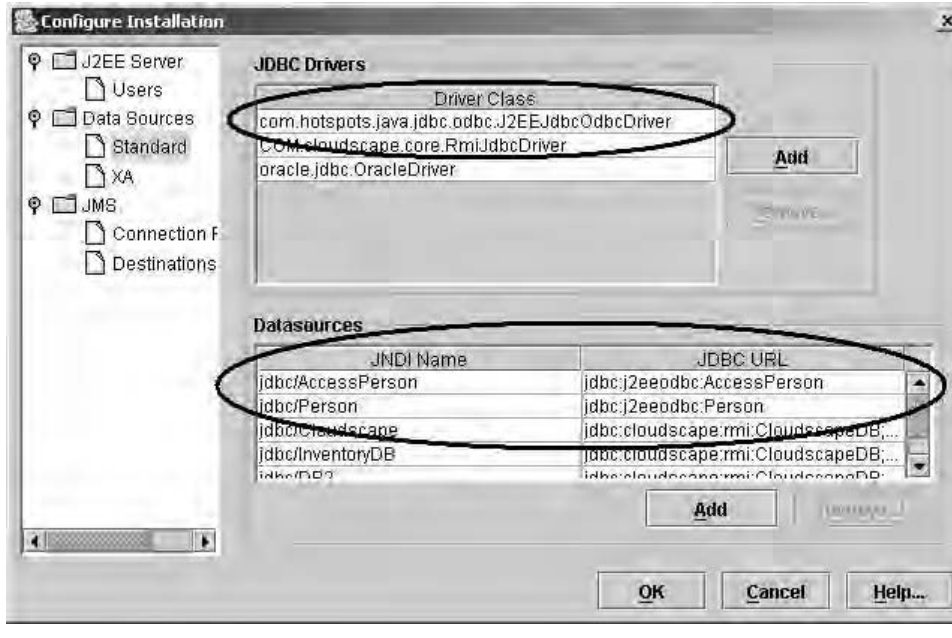
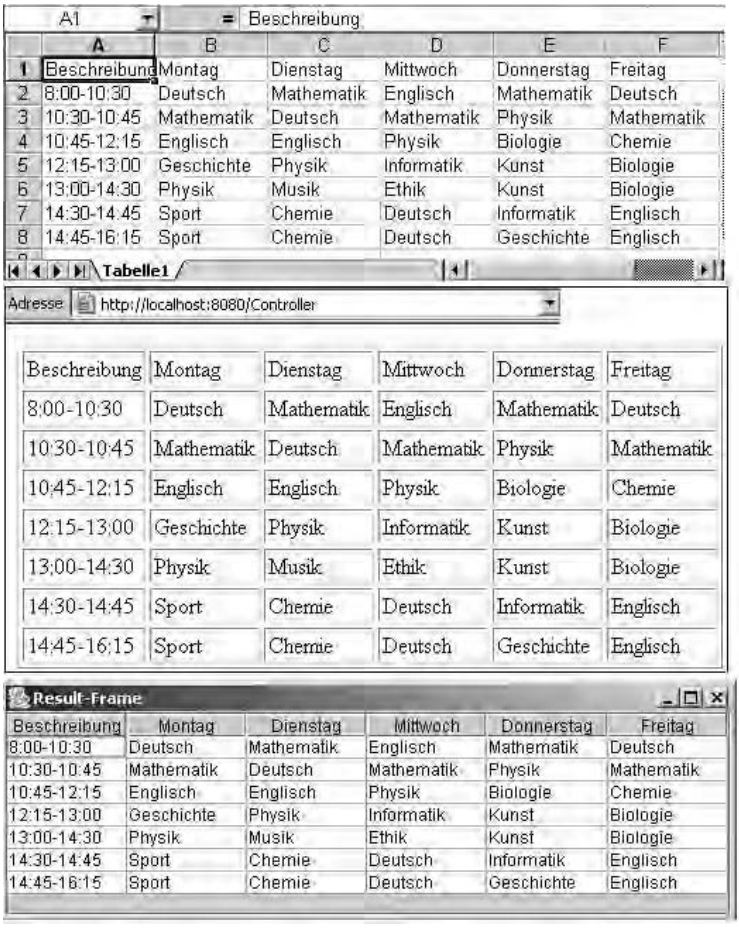


Abbildung 1.49: Einbinden des J2EE-JDBC-Drivers in der Referenzimplementierung

Somit ist es auch mit einem J2EE-Applikationsserver ohne jegliches Problem möglich, Microsoft Office-Datenbestände über eine Java-Anwendung oder über das Web zu präsentieren. Liegen die Daten in Form einer Microsoft Access Tabelle vor, spricht auch nichts gegen einen Update durch den Applikationsserver.



Beschreibung	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
8:00-10:30	Deutsch	Mathematik	Englisch	Mathematik	Deutsch
10:30-10:45	Mathematik	Deutsch	Mathematik	Physik	Mathematik
10:45-12:15	Englisch	Englisch	Physik	Biologie	Chemie
12:15-13:00	Geschichte	Physik	Informatik	Kunst	Biologie
13:00-14:30	Physik	Musik	Ethik	Kunst	Biologie
14:30-14:45	Sport	Chemie	Deutsch	Informatik	Englisch
14:45-16:15	Sport	Chemie	Deutsch	Geschichte	Englisch

Abbildung 1.50: Präsentation einer Excel-Tabelle (oben) im Internet-Browser (Mitte) und Standalone Java Client (unten). Der Zugriff auf die Tabelle erfolgt über die modifizierte JDBC-ODBC-Bridge.

Das Apache POI-Projekt (*http://www.apache.org/poi*) setzt das OLE2-Dateiformat vollständig in Java-Klassen um. Somit können auch dynamische Microsoft-Office-Dokumente, z.B. für Reports und Statistiken erzeugt werden. Die folgende Klasse speichert unter Verwendung der HSSF-Bibliothek (»Horrible Stylesheet Format«) Informationen in einer Excel-Datei namens »Data.xls« ab:

```
package com.hotspots.java.client.persistence;
import java.util.*;
import com.hotspots.javax.cjb.*;
import java.io.*;
import org.apache.poi.hssf.usermodel.*;
public class HSSFPersistenceServiceImpl implements PersistenceService, ClientBean{
//…
```

```
private HSSFWorkbook cBook;
private HSSFSheet cSheet;
//…
 public void init() {
 cBook = new HSSFWorkbook();
 cSheet = cBook.createSheet();
 cBook.setSheetName(0,"Data");
 }
//…
public void write(List pDescriptor, List[] pData){
 writeHeader(pDescriptor);
 for(int i = 0; i < pData.length; i++){
 writeRow(pData[i], (short)(i+1));
 }
 try{
 FileOutputStream lOut = new FileOutputStream("Data.xls");
 cBook.write(lOut);
 }
 catch(Exception pException){
 cContext.handleException(pException);
 }
}
//…
private void writeHeader(List pHeader){
 HSSFRow lRow = cSheet.createRow((short)0);
 for (short i = 0; i < pHeader.size(); i++){
 lRow.createCell(i).setCellValue(pHeader.get(i).toString());
 }
}
private void writeRow(List pRow, short pNum){
 HSSFRow lRow = cSheet.createRow(pNum);
 for (short i = 0; i < pRow.size(); i++){
 lRow.createCell(i).setCellValue(pRow.get(i).toString());
 }
}
}
```

Das Schreiben von Word-Dokumenten ist auf diese Art und Weise jedoch noch nicht möglich, da das Apache-Projekt »HDF« (»Horrible Document Format«) noch nicht fertig gestellt wurde.

Die `HSSFWorkbook`-Klasse ist zwar selbst nicht serialisierbar, kann sich durch die Write-Methode jedoch in einen beliebigen Output-Stream schreiben.

Beim momentanen Stand der Java-Klassenbibliotheken können Excel-Arbeitsmappen und Access-Tabellen durch die JDBC-ODBC-Bridge verlässlich in der Java-Welt gelesen und beschrieben werden. Die Integration in einen Applikationsserver ist mit einem leicht modifizierten JDBC-ODBC-Driver möglich, so dass Daten auch ohne Konvertierung verwendet werden können. Notwendig ist jedoch immer eine nicht triviale Konfiguration der ODBC-Treiber auf jeder Maschine.

1.13.5 Performance

Zur Verwaltung von Daten im Hochlastbereich vieler gleichzeitiger Client-Zugriffe und großer zu transferierender Datenmengen war die Access-Datenbank bekanntermaßen nicht geeignet und wurde auch nie dafür konzipiert. Die neueste Version von Microsoft Accress enthält jedoch als Datenbank-Engine den SQL-Server, so dass für die Datenhaltung auf dem Endbenutzer-PC eine vollwertige Datenbank zur Verfügung stehen wird. Die hier getestete Office-Version ist Office 2000.

Die Intention des hier vorgestellten Tests darf deshalb auch nicht missverstanden werden: Es geht darum, ob es sich lohnt, Daten, die in einem Windows-System in einer einfach-strukturierten Form als Arbeitsblatt, einfache Tabelle oder sogar als einfache Textdatei vorliegen, in eine Datenbank zu konvertieren.

Zum Performance-Test wurde eine Menge von 65.000 (ein Excel-Arbeitsblatt kann maximal 65536 Zeilen verwalten) einfachen Datensätzen der Sternendatenbank in verschiedenen »Datenbanken« abgelegt.

Nummer	Treiber	Beschreibung
1	Microsoft Text-Treiber	Textdateien mit in Zeilen organisierten Daten
2	Microsoft Excel-Treiber	Lesen benannter Bereiche eines Excel Arbeitsblattes
3	Microsoft Access-Treiber	Zugriff auf die Access-Datenbank
4	Einfacher Cloudscape Treiber	Treiber und Programm laufen in der gleichen virtuellen Maschine
5	Cloudscape RMI-Treiber	Der Treiber wird über RMI angesprochen
6	Oracle Thin	Der Oracle Typ 4-Treiber
7	Oracle OCI8	Der native Oracle OCI-Treiber

Die Tabellen der echten Datenbanken wurden nicht indiziert, um den direkten Vergleich der Systeme zu ermöglichen. Aus dieser Menge wurden durch eine Abfrage mit einem java.sql.Statement insgesamt 1163 Datensätze gelesen. Das erzeugte ResultSet wurde in der ersten Testreihe einfach durchlaufen, während in der zweiten Reihe die einzelnen Spalten mit getObject(int) ausgelesen wurden.

Nummer	Reihe 1 [msec]	Reihe 2 [msec]
1	911	1292
2	294	747
3	**146**	470
4	**1.496**	1.555
5	1.144	**38.100**
6	200	360
7	174	**330**

Das Ergebnis ist doch etwas überraschend, da in beiden Varianten der Zugriff auf die Access-Datenbank über die JDBC-ODBC-Bridge zusammen mit Oracle in der Spitzengruppe rangiert. Das bedeutet natürlich nicht, dass Access mit Oracle als Datenbank direkt konkurrieren kann, sondern dass in dieser speziellen Situation (eine unsortierte Tabelle mit 65.000 Datensätzen ohne Relationen wird durchsucht) bei bereits vorhandenen ODBC-Produkten eine weitere Installation nicht lohnt. Selbst eine einfache Textdatei zeigt sich in diesem Testszenarium der Cloudscape-Datenbank überlegen. Insbesondere beim Auslesen des ResultSets ist die Implementierung des RMI-Treibers offensichtlich verbesserungsfähig. Das Problem liegt bei der schwachen Implementierung des RMI-Treibers, der offensichtlich bei jedem Auslesen nochmals eine aufwändige Netzwerk-Kommunikation startet.

Resümee:

Mit Hilfe einer leicht modifizierten JDBC-ODBC-Brücke ist die Integration beliebiger Microsoft Office Daten in die J2EE-Plattform möglich. Als persistente Schicht für Entity-Beans können Access Datenbanken, aber auch Excel Datenblätter oder sogar einfache Dateistrukturen dienen. Als Zwischenspeicher sind die im Windows Betriebssystem enthaltenen Treiber zumindest für Client Caches durchaus geeignet.

1.14 J2ME und die J2EE

1.14.1 Problemstellung

Wie ist das Zusammenspiel zwischen der Java 2 Micro und Enterprise Edition?

1.14.2 Technischer Hintergrund

Die Java 2 Micro Edition

Die Java 2 Micro Edition enthält neben eigens eingeführten Typen eine Untermenge der J2SE Klassenbibliotheken. Aufgrund der extrem heterogenen Umgebung auf dem Kleingerätemarkt ist es nicht möglich, eine einheitliche Java-Laufzeitumgebung zu definieren. Stattdessen wird die Mächtigkeit eines Gerätes durch die so genannten »Konfigurationen« charakterisiert. Es wird grob unterschieden zwischen der so genannten »Connected Device Configuration« (CDC), bei der eine permanente Netzwerkverbindung vorausgesetzt wird und der »Connected Limited Device Configuration« (CLDC). Eine feinere Klassifizierung ermöglichen dann die »Profiles«, deren momentan wohl bekanntester Vertreter das »Mobile Information Device Profile« (MIDP) für Mobiltelefone und Organizer ist.

Das direkte Ansprechen von EnterpriseBeans über einen JNDI-Lookup und anschließendes Laden eines Stubs ist weder für die »Connected Limited Device Configuration« noch

für die »Connected Device Configuration« möglich. Ein dynamischer Download eines serialisierten Stubs sowie die Kommunikation über IIOP übersteigen die beschränkten Ressourcen eines Kleingerätes. Die MIDP-Anwendung kann aber direkt via http mit einem Servlet kommunizieren:

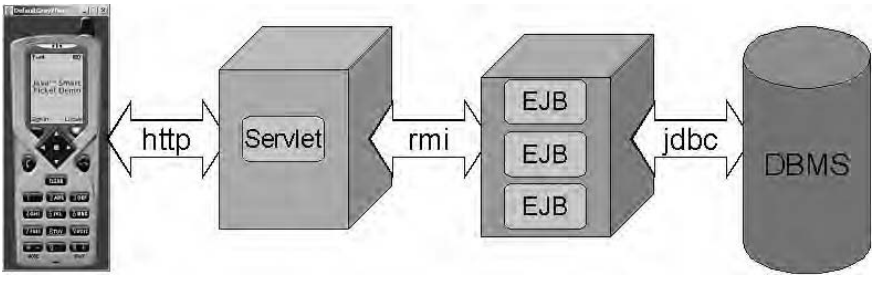


Abbildung 1.51: Anbindung eines MIDP-Kleingerätes an ein Servlet

Ein direkter Zugriff auf den Applikationsserver wird erst durch das RMI Profile möglich werden. Diese Spezifikation definiert eine Untermenge der J2SE-RMI-Bibliotheken, mit der das »Kleingerät« Stubs vom Server laden kann. Das RMI-Profile ist seit Ende Januar 2002 in der »public draft« Version veröffentlicht und wird von keinen kommerziellen Geräten unterstützt. Dieses Profil basiert auf der CDC und ist damit für Mobiltelefone nicht verfügbar.

Das Wireless Application Model

Eine Alternative zum J2ME-Ansatz ist das »Wireless Application Model« (WAP). Ähnlich wie HTML-Seiten werden WAP-Applikationen von einem WAP-Server als einfach lesbare Datenströme übermittelt. Die zugrunde liegende Sprache, die »Wireless Markup Language« entspricht dem XML-Standard. Auch eine Skript-Sprache, »WMLScript«, ist bereits definiert und ermöglicht die Ausführung von Programmen auf dem Kleingerät.

Die Entscheidung, ob für die Erstellung einer Anwendung für Kleingeräte die J2ME oder WAP eingesetzt werden soll, ist genauso wenig eindeutig zu treffen wie die Analogie zu Applets und HTML-Formularen. Je mehr Funktionalität auf das Kleingerät ausgelagert werden kann oder muss, desto geeigneter ist eine Implementierung mit Hilfe der J2ME. So definiert das MID-Profile beispielsweise so genannte »RecordStores«, mit denen eine einfache Datenhaltung auf dem MIDP-Device erfolgen kann. Genauso ist zu erwarten, dass sich die Performance und Speicherverwaltung einer Virtuellen Maschine für die J2ME einem Script-Interpreter als überlegen erweisen wird. Für die Präsentation statischer Inhalte und für die Realisierung einer einfachen Datenerfassung ist WAP natürlich eine echte Alternative.

Es ist aber natürlich zu beachten, dass WAP und WMLScript eigene Programmiersprachen sind, die vom Entwickler erst gelernt werden müssen. Die Integration von HTML und WAP unter der gemeinsamen Obermenge XHTML ist noch nicht gelungen.

Ein Java-Entwickler findet aber in der J2ME eine bekannte Syntax und Semantik vor, so dass eine Einarbeitung in die neuen bzw. modifizierten Klassenbibliotheken problemlos und schnell erfolgen kann.

Für den Server-Programmierer stellt sich das Problem, dass beide Varianten berücksichtigt werden müssen. WAP wird in der Regel durch ein zusätzliches Gateway aus den Ausgaben eines Web Servers generiert:

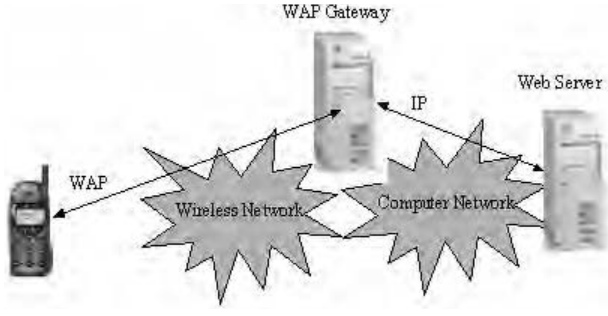


Abbildung 1.52: Das WAP-Netzwerk

Das WAP-Gateway ist eine separat zu konfigurierende Schicht, die, wie beispielsweise beim Weblogic Applikationsserver, bereits in den Web Server integriert sein kann. Programmtechnisch kann eine WAP-Seite als JSP ausgebildet werden. Alternativ kann aber auch ein Servlet-Filter zwischengeschaltet werden, der eine XSL-Transformation der normalen HTML-Seite nach WML durchführt. Als MIME-Typ ist »text/vnd.wap.wml« zu verwenden. Eine generische Umsetzung von HTML-Seiten nach WML ist im Kapitel »Die Präsentationsschicht« gegeben.

1.14.3 Lösung

Eine vergleichbare Implementierung mit der J2ME wird die CLDC und MIDP verwenden. Damit muss vom Kleingeräte-Hersteller der javax.microedition.Connector implementiert werden, der eine http-Verbindung mit einem Webserver und damit auch eine Kommunikation mit Servlets aufbauen kann. Im Gegensatz zu browser-basierten Clients ist HTML als Datenaustauschformat in der Regel ungeeignet. Einige Geräte enthalten in ihrer Konfiguration einfache SAX-basierte Parser, so dass XML-Dateien übertragen werden können. SAX ist aber im MIDP nicht zwingend vorgesehen und wird deshalb hier nicht verwendet. Das Open Source Projekt des »kxml« ist damit beschäftigt, eine schlanke SAX-Implementierung für Kleingeräte zu erstellen. Häufig kommen anwendungsspezifische Formate zum Einsatz.

Mit den für ein Kleingerät zur Verfügung stehenden UI-Komponenten können einfache Oberflächen aufgebaut werden, mit denen die Eingabe und Aufbereitung der empfangenen Informationen erfolgen. Die darzustellenden Daten werden vom Web- bzw. Applikationsserver geliefert. Bei der Implementierung sind die J2ME-spezifischen Besonderheiten zu berücksichtigen. Im MID-Profile sind diese:

- Keine vollständige Unterstützung der Java-APIs der J2SE. Die verwendeten Klassenbibliotheken gründen teilweise auf dem JDK 1.1.8; aus den Klassen wurden alle »unnötigen« Methoden und Attribute entfernt. Es stehen weder der Serialisierungsmechanismus (java.io mit den Typen Serializable, ObjectInputStream und ObjectOutputStream) noch die »Long-Term Persistence« für JavaBeans zur Verfügung. Auch der aufwändige Reflection-Mechanismus wird nur durch die statische Methode Class.forName(String) sowie newInstance() unterstützt. Damit kann reflektiv nur der parameterlose Konstruktor einer Klasse aufgerufen werden. Auch das .class-»Attribut« der Java-Datentypen ist nicht bekannt. Das Collections-API ist nur in der CDC enthalten. Als Objekt-Container dienen der Vector und die Hashtable, das Collection-Framework wird nicht unterstützt.
- Die Netzwerkverbindung ist bei der Connected *Limited* Device Configuration nicht permanent. Die Verbindung kann zeitweise unterbrochen sein bzw. nur mit niedriger Bandbreite zur Verfügung gestellt werden.
- Das extrem niedrige Speichervolumen eines Kleingeräts im Vergleich zu einer Workstation oder einem Server verlangt möglichst kompakte Implementierungen. So werden häufig zum Einsparen von Klassen Listener-Schnittstellen bereits in den UI-Klassen implementiert. Die virtuelle Maschine der Laufzeitumgebung (die so genannte »KVM«, wobei das »K« für Kilobyte steht) unterstützt Hotspot nicht und ist in der Speicherverwaltung aufwändig. Die Wiederverwendung von Objekten ist somit wichtig.

Insbesondere das geringe Speichervolumen verführt zu einem monolithischen Programmierstil, der nicht leicht wieder verwendbare Programme generiert. Nachdem insbesondere auf dem heterogenen und sich schnell ändernden Kleingerätemarkt eine möglichst flexible Struktur notwendig ist, ist beim Design der Anwendung auf die konsequente Verwendung von Pattern zu bestehen. Bei erkannten Problemen ist erfahrungsgemäß ein vernünftiger modularer Aufbau der Anwendung die notwendige Grundlage für den Einsatz effizienter Optimierungsstrategien.

Obwohl die eingeschränkte Programmbibliothek die Wiederverwendung bereits vorhandener Klassenbibliotheken zumindest erschwert, ist gerade bei der Programmierung von Kleingeräten auf eine strenge Schichtentrennung zu achten, da zu erwarten ist, dass sich die Mächtigkeit der Geräte und damit der Umfang der Profile schnell ändern wird. Weiterhin bieten viele Hersteller eigene Entwicklungstools und Bibliotheken an, um geräteabhängige Erweiterungen zu unterstützen. Bei einem Wechsel des Gerätes müssen diese Programmteile modular austauschbar sein.

1.14.4 Praxis

Es wird deshalb das »Client JavaBeans«-API in einer für die CLDC angepassten Form verwendet. Die folgenden Änderungen müssen durchgeführt werden:

- Die Schnittstellen des CJB-APIs müssen von den Collection-Schnittstellen List und Map auf Vector und Hashtable umgestellt werden. Dadurch ist es notwendig, die Schnittstellen in ein eigenes Paket (com.hotspots.javax.microedition.cjb) auszulagern, da ansonsten die von der Spezifikation geforderte Kompatibilität gebrochen wird (Eine MIDP-Anwendung muss auch gegen die unveränderten Standard-Klassen der J2SE kompiliert werden können).
- Der ClientContext liefert als Logger eine eigene schlanke Implementierung, da das Logging-API der J2SE nicht zur Verfügung steht.
- Das Kleingerät wird eine andere Implementierung des Client Containers benötigen. Dieser wird aufgrund des fehlenden java.lang.reflect-APIs die dynamische Umhüllung der ClientBeans nicht implementieren können.
- Die Konfiguration einer MIDP-Applikation erfolgt im Java Deskriptor. Damit wird eine eigene Implementierung des Property Managers notwendig.
- Die UI-Elemente erzeugen keine Events aus dem Paket java.awt.event, sondern verwenden bereits ein Command-Objekt (javax.microedition.lcdui.Command). Zum Ansprechen des Controllers muss eine eigene Adapter-Klasse erzeugt werden.

Der deskriptive Programmierstil wird im MIDP durch den »Java Application Descriptor« gefördert. Dieser Deskriptor ist eine einfach lesbare Datei mit der Endung »jad« und wird vom Kleingerät zusätzlich zum eigentlich funktionellen Java Archiv geladen. Die JAD-Datei enthält, wie der Name bereits vermuten lässt, programmbeschreibende Informationen:

```
MIDlet-1: MIDP-Starfinder, MIDP-Starfinder.png,
com.hotspots.starfinder.microedition.MIDPStarfinder
MIDlet-Jar-Size: 80206
MIDlet-Jar-URL: MIDP-Starfinder.jar
MIDlet-Name: MIDP-Starfinder
MIDlet-Vendor: Sun Microsystems
MIDlet-Version: 1.0
com.hotspots.javax.microedition.client.view.CJBFormDispatcher:
com.hotspots.javax.microedition.client.view.CJBFormDispatcherImpl
exceptionhandler.class:
com.hotspots.javax.microedition.util.DefaultExceptionHandler
genericfactory.class: com.hotspots.javax.microedition.util.DefaultGenericFactory
propertymanager.class: com.hotspots.javax.microedition.util.DefaultPropertyManager
starfinder.dispatcher:
com.hotspots.javax.microedition.client.view.event.ControlDispatcherImpl
...
```

Von der Idee her ist der JAD-Deskriptor nichts anderes als eine externe Manifest-Datei und ist deshalb auch in deren Stil geschrieben. Es ist jedoch problemlos vorstellbar, dass sich in naher Zukunft, wenn XML-Parser zur Verfügung stehen, daraus eine DTD für das MIDP entwickeln wird.

Die grafische Benutzeroberfläche einer MIDP-Anwendung wird mit Hilfe von UI-Klassen aus dem Paket javax.microedition.lcdui realisiert. Es steht ein Satz schwergewichtiger Komponenten und Container zur Verfügung, mit denen Oberflächen aufgebaut werden. Das Layout der Anwendung wird ausschließlich von der MIDP-Implementierung des Kleingerätes übernommen. Im Gegensatz zu einer »echten« GUI-Anwendung wird nur ein einziger Event-Typ, nämlich als Reaktion auf eine Aktionstaste, ein »Command«, generiert. Damit ist das »Look & Feel« einer MIDP-Anwendung identisch mit einer vom Browser gerenderten HTML-oder WAP-Seite: Der Seitenwechsel erfolgt über die Auswahl von Schaltflächen, nicht durch einen Mausklick auf einen Frame oder Reiter. Damit benötigt aber auch die MIDP-Anwendung eine »Service to Worker«-Implementierung. Dies ist im Übrigen durchaus ein Kriterium für die Wahl der eingesetzten Technologien: Ist der Ablauf der Seiten bereits auf dem Webserver definiert, ist es wahrscheinlich konsequenter, neben den reinen Daten auch das Aussehen und die wahrscheinlich einfache Funktionalität mit Hilfe von WAP-Seiten zu übertragen. Das WAP-Portal benötigt nur einen WAP-Filter. Sollen jedoch ausschließlich Daten übertragen werden, kann dann eine MIDP-UI die Darstellung übernehmen.

Im Rahmen der ClientBean-Architektur werden die Views durch das Interface `com.hotspots.javax.cjb.ResultDestination` repräsentiert. Die Kontrolle über die Auswahl des Views liegt somit in der Implementierung des ResultDestinationHome-Interfaces. Auswahl, Instanzierung und Caching der View-Komponente ist damit Aufgabe des Client-Containers, der die benötigten Konfigurationen aus der JAD-Datei liest. Als Schlüssel dient wie üblich ein ClientKey:

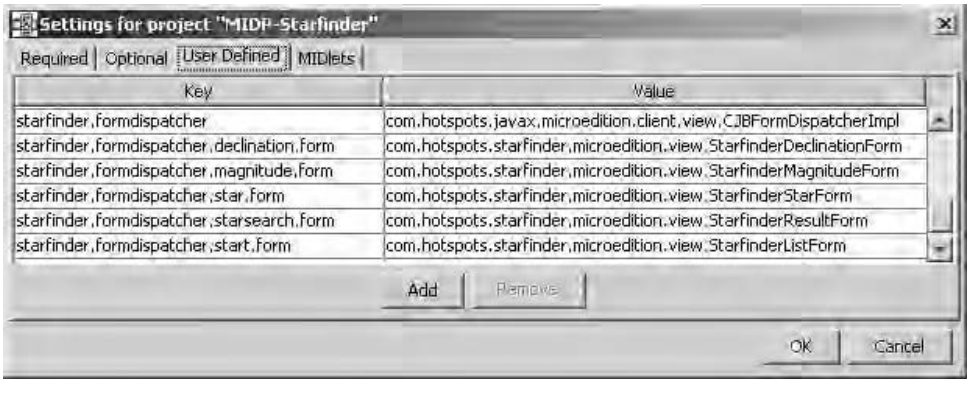


Abbildung 1.53: Die Konfiguration einer J2ME-Anwendung

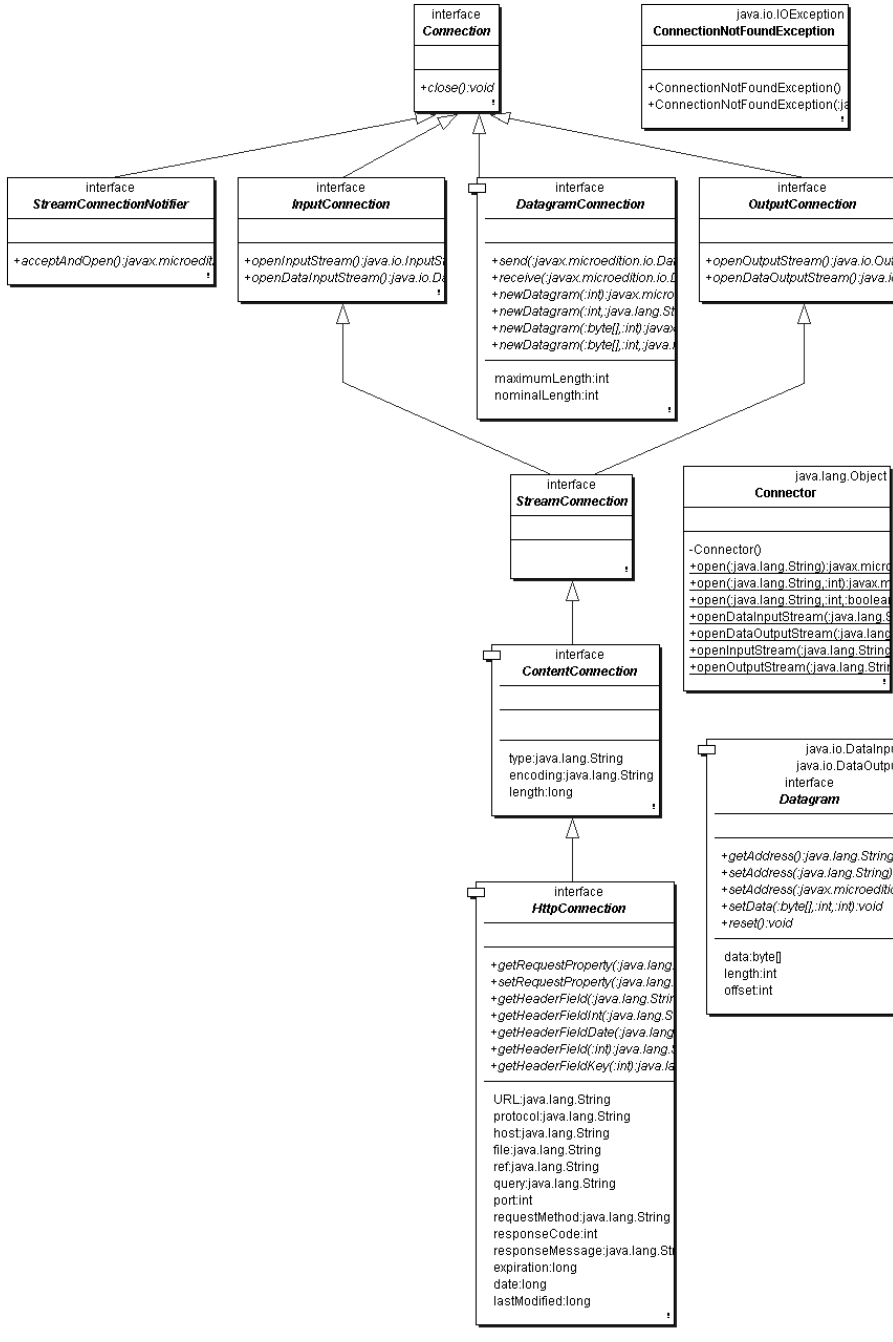


Abbildung 1.54: Das Connection-Framework der Micro-Edition enthält auch das Interface HttpConnection

Der Zugriff auf den Applikationsserver benötigt eine http-Verbindung. Das MIDP enthält zur Unterstützung dieses Protokolls als Bestandteil des javax.microedition.io-Paketes die Schnittstelle HttpConnection. Die HttpConnection wird vom generischen Connector-Framework aufgebaut.

Zur Schonung der beschränkten Ressourcen des Kleingerätes bietet sich hier ein separat laufender Worker-Thread an, der http-Anfragen sequenziell abarbeitet und pro MIDP-Anwendung verwendet wird. Der HttpWorker ist unabhängig vom ClientBean-Framework konzipiert und benötigt bei seiner Instanzierung die URL (in der Regel ein Servlet) in Form eines Strings. Der javax.microedition.io.Connector erzeugt daraus eine http-Verbindung, in die ein Request-String geschrieben wird. Die Antwort wird ebenfalls als String interpretiert und einer Implementierung der HttpWorkerListener-Schnittstelle übergeben:

```
lConn = (HttpConnection)Connector.open(cUrl);
lConn.setRequestMethod(HttpConnection.POST);
lConn.setRequestProperty("Content-Length", Integer.toString(pRequest.length()));
lOut = lConn.openOutputStream();
int lRequestLength = pRequest.length();
for (int i = 0; i < lRequestLength; ++i){
 lOut.write(pRequest.charAt(i));
}
lIn = lConn.openInputStream();
StringBuffer lResponseBuffer;
long lResponseLength = lConn.getLength();
if (lResponseLength > 0){
 lResponseBuffer = new StringBuffer((int)lResponseLength);
}
else{
lResponseBuffer = new StringBuffer();
}
int lChar;
while ((lChar = lIn.read()) != -1){
 lResponseBuffer.append((char)lChar);
}
lResponseStr = lResponseBuffer.toString();
if (lHasError){
 pListener.handleHttpError(lErrorStr);
}
else{
 pListener.handleHttpResponse(lResponseStr);
}
```

Zwischen der MIDP-Anwendung und den serverseitigen Programmen muss in der Regel noch ein anwendungsspezifisches Protokoll definiert werden. Die Übergabeparameter können zwar noch als allgemeine Key-Value-Paare mittels http-POST übertragen werden, jedoch sollte der Rückgabestring keine Format-Anweisungen enthalten. Stattdessen sollten nur die puren Daten übermittelt werden, um möglichst wenig Daten

über die Netzverbindung übertragen zu müssen. Diese Aufgabe kann das Servlet selbst übernehmen, da im Gegensatz zu browser-basierten Anwendungen keine Render-Informationen übernommen werden müssen. Sinnvollerweise übernimmt diese Aufgabe jedoch eine eigene austauschbare Hilfsklasse, die beispielsweise die Properties des Value Object (häufig ein JavaBean) ausliest und diese in den Ausgabestream schreibt. Damit können problemlos Anforderungen wie Minimierung des auszutauschenden Datenvolumens und Verschlüsselung integriert werden. Es genügt aber auch ein simpler »PropertyWriter« oder, momentan aber nur auf Server-Seite, die Verwendung der »Long Term Persistence« für JavaBeans.

Diese »Hilfsklasse« ist aber natürlich nichts anderes als eine Java Server Page, die die Value Objects auf Server-Seite in Zeichenketten zerhackt.

Bei der Programmierung einer MIDP-Anwendung ist, wie bereits erwähnt, ein hohes Augenmaß auf die Austauschbarkeit der Komponenten zu legen. Aufgrund des Fehlens des Reflection-APIs müssen die Factories der J2ME einfach gehalten werden und können nur den parameterlosen Konstruktor aufrufen:

```
package com.hotspots.javax.microedition.util;
import com.hotspots.java.util.*;
public class DefaultGenericFactory implements GenericFactory {
 public Object create(String pClassName) throws CreateException{

 try{
 return Class.forName(pClassName).newInstance();
 }
 catch(Exception pException){
 com.hotspots.java.util.logging.Logger.getAnonymousLogger().throwing(pException);
 throw new CreateException(pException.getMessage());
 }

 }
```

Die ClientBeans verwenden zum Finden von Implementierungen häufig den voll qualifizierten Klassennamen als Bestandteil des Suchschlüssels. Nachdem die ».class«-Syntax vom J2ME-Compiler nicht verstanden wird, übernimmt eine Hilfsklasse diese Aufgabe:

```
package com.hotspots.javax.microedition.util;
import com.hotspots.java.util.*;
public final class ClassUtil{
 private static final Class DEFAULT = (new Object()).getClass();
 private ClassUtil(){}
 public static Class getClass(String pClassName){
 try{
 return Class.forName(pClassName);
 }
```

```
catch(Exception pException){
return DEFAULT;
}
}
}
```

Diese Klasse benötigt im Übrigen keinerlei eigene Cache-Funktionalität, da diese bereits innerhalb der Klasse `java.lang.Class` implementiert ist.

Hauptprobleme einer J2ME-Anwendung sind der limitierte Speicher des Kleingerätes, der nach der realen Installation schnell zu »OutOfMemoryError«-Meldungen führen kann. Es ist deshalb interessant, die hier vorgestellte Architektur bezüglich der Ausführung auf einem konkreten Gerät zu testen. Das von Sun kostenlos zur Verfügung gestellte »Wireless Toolkit« sowie die Entwicklungsumgebungen kommerzieller Hersteller wie Siemens oder Nokia laufen ja in der Regel auf einem mächtigen PC mit fast beliebigen Speicherressourcen und sind damit als realer Maßstab nicht zu verwenden. Die vorgestellten Programme wurden deshalb alle auf drei kommerziellen Produkten installiert, und zwar

- Siemens SL45i
- Nokia Communicator
- Palm

Die Demonstrationsanwendung, der »Smart Ticket«, ein Kartenreservierungsprogram, läuft nach der Installation der Enterprise-Komponenten unter der Emulation sofort problemlos. Das MID-Profil gliedert sich nahtlos in das J2EE –Programmierungsmodell ein. Das Midlet kommuniziert über eine javax.microedition. HttpConnection mit einer einfachen JSP, die eine Zeichenkette mit den Daten zurück sendet. Soll zwischen dem Midlet und dem Webserver eine Session aufgebaut werden, kann diese einfach im Kontext des Midlets abgelegt werden.

```
<%StarDelegateIF delegate = null;
if((delegate = (StarDelegateIF)session.getValue("starsearch.delegate")) == null){
JspWriter jsp_out = pageContext.getOut();
out.print("jsessionid");
out.print(session.getId());
out.print('#');
DelegateSessionManager manager = new DelegateSessionManager();
manager.setHttpSession(session);
StarDelegateFactory factory = new
StarDelegateFactory(StarSearchConst.DELEGATE_CLASS);
manager.storeStarDelegateIF(delegate = factory.getStarDelegateIF(false));
}
delegate.searchByMagnitude(Float.parseFloat(request.getParameter("magnitude")));
StarVO[] voArray = delegate.getNext(1);
if (voArray  != null && voArray.length > 0){
```

```
    StarVO vo = voArray[0];
    JspWriter jsp_out = pageContext.getOut();
    jsp_out.print(vo.getCatalogPrefix());
    jsp_out.print("#");
    jsp_out.print(vo.getZoneSign());
    jsp_out.print("#");
…
}else{
  pageContext.getOut().print("No Stars!");
}
%>
```

Im Midlet:

```
private String checkSession(String data){
if (data.startsWith(SESSION_ID)){
   int end = data.indexOf(DELIMITER);
   InitialClientContextFactory.getInitialClientContext().bind(SESSION_ID,
";jsessionid=" + data.substring(SESSION_ID.length(), end));
   return data.substring(end+1, data.length());
}
return data;
}
…
```

2 Die Präsentationsschicht

2.1 Anbindung von EnterpriseBeans

2.1.1 Problemstellung

Wie werden EnterpriseBeans aus Web-Komponenten heraus angesprochen?

2.1.2 Technischer Hintergrund

Servlets und JavaServer Pages übernehmen im server-seitigen MVC-Modell bekanntermaßen die Rolle des Controllers und der View, während die EnterpriseBeans dem Datenmodell entsprechen. Zur Laufzeit der Anwendung müssen die einzelnen Komponenten durch Referenzen verbunden sein, die im JNDI-Kontext des Applikationsservers gefunden werden. Auch Servlets und JavaServer Pages referenzieren als Client eine EnterpriseBean durch einen JNDI-Lookup auf das Home-Interface und anschließenden Aufruf einer create()-Methode. Die EJB-Spezifikation bietet dafür ab der Spezifikation 2.0 zwei grundsätzlich unterschiedliche Varianten von Komponenten-Interfaces an:

- Erbt das Komponenten-Interface von `javax.ejb.EJBObject` und damit indirekt von `java.rmi.Remote`, so werden Stubs zur Kommunikation über ein Netzwerkprotokoll verwendet. Parameter und Rückgabewerte werden entweder als Kopie oder serialisierte Stubs übertragen. Der Servlet Container und der EJB-Container können sich dann auf verschiedenen Maschinen befinden.
- Falls sich die Container innerhalb der gleichen Virtuellen Maschine befinden, können die lokalen Komponenten-Interfaces verwendet werden. Dann wird der gemeinsam zugreifbare Heap-Speicher der Virtuellen Maschine zum Austausch der Referenzen verwendet.

2.1.3 Lösung

- Befinden sich Servlet und EJB innerhalb der gleichen Applikation, können laut J2EE-Spezifikation auch hier lokale Interfaces verwendet werden.

2.1.4 Performance

- Es stellt sich nun aber natürlich sofort die Frage, was die lokalen Komponenten-Interfaces im Vergleich zu den Remote-Interfaces an Performance-Vorteilen bringen. Zu diesem Zweck dient ein einfaches Beispiel (ein Logger), mit dem Meldungen sowohl über ein Web-Frontend als auch über Standalone-Anwendungen an den Applikationsserver gesendet werden können. Logger-Servlet und das Logger-EJB befinden sich in der gleichen J2EE-Applikation. Das Servlet dient als einfacher Controller mit drei unterschiedlichen Modi:
- NONE: Keine Delegation als Offset-Messung,
- LOCAL: Delegation an das lokale,
- REMOTE: Delegation an das remote Komponenten-Interface des LoggerBeans.

Für die Referenz-Implementierung von Sun und den JBoss ergaben sich für 1.000 Zugriffe die folgenden Zeiten:

	Referenz-Implementierung	Normiert	Weblogic 7.0	Normiert	JBoss	Normiert
NONE	930	100%	880	100%	880	100%
LOCAL	950	102%	960	110%	1000	114
REMOTE	1620	174%	1290	147	1020	116

Tabelle 2.1: Vergleich zwischen den Aufrufzeiten eines Servlets bei der Delegation an eine EJB über lokale und remote Komponeten-Interfaces

Die Unterschiede zwischen der Referenz-Implementierung und dem JBoss sind evident: Während beim JBoss kaum ein Unterschied zu bemerken ist (etwa 2% Verlängerung), sind bei der Referenz-Implementierung 70% Mehraufwand zu beobachten. Beim WebLogic sind es immerhin noch 47%. Bei noch genauerer Betrachtung fällt auch auf, dass der JBoss bei der lokalen Variante zunächst der langsamste ist. Diese Effekte müssen genauer analysiert werden.

Die Langsamkeit des JBoss in der lokalen Variante lässt sich erst einmal einfach erklären: Auch die lokalen Interfaces-Implementierungen der einzelnen Applikationsserver benötigen selbstverständlich Stub-Klassen zur Umhüllung der funktionellen Bean-Instanz. Die Referenz-Implementierung generiert hierfür ebenso wie Weblogic eine spezielle Hilfsklasse, während der JBoss einen generischen Invocation-Handler verwendet. Dieser ist durch die Verwendung des Reflection-API etwas langsamer. Dass sich im Folgenden die Antwortzeiten kaum ändern, lässt sich auch durch brillanteste Algorithmen innerhalb der JBoss-Klassen nicht mehr erklären. Stattdessen wird offensichtlich ein Trick verwendet: Befinden sich die beiden J2EE-Komponenten in der gleichen Applikation, so wird auch bei der Angabe eines remote Komponenten-Interfaces

ein lokaler Stub verwendet! Dies lässt sich leicht beweisen, indem als Log-Meldung ein spezielles serialisierbares Log-Objekt verwendet wird, in dem die Serialisierungsmethoden `readObject` und `writeObject` überschrieben sind: Beim JBoss werden diese Methoden im Gegensatz zur Referenz-Implementierung und WebLogic nie aufgerufen, da die Objekte bei lokaler Parameterübergabe natürlich nicht in ein Byte-Array umgewandelt werden müssen.

Diese Optimierung hat jedoch ein gravierendes Problem: Sind Übergabeparameter nicht unveränderbar, so kann sich die Anwendung im JBoss gänzlich anders verhalten als in anderen Applikationsservern, da sich eine Änderung des Parameters innerhalb der EJB in die Präsentationsschicht durchschlagen wird. Dies kann zu ungewollten Nebeneffekten führen.

2.2 Ablage von Objektreferenzen

2.2.1 Problemstellung

In welchen Objekten können Informationen der Web-Anwendung abgelegt werden?

2.2.2 Technischer Hintergrund

Die J2EE-Spezifikation definiert für Servlet-Komponenten und JSPs vier standardisierte Möglichkeiten, Referenzen auf Objekte übergreifend abzulegen:

- Das Servlet-Request-Objekt,
- Implementierungen des Interfaces `javax.servlet.ServletContext`,
- von `javax.servlet.http.HttpSession` und
- von `javax.servlet.jsp.PageContext`.

Diese Interfaces definieren alle die Methoden

```
Object getAttribute(String name)
void setAttribute(String name, Object value)
```

und sind somit geeignet, beliebige Objektreferenzen ablegen zu können. Diese können dann von allen beteiligten Web-Komponenten gelesen werden, ohne als Parameter übergeben werden zu müssen.

Weiterhin existiert noch der JNDI-Kontext des Applikationsservers, in dem ebenfalls Objektreferenzen gebunden werden können.

Als dritte Alternative existieren dann natürlich rein anwendungsspezifische Varianten: Singletons, Zugriff auf Persistenz-Dienste eines Frameworks oder spezielle eigene Lösungen.

Diese Vielzahl an potenziellen Möglichkeiten führt immer wieder zu Verunsicherungen darüber, welche Möglichkeiten eingesetzt werden sollen.

2.2.3 Lösung

Verwendung des Request-Objektes

Nachdem das Request-Objekt ausschließlich während der Ausführung der service()-Methode des zentral aufgerufenen Servlets existiert, sollte dieser ausschließlich für kurzlebige Objekte verwendet werden, die nach Ende des Requests nicht mehr benötigt werden.

Verwendung des Servlet-Kontextes

Der Servlet-Kontext ist die direkte Schnittstelle zum Servlet-Container. Dieser existiert exakt einmal pro

- Cluster,
- Virtueller Maschine und
- Web-Applikation.

Eine Verteilung über den Cluster ist dabei nur dann möglich ist, wenn die Web-Applikation im Deskriptor als »distributable« gekennzeichnet ist. Was den Servlet-Kontext interessant macht, ist, dass im API zwei Listener-Klassen Notifizierungs-Mechanismen vorhanden sind:

- `javax.servlet.ServletContextListener` und
- `javax.servlet.ServletContextAttributeListener`.

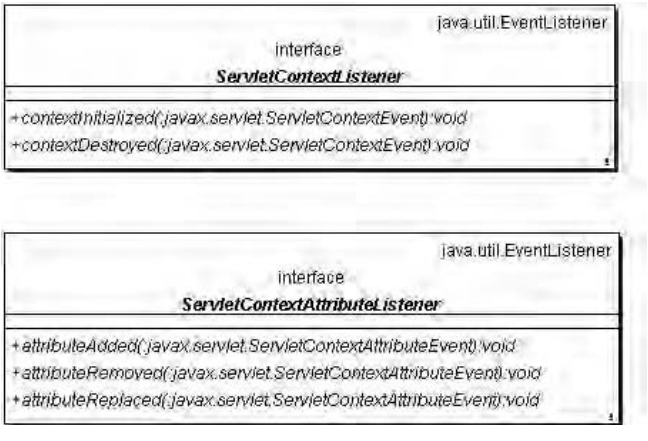


Abbildung 2.1: Die Listener-Schnittstellen für den ServletContext

Der `ServletContextListener` legt den Lebenszyklus des Containers fest und ist damit perfekt geeignet, Initialisierungsvorgänge und Aufräumarbeiten für die gesamte Web-Applikation durchzuführen, also z.B. Erzeugen des Service-Locators, Aufbau von Datenbankverbindungen etc.

Ein `ServletContextAttributeListener` reagiert auf das Hinzufügen, Ändern oder Löschen von Attributen. Damit können event-gesteuerte Abläufe definiert werden.

Wichtig bei der Verwendung dieser beiden Schnittstellen ist die Tatsache, dass Implementierungen nicht dynamisch über Methoden wie »`addServletContextListener(…)`« hinzugefügt werden, sondern nur statisch im Deskriptor eingetragen werden können.

Insgesamt verhält sich der Servlet-Context, wenn eine Cluster-Umgebung vernachlässigt werden kann, wie ein Singleton für die gesamte Web-Applikation. Nachdem dieser Kontext in jedem Falle pro Virtueller Maschine existiert, können damit auch bedenkenlos Referenzen auf große Objekte und nicht-serialisierbare Typen abgelegt werden.

Verwendung der Http Session

Die Schnittstelle `HttpSession` ist in ihrer Verwendung auf eine robuste Session-Verwaltung ausgelegt. Das Session-Objekt existiert:

- trivialerweise einmal pro Session, diese muss also erst einmal angelegt werden
- exakt einmal pro Cluster.

Der letzte Punkt ist der zentrale Unterschied zum Servlet-Kontext: Sessions sind cluster-übergreifend.

Daraus ergibt sich aber bereits eine Einschränkung in der Verwendung des Session-Kontextes: Die Referenzen müssen serialisierbar sein, um eine Migration der Session von einem Cluster-Rechner zum nächsten zu ermöglichen. Weiterhin dürfen die hinzugefügten Attribute eine vernünftige Größe nicht überschreiten, um bei einer Passivierung oder Migrierung der Session den Web-Container nicht zu überfordern.

Auch für die Session existieren Listener (siehe Abbildung 2.2)

Der `HttpSessionAttributeListener` wird analog dem `ServletContextAttributeListener` verwendet.

Die beiden anderen Schnittstellen erweitern den Lebenszyklus der Session bezüglich der robusten Session-Verwaltung: Der `HttpSessionActivationListener` definiert Callback-Methoden, die der Servlet-Container aufrufen kann, wenn

- die Session migriert oder
- passiviert wird.

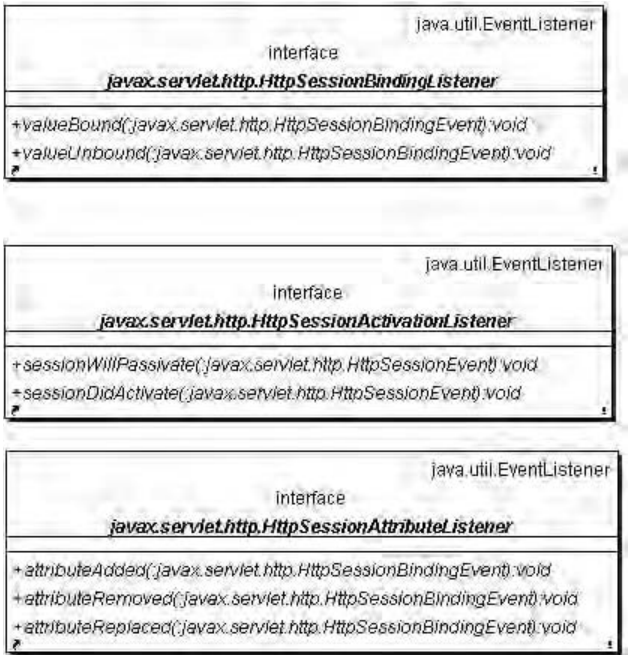


Abbildung 2.2: Die Listener-Schnittstellen für die javax.servlet.http.HttpSession

Nicht-serialisierbare Objekte können bei der Aktivierung der Session im neuen Cluster-Bestandteil wieder aufgebaut werden.

Mit der letzten Schnittstelle, dem `HttpSessionBindingListener`, kann der Anwendungsprogrammierer auf das Ende einer Session reagieren. Gebundene Werte werden ja automatisch aus der Session entfernt, wenn diese invalidiert wird, oder ein Timeout stattgefunden hat.

Verwendung des PageContext

Dieser Kontext bleibt JSP-Entwicklern vorbehalten. Ein JSP-Tag legt im Page-Kontext Objektreferenzen ab, die das umhüllende Tag dann auslesen kann. So kann die Darstellung eines Vektors mit dem forEach-Tag der *JavaServer Pages Standard Tag Library* (JSTL) innerhalb einer JSP geschrieben werden als:

```
<html><head><title>Scriptlet JSP</title></head><body>
<c:forEach var="element" items="${requestScope.vector}">
<c:out value="${element}"></c:out></td>
</c:forEach>
```

Innerhalb des Quellcodes der forEach-Implementierung (bzw. in der verwendeten Superklasse `javax.servlet.jsp.jstl.core.LoopTagSupport`) werden in der Methode

```
exposeVariables()
```

dann die aktuellen Werte in den Page-Kontext gestellt:

```
private void exposeVariables() throws JspTagException{
   if(itemId != null)
      if(getCurrent() == null)
         super.pageContext.removeAttribute(itemId, 1);
       else
         super.pageContext.setAttribute(itemId, getCurrent());
       if(statusId != null)
          if(getLoopStatus() == null)
             super.pageContext.removeAttribute(statusId, 1);
          else
             super.pageContext.setAttribute(statusId, getLoopStatus());
    }
```

In diesem Beispiel wird unter dem Schlüssel »`element`« (angegeben im forEach-Attribut »`var`«) der Inhalt des Vektors an der aktuellen Position gestellt. Das Out-Tag evaluiert den Variablennamen »`${element}`« und holt sich aus dem Page-Kontext den Wert. Dieser wird in den JspWriter geschrieben.

JNDI-Kontext und eigene Lösungen

Der JNDI-Kontext des Applikationsservers legt Informationen übergreifend ab, die Referenzen sind zumindest für alle J2EE-Komponenten des Applikationsservers verfügbar. Ist der JNDI-Kontext pro Cluster wie bei der HTTP-Session definiert, ist auch hier darauf zu achten, dass gebundene Referenzen serialisiert werden müssen. Im Unterschied zur Session, bei denen die Serialisierung nur im Falle der Passivierung oder der Session-Migration notwendig ist, wird beim Binden in den JNDI-Kontext stets serialisiert. Die Verwendung des JNDI-Kontextes wird somit sicherlich langsamer sein als die bisher vorgestellten Varianten.

Für eigene Lösungen bleibt somit nicht mehr allzu viel Bedarf. Singletons können selbstverständlich weiter benutzt werden. Lebenszyklus und Gültigkeitsbereich sind für Singletons die gleichen wie für den Servlet-Kontext. Bei der Anbindung an eigene Frameworks können die Listener-Schnittstellen sinnvoll verwendet werden.

2.2.4 Performance

	Referenz-Implementierung [msec]	Normiert	JBoss [msec]	Normiert
Ohne Setzen von Attributen, ohne Erzeugung der Session	1140	64%	1060	90%
Ohne Setzen von Attributen	1.790	100%	1.180	100%
Request	1.950	109%	1.260	107%
Servlet-Kontext	1.850	103%	1.230	104%
Session-Kontext	1.950	109%	1.340	114%
JNDI-Kontext	850.000	-	8.500	720%
Singleton	1.800	101%	1.210	103%

Tabelle 2.2: Laufzeit des Setzens von Attributen in den verschiedenen Kontexten. Gemessen wurden jeweils 1.000 Client-Requests, wobei bei jedem Request 100 Attribute gesetzt und in einem weiteren Servlet gelesen wurden.

Die obere Tabelle zeigt die Laufzeitunterschiede der verschiedenen Varianten auf. Am schlechtesten schneidet hierbei der JNDI-Kontext ab. Dies ist an Hand des oben Erläuterten nicht weiter verwunderlich.

Die anderen Varianten unterscheiden sich im Laufzeitverhalten dann zwar weniger auffällig, aber immerhin noch deutlich messbar. Die Singleton-Variante schneidet am besten ab, dicht gefolgt vom Servlet-Kontext. Diese beiden Varianten sind aber nicht für den Cluster geeignet. Das Halten der Informationen im Request-Objekt und Session-Kontext ist dagegen deutlich langsamer. Beide müssen die Objekte in temporären Objekten halten, die finalisiert werden. Servlet- und Session-Kontext müssen weiterhin noch beim Binden die registrierten Listener informieren.

Insgesamt lässt sich aus den gemessenen Werten ablesen, dass sich aus Performance-Gründen der Einsatz von Singletons nicht lohnen wird; der Servlet-Kontext ist eine vollwertige Alternative. Nachdem der Einsatz von Singletons in einer Server-Umgebung, in der notwendigerweise eine Unzahl von Virtuellen Maschinen und Klassenladern miteinander in Kontakt treten, fehleranfällig ist, sollte darauf verzichtet werden. »Echte« Singletons in verteilten Anwendungen müssen über Stubs angesprochen werden, die im JNDI-Kontext abgelegt werden.

Ganz wichtig in diesem Zusammenhang ist aber, dass bei der Verwendung des Servlet-Kontexts nicht vergessen wird, unnötige Referenzen wieder zu entfernen. Es ist alleinige Aufgabe des Anwendungsprogrammiers, removeAttribute aufzurufen. Unterbleibt dies, hat die Web-Applikation ein Memory Leak. Die gleiche Argumentation gilt natürlich auch für Singletons.

Erst die Verwendung von Request oder Session umgeht dieses Problem auf Kosten einer Performance-Einbuße. Die Entscheidung, welches Objekt die Attribute aufnehmen soll, ist rein fachlich begründet.

2.3 Deaktivierte Cookies

2.3.1 Problemstellung

Wie kann die Web-Anwendung erkennen, ob der Benutzer im Browser Cookies deaktiviert hat?

2.3.2 Technischer Hintergrund

Web-Server erkennen automatisch an Hand der Browser-Kennung, ob dieser Cookies unterstützt oder nicht. Davon abhängig schaltet der Server dann vom Session-Cookie (Name: »JSESSIONID«) um auf URL Rewriting (der URL aller Links wird »;jsessionid=…« angehängt). Unterstützt der Browser zwar Cookies, hat der Benutzer aber deren Verwendung deaktiviert, kann der Web Server dies jedoch nicht erkennen. Dem Response wird zwar ein Cookie mitgegeben, das aber vom Browser nicht geschrieben wird. Somit werden ständig neue Sessions generiert.

2.3.3 Lösung

Es wird ein »Cookie-Detektor« eingesetzt. Dieses ist ein Servlet, das einfach ein Cookie erzeugt und an ein weiteres Servlet delegiert:

```
public class CookieDetectorServlet extends HttpServlet {
   public static final String COOKIE_CHECKER = "CookieChecker";
   protected void doGet(
      HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      response.addCookie(new Cookie("supportsCookies", ""));
      response.sendRedirect(
         getServletContext().getInitParameter(COOKIE_CHECKER));
   }
}
```

Dieses prüft einfach, ob das Servlet geschrieben wurde und aktiviert gegebenenfalls das URL-Rewriting:

```
Public class CookieCheckerServlet extends HttpServlet {
   public static final String SESSION_SERVLET = "SessionServlet";
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
```

```
    Cookie cookie = CookieUtil.getCookie(request, "supportsCookies");
    HttpSession session  = request.getSession(true);
    if (cookie ==  null){

response.sendRedirect(getServletContext().getInitParameter(SESSION_SERVLET)+";jses
sionid=" +session.getId());
    }
    else{

response.sendRedirect(getServletContext().getInitParameter(SESSION_SERVLET));
    }
  }
}
```

2.4 Die Performance der JavaServer Pages Standard Tag Libraries

2.4.1 Problemstellung

Die neuen JavaServer Pages Tag Libraries vereinfachen die Entwicklung dynamischer Web-Seiten für den HTML-Designer. Die Notwendigkeit, durch Scriptlets Aufbereitungslogik wie Schleifen oder Abfragen zu kodieren, entfällt durch JSP-Tags. Es stellt sich jedoch die Frage nach der Performance der so erstellten Seiten. Können die neuen Tags Scriptlets ersetzen?

2.4.2 Technischer Hintergrund

Mit der JSTL-Bibliothek integriert Sun in die J2EE einen Satz geläufiger Tags für JSP-Seiten. Die Bibliothek lässt sich in die vier Bereiche

- Core mit Iterationen, Schleifen und Ausdrücken,
- XML zum Parsen von Dokumenten,
- Internationalisierung und
- SQL-Abfragen

unterteilen. Damit kann in JSP-Seiten einfache Präsentationslogik integriert werden, ohne Kenntnisse von Java besitzen zu müssen. Eine Schleife zur Darstellung eines Vektors kann durch ein Scriptlet:

```
<html><head>
<title>Scriptlet JSP</title>
</head><body>
<%      Vector vec = (Vector)request.getAttribute("vector");
```

```
int vecSize = vec.size();
for (int i = 0; i < vecSize; i++){
     out.write(vec.get(i).toString());
}%>
```

oder alternativ vom HTML-Designer mit Tags realisiert werden, wie im vorhergehenden Abschnitt gezeigt wurde.

Was sind die Unterschiede dieser Verfahren?

2.4.3 Performance

Als Referenz dient eine einfache HTML-Seite mit der 100fachen Wiederholung der Zeichenkette » Test «. Diese Seite wird über einen lokal installieren Apache Tomcat auf drei Arten übermittelt: direkt als statische Seite, erstellt über eine JSP mit Scriplets und als letztes über das forEach-Tag der JSTL.

Methode	Zeit für 1.000 Zugriffe in msec
Direkt	470
Scriptlet	980
forEach-Tag	1.300

Tabelle 2.3: Die Performance des forEach-Tags im Vergleich zur direkten Erzeugung und zu Skriptlets

Das forEach-Tag ist deutlich langsamer als die Scriptlet-Variante. Kritischer wird der Effekt bei komplexeren Seiten mit intensiver Verwendung von JSP-Tags. So ändern sich die Zahlen für den Aufbau einer Tabelle mit 50 Spalten und 20 Zeilen unter Verwendung zweier if-Tags bereits auf die folgenden Zeiten:

Methode	Zeit für 1.000 Zugriffe in msec
Direkt	480
Scriptlet	980
forEach- und if-Tags	3.200

Tabelle 2.4: Die Performance mehrerer JSTL-Tags (forEach, if) im Vergleich zur direkten Erzeugung und zu Skriptlets

Während das Scriptlet bei einer Erhöhung der Komplexität der Ausgabe praktisch keinen zusätzlichen Zeitaufwand bedingt (es werden zusätzlich nur zwei einfache if-Anweisungen benötigt, für die Virtuelle Maschine kaum Aufwand), steigt bei den Tags der Aufwand überproportional an, was bei komplexen Seiten fühlbare Performance-Einbußen zur Folge haben und zu wesentlich stärkerer Belastung des Servers führen wird. Weniger ist bei den Taglibs auch aus Performance-Gründen mehr!

2.5 Auswirkungen des SingleThreadModel-Interfaces

2.5.1 Problemstellung

Welche Auswirkungen hat die Implementierung des Interfaces

```
javax.servlet.SingleThreadModel?
```

2.5.2 Technischer Hintergrund

Im Gegensatz zum EnterpriseBean-Container synchronisiert der Servlet-Container den Zugriff auf Servlet-Komponenten nicht automatisch. Zugriffe auf das Servlet werden pro Anfrage häufig in separaten Threads an eine einzige Instanz des Servlets delegiert. Der Entwickler eines Servlets ist selbst dafür verantwortlich, die Methodenaufrufe thread-sicher zu gestalten.

Kann oder will der Entwickler die Synchronisierung nicht selbst übernehmen, kann die Servlet-Klasse das Interface `SingleThreadModel` implementieren. Damit »weiß« der Servlet-Container, dass die Servlet-Instanz nicht thread-sicher ist und synchronisiert den Zugriff automatisch.

Um trotzdem ein performantes Antwortverhalten zu erreichen, ändert der Container einfach seine Strategie: Statt mehrerer Threads wird nun von vorn herein ein Pool von Servlet-Instanzen aufgebaut. Jeder Request wird von einem separaten Objekt bearbeitet.

2.5.3 Performance

Ein Servlet simuliert durch einen `Thread.sleep(1000)` eine lange andauernde Berechnung. Das Servlet wird in zwei unterschiedlichen Ausprägungen implementiert: ohne und mit SingleThreadModel. Die Installation erfolgt im Tomcat 4.0.

Eine Client-Anwendung öffnet in separaten Threads gleichzeitig 200 URL-Verbindungen, gemessen wird die Zeit zwischen dem Starten der Threads und der letzten beendeten Bearbeitung.

Im ersten Falle begnügt sich der Servlet-Container offensichtlich mit einer einzigen Instanz der Servlet-Klasse. Die 200 Zugriffe dauern in der Summe etwa 2.000 msec.

Implementiert die Servlet-Klasse `SingleThreadModel` ändert sich an der Antwortzeit faktisch nichts. Der Servlet-Container instanziert jetzt jedoch 200 Instanzen des Servlets, um alle Anfragen bedienen zu können. Das Interface führt also in erster Näherung nicht zu einer Änderung der Antwortzeit, sondern erhöht den Speicherbedarf auf dem Server.

Starke Auswirkungen zeigen sich jedoch dann, wenn das Servlet einen internen Zustand thread-sicher synchronisieren muss: Die aufrufenden Threads des Servlet-

Containers blockieren, die Anfragen werden damit sequenziell abgearbeitet. Hier ermöglicht erst das SingleThreadModel auf Kosten des Speicherbedarfes eine parallele Verarbeitung. Der Servlet-Container registriert das Blockieren der Threads nicht und kann somit nicht »vernünftig« durch Erzeugung weiterer Servlet-Instanzen das Antwortverhalten optimieren.

2.6 JavaServer Pages zur Konfiguration

2.6.1 Problemstellung

Wie können Konfigurationsdateien zentral auf dem Server administriert und für entfernte Anwendungen zur Verfügung gestellt werden?

2.6.2 Technischer Hintergrund

Für die Konfiguration von Anwendungen enthält das Java API eine Reihe von Hilfsklassen, die eine Vielzahl von Technologien benutzen. Neben einfachen Umgebungsvariablen (`System.getProperty(…)`) werden `java.util.Properties` und `ResourceBundle` sowie die neuen `java.util.prefs.Preferences` angeboten; J2EE-Komponenten lesen durch den JNDI-Kontext indirekt standardisierte Deskriptoren.

Java-Anwendungen verwenden natürlich keine der angeführten Klassen direkt. Es ist eine der ersten Aufgaben selbst des einfachsten Frameworks, einen Property Manager zu definieren, der die verwendeten Technologien kapselt. Es kommt damit nur noch darauf an, ein möglichst einfach administrierbares System aufzusetzen.

2.6.3 Lösung

Für eine zentrale Verwaltung von Properties bietet sich eine Datenbank-Tabelle an. Nachdem JDBC-Treiber jedoch nicht einfach Firewalls durchtunneln können und SQL-Abfragen auch einen beträchtlichen Overhead erzeugen, wird auf dem Server eine JSP eingesetzt.

2.6.4 Praxis

Die JSP wird als J2EE-Komponente definiert und hat damit sofort Zugriff auf eine Data-Source. Die Konfigurationen werden einfach über ein Select-Statement erhalten:

```
<%@ page import="javax.naming.*, java.sql.*, javax.sql.*, java.text.*"%>
<%MessageFormat format = new MessageFormat("{0}={1}");
InitialContext ctx = new InitialContext();
String tableName = (String)ctx.lookup("java:comp/env/tableName");
DataSource src = (DataSource)ctx.lookup("java:comp/env/jdbc/ConfigDB");
```

```
Connection con = src.getConnection();
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("select * from " + tableName);
JspWriter writer = pageContext.getOut();
Object[] params = new Object[2];
 while(rs.next()){
   params[0] = rs.getObject(1);
   params[1] = rs.getObject(2);
   out.write(format.format(params));
   out.write("\n");
}
%>
```

Der Zugriff darauf erfolgt über die URL der installierten JSP:

```
public class JSPPropertyManager implements PropertyManager {
   private Properties props;
   public JSPPropertyManager(URL pUrl) {
   try {
      props = new Properties();
      BufferedInputStream bis =
      new BufferedInputStream(pUrl.openStream());
      props.load(bis);
   }
…
   public Object getProperty(String pPropertyName) throws
   PropertyAccessException {
   try {
      return props.get(pPropertyName);
   } catch (Exception e) {}
```

Die Key-Value-Paare der Konfiguration werden in diesem Beispiel beim Zugriff komplett auf den Client übertragen. Es spricht aber natürlich nichts dagegen, im Request die zu suchenden Schlüssel anzugeben und so die JSP als zentrale Read-Only Hashtable zu verwenden.

2.7 JavaServer Pages zur Erzeugung beliebiger Formate

2.7.1 Problemstellung

Kann eine JavaServer Page auch andere Formate als HTML generieren?

2.7.2 Technischer Hintergrund

Verteilte Anwendungen erhalten die Daten von einem Server in speziell aufbereiteter Form. RMI-Clients erwarten einfache Datentypen oder serialisierte Java-Objekte,

Browser formatierte HTML-Dateien, SOAP-Clients XML-Daten usw. Ein komplettes Framework, das beliebig transformierte XML-Dokumente zur Verfügung stellt, ist Apache Cocoon (http://www.apache.org). Darin sind eine Menge von Ausgabeformaten unterstützt; das Spektrum reicht von (natürlich) HTML und XML über PDF, SVG etc. bis hin zu WML.

Für eigene spezielle Anforderungen können aber auch mit geringem Aufwand JavaServer Pages eingesetzt werden. JSPs sind bisher häufig nur im Einsatz, um komfortabel HTML-Seiten zu generieren. Diese Beschränkung ist aber nicht system-immanent! Eine JSP wird bekanntermaßen intern in ein Servlet umgewandelt, das eigentlich nichts anderes macht, als Daten in einen Ausgabestrom zu schreiben. Das Dateiformat bleibt vollkommen frei.

2.7.3 Praxis

Wir haben bereits im Client-Teil beim Hotspot zum Thema der »Java 2 MicroEdition« gesehen, dass eine JSP eingesetzt werden kann, um einem MIDlet-Client Daten in Form einer einfachen Zeichenkette zu übermitteln. Selbstverständlich kann dies analog auch in anderen Anwendungsszenarien genutzt werden. So kann eine spezielle JSP die Value Objects einer J2EE-Anwendung direkt als Byte-Array an einen Client übermitteln, wenn dieser die serialisierten Objekte darstellen kann. Es bietet sich an, als Kriterium für die Wahl der zu verwendenden JSP einfach einen Http-Parameter mitzuschicken. Das Servlet kann auch automatisch an Hand des http-Headers »accept« entscheiden, in welchem Format die Daten gesendet werden sollen. Ein Servlet-Controller liest den Typ aus dem Request den Typ und lässt einen View Helper die geeignete JSP aussuchen:

```
public class ViewHelper {
   private Context environment;
   private ViewHelper(){
   try {
      InitialContext ctx = new InitialContext();
      environment = (Context)ctx.lookup("java:comp/env");
   } catch (Exception e) {
      e.printStackTrace();
   }
   }
   private static ViewHelper instance = new ViewHelper();
   public static ViewHelper getInstance(){
      return instance;
   }
   public String getViewForMimeType(String pMimeType){
      if (pMimeType == null){
         pMimeType = "default";
      }
```

```
        try {
           Object obj = environment.lookup(pMimeType);
           System.out.println("Object: " + obj);
           return obj.toString();
        } catch (Exception e) {
           e.printStackTrace();
           return null;
        }
     }
  }
```

Die konkret eingesetzten JSPs können hier im Deskriptor der Web-Applikation eingetragen werden.

Übermittlung der Value Objekte

Kann ein »Fat Client« die Aufbereitung von Value Objekten selber übernehmen, überträgt die JSP die Daten in serialisierter Form. Als Dateiformate können sofort der Standard-Serialisierungsmechanismus oder die »Long Term Persistence« von Java Beans eingesetzt werden. Die zugehörigen JSPs haben folgendes Aussehen:

```
<%
java.io.ObjectOutputStream oos = new
java.io.ObjectOutputStream(response.getOutputStream());
oos.writeObject(request.getAttribute("vo"));
%>
```

bzw.

```
<%
java.beans.XMLEncoder encoder = new
java.beans.XMLEncoder(response.getOutputStream());
encoder.writeObject(request.getAttribute("vo"));
encoder.close();
%>
```

Auf der Client-Seite werden die Objekte direkt aus dem Eingabestrom gelesen und können ausgelesen werden. Damit lässt sich problemlos ein sehr schlankes RMI-Protokoll definieren, das über Port 80 funktioniert. Wer also den enormen Overhead von »RMI over http« oder SOAP vermeiden will, schickt wie gehabt das Aktionskommando und die Parameter zum Controller-Servlet, erhält aber von der JSP serialisierte Daten-Objekte zur Aufbereitung.

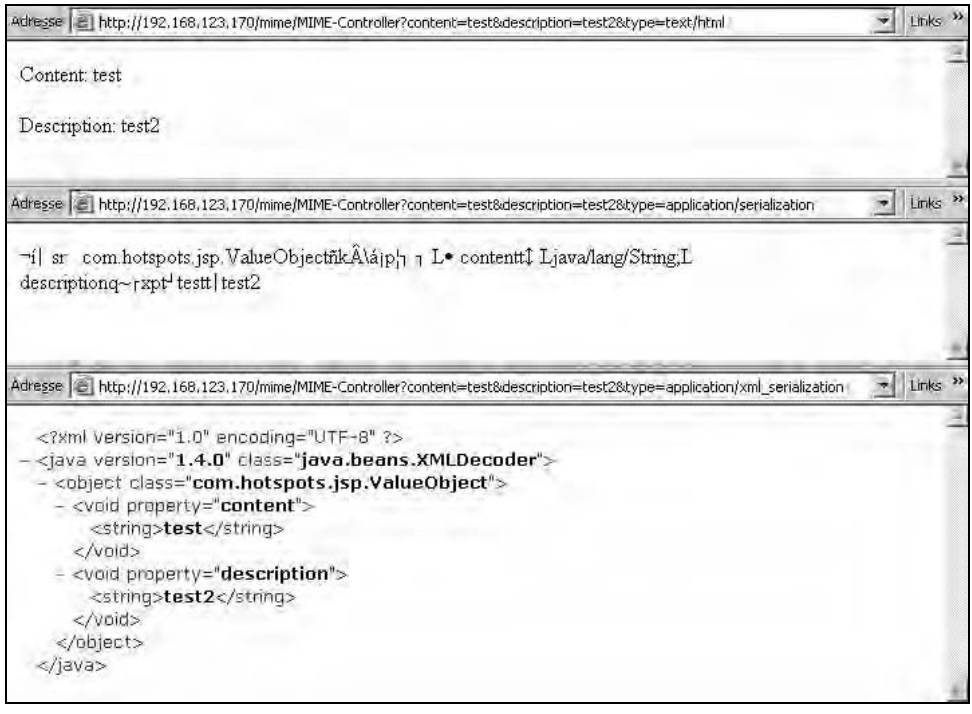


Abbildung 2.3: Die Ergebnisse der verschiedenen JavaServer Pages: oben: Rückgabeformat HTML, Mitte: ein mit java.io serialisiertes Java-Objekt, unten: serialisiert mit dem XMLEncoder des JDK 1.4

WAP

Auch WML-Seiten können direkt aus der JSP heraus generiert werden:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE wml PUBLIC '-//WAPFORUM//DTD WML 1.2//EN'   'http://www.wapforum.org/
DTD/wml_1.2.xml' >
<%@ page language="java" contentType="text/vnd.wap.wml"%>
<wml>
<card id="Demo" title="Value Object">
   <jsp:useBean id="vo" class="com.hotspots.jsp.ValueObject" scope="request"/>
   <p>Content:
   <jsp:getProperty name="vo" property="content"/></p>
   <p>
   Description:
   <jsp:getProperty name="vo" property="description"/></p>
</card>
</wml>
```

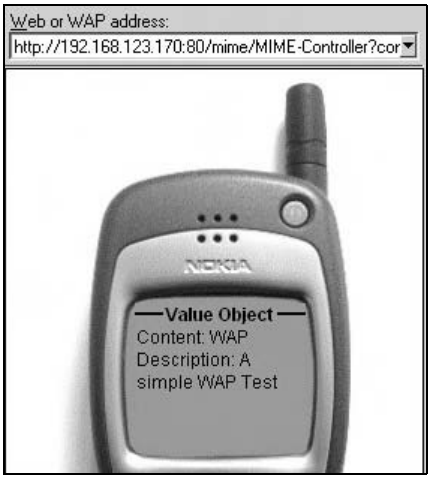


Abbildung 2.4: Die generierte WAP-Seite, dargestellt in einem WAP-Emulator (http://www.pyweb.com)

Diese statische Erzeugung ist aber mit Mehraufwand behaftet, da nun WML- und HTML-Seiten gemeinsam gepflegt werden müssen. Deshalb ist es häufig sinnvoller, die Umwandlung einer HTML in eine WML-Seite durch eine XSL-Transformation (Datei auf der Begleit-CD enthalten) durchzuführen. Zur Umsetzung bietet sich das J2EE-Pattern »Decorating Filter« an.

2.8 Realms

2.8.1 Problemstellung

- Wie werden die Benutzer einer Web-Anwendung konfiguriert?

2.8.2 Technischer Hintergrund

Ein Servlet-Container hat die Aufgabe, die web-basierte Authentifizierung eines Benutzers zu übernehmen. Je nach Konfiguration wird dabei unterschieden zwischen:

- Basic Authentication: Der Browser zeigt auf Anforderung des Web-Containers einen Eingabedialog an, indem Benutzername und Passwort eingegeben werden müssen.
- Form Based: Die Web-Applikation enthält ein Login-Formular mit den beiden Eingabefeldern »j_username« und »j_password« und der Submit-Aktion »j_security _check«. Im Deskriptor der Anwendung wird dieses Formular eingetragen.

- Client Authentifizierung: Hier baut der Browser über das SSL-Protokoll eine verlässliche und verschlüsselte Verbindung mit dem Server auf. Diese Art der Authentifizierung verlangt die Installation von geeigneten Zertifikaten im Browser.

Benutzer und -rollen werden von den verschiedenen Applikationsservern unterschiedlich konfiguriert. In der DTD für den Web-Deskriptor kann bezüglich der Sicherheit nur die Liste der Einschränkungen eingetragen werden.

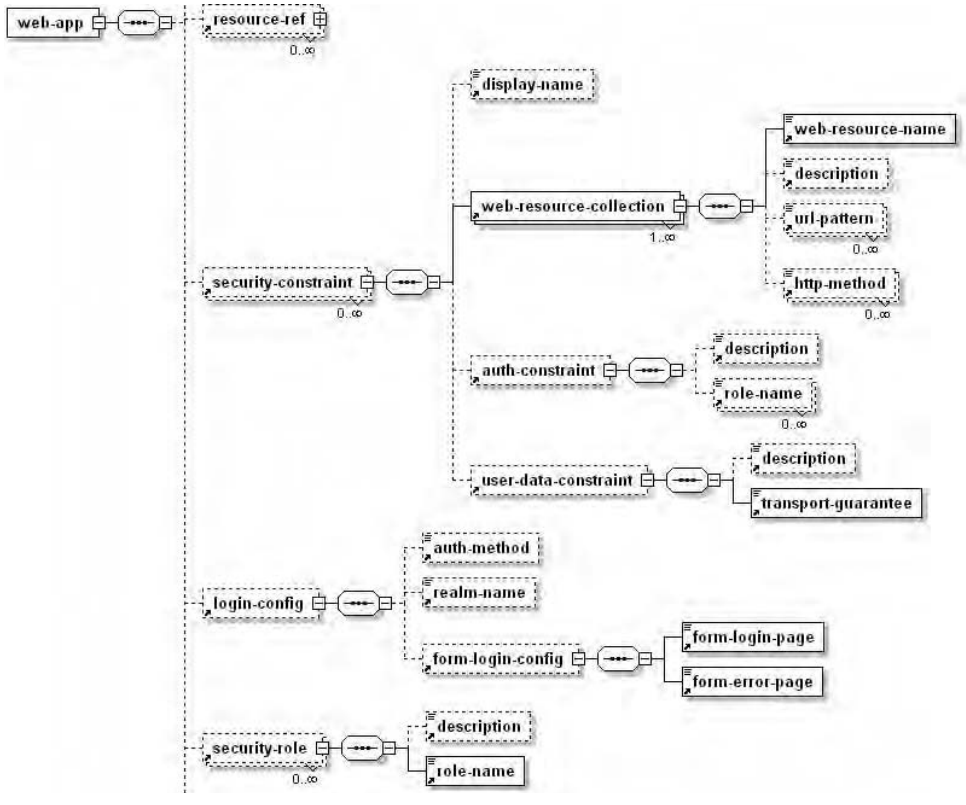


Abbildung 2.5: Ausschnitt aus der DTD der Web-Applikation. Gezeigt sind die Tags mit direktem Bezug zur Sicherheit

Der Zugriff auf eine Resource oder, wie im folgenden Beispiel auf ein Servlet, wird durch einen Eintrag in der web.xml-Datei geschützt:

```
<security-constraint>
   <web-resource-collection>
      <web-resource-name>WRCollection</web-resource-name>
      <url-pattern>/AuthenticationServlet</url-pattern>
      <http-method>POST</http-method>
```

```
        <http-method>GET</http-method>
      </web-resource-collection>
      <auth-constraint>
        <role-name>user</role-name>
      </auth-constraint>
      <user-data-constraint>
        <transport-guarantee>NONE</transport-guarantee>
      </user-data-constraint>
  </security-constraint>
  <login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>Default</realm-name>
  </login-config>
```

Wo und wie die eigentlichen Benutzer-Informationen gefunden und geprüft werden, bleibt den herstellerabhängigen Deskriptoren vorbehalten. In der Referenz-Implementierung von Sun werden einfach in der `sun-j2ee-ri.xml`-Datei Benutzern Rollen zugeordnet, die entweder in einer Datei (`config/realms/keyfile`) oder einem Java-Keystore (`config/realms/certstore`) gehalten werden.

Beim JBoss werden Ressourcen dadurch geschützt, indem in der jboss-web.xml-Datei die Web-Applikation einem Realm zugeordnet wird:

```
<jboss-web>
<security-domain>java:/jaas/web-login</security-domain>
</jboss-web>
```

Security-Realms werden in der login-config.xml-Datei definiert. Wo die Benutzer-Informationen und Rollen schließlich abgelegt werden, hängt von der Konfiguration des realms und dem darin verwendeten Login-Modul ab. Beispiele hierzu sind im Kapitel 3 im Abschnitt über die Konfiguration der Applikationsserver-Security zu finden.

BEA Weblogic verwendet ebenfalls einen speziellen Deskriptor zur Konfiguration, die Datei weblogic-application.xml. Darin wird für die J2EE-Anwendung ein Realm eingetragen:

```
<weblogic-application>
…
   <security>
      <realm-name>web-realm</realm-name>
   </security>
</weblogic-application>
```

Die Zuordnung von Benutzern zu einer Anwendungsrolle erfolgt dann in der Datei WEB-INF/weblogic.xml:

```
<weblogic-web-app>
…
   <security-role-assignment>
      <role-name>user</role-name>
      <principal-name>jxpert</principal-name>
   </security-role-assignment>
</weblogic-web-app>
```

Apache Tomcat alleine ist auch bereits mit Security Realms ausgestattet. In der Standard-Installation sind enthalten:

- Ein Memory-Realm. Dieses liest die Benutzerinformationen aus einer XML-Datei.
- Das JDBC-Realm liest aus Datenbank-Tabellen (Benutzername und Passwort sowie Benutzername und zugehörige Rollen).
- Das JNDI-Realm verwendet einen JNDI-Lookup, der sich oft auf einen LDAP-Server bezieht.

Die Konfiguration von globalen Realms ist einfach und erfolgt in der Datei server.xml, hier z.B. ein JDBC-Realm:

```
<Realm  className="org.apache.catalina.realm.JDBCRealm"
   driverName="COM.cloudscape.core.RmiJdbcDriver"
   connectionURL="jdbc:cloudscape:rmi://www.jxperts.de:1099/
TomcatUserDB?user=test;password=test"
   userTable="users" userNameCol="col_user " userCredCol="col_pwd"
    userRoleTable="col_roles" roleNameCol="col_role " />
```

Schließlich können auch Realms pro virtuellen Host bzw. pro Web-Applikation definiert werden.

3 Geschäftslogikschicht

3.1 Optimistische Locks mit EJB 2.0 und der Container Managed Persistence (CMP)

3.1.1 Problemstellung

Eine typische Web-Anwendung erlaubt einer großen, unbestimmten Anzahl von Benutzern die Arbeit mit der gleichen Datenbank. Dabei können die Zugriffe gleichzeitig auf die gleichen Daten stattfinden. Bei dem lesenden Zugriff ist dieser Vorgang unproblematisch, die Datenbank liest die benötigten Daten und gibt diese an die EJB-Schicht zurück. Da es sich in dem Fall lediglich um den lesenden Zugriff handelt, werden diese Daten nicht modifiziert. Diese können hervorragend in jeder Schicht gepuffert (gecached) werden. Dies steigert die Performance, da dadurch die Kommunikation mit der Integrationsschicht und der DB entfällt. Falls sich bereits alle Daten im Cache befinden, kann die Datenbank (in der Praxis wird die Datenbankverbindung von dem Applikationsserver ständig überprüft) heruntergefahren werden. Bei den meisten Zugriffen einer Durchschnittsanwendung handelt es sich um die lesende »Sorte« (ca. 80%).

Ein echtes Problemkind sind die Schreibzugriffe, da sich hier logische Abhängigkeiten zwischen den Datensätzen mehrerer Clients ergeben können. Auch unsere Caches müssen bei einem schreibenden Zugriff gleichzeitig mit der Datenbank aktualisiert werden. Konzentrieren wir uns aber hier auf die »logischen« Datenabhängigkeiten zwischen den Clients.

Stellen wir uns ein Bücherladen vor, wo jeder Internetbenutzer lesend und jeder Mitarbeiter des Ladens auch schreibend auf die Bücherdaten zugreifen kann. Bei dem schreibenden Zugriff könnten sich die Mitarbeiter ihre Daten gegenseitig überschreiben. Um diese Situation zu simulieren müsste der erste Mitarbeiter ein bestimmtes Buch z. B. mit der ISBN 382731903X lesen.

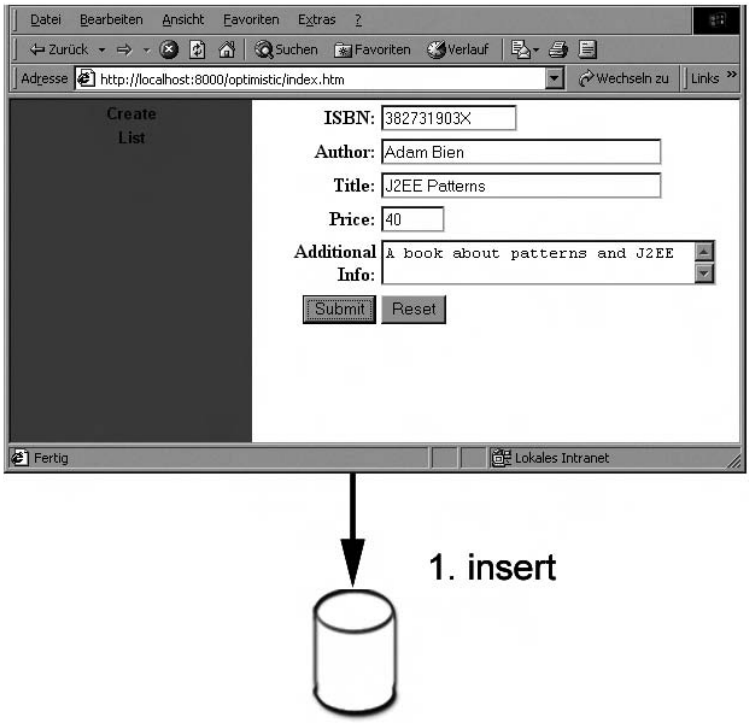


Abbildung 3.1: Eingabe des Datensatzes

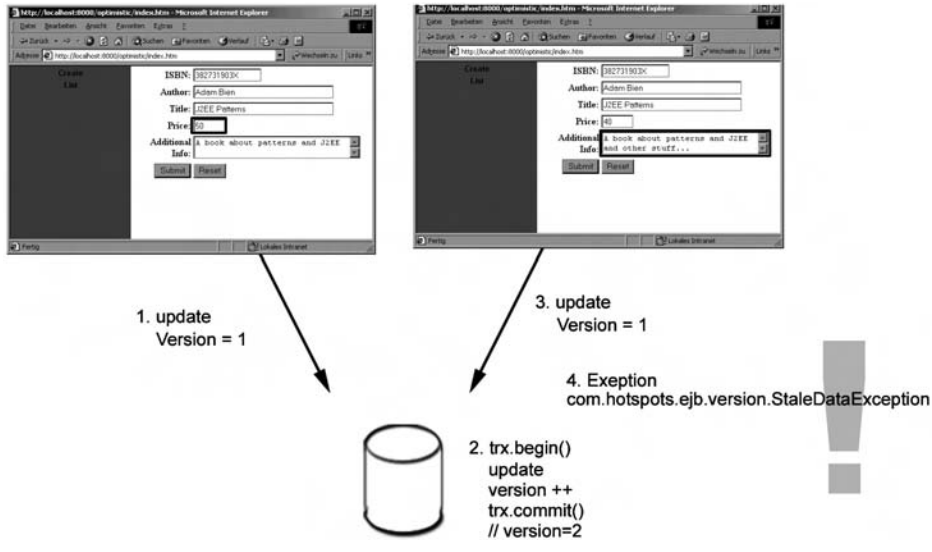


Abbildung 3.2: Der Versionskonflikt zweier Clients

Nach dem Lesen ändert er die Buchbeschreibung. Für die Formulierung braucht er aber einige Minuten. Währenddessen liest auch der zweite Mitarbeiter der Firma das Buch mit der ISBN 382731903X, um den Preis anzupassen. Dabei wird natürlich nicht nur der Preis, sondern der komplette Datensatz gelesen.

Der erste Mitarbeiter schreibt seine Daten in die Datenbank zurück. Hier werden allerdings nicht nur die Buchbeschreibung, sondern auch die anderen »Buchattribute« in die Datenbank geschrieben. Auch der zweite Mitarbeiter schreibt jetzt seine Änderung in die Datenbank zurück. Er überschreibt jetzt mit seinem Datensatz die Buchbeschreibung, die mit viel Mühe vom ersten Mitarbeiter eingegeben wurde.

3.1.2 Technischer Hintergrund

Dieses Problem tritt natürlich nicht nur in den Web-, sondern auch in den »alten« Client-Server-Anwendungen auf. Meistens hat man hier das Problem mit dem »pessimistischen Locking« des Datensatzes gelöst. Bei einer SQL-Datenbank kann dies mit dem Anweisung »SELECT FOR UPDATE« erfolgen. Ob hier der komplette Datensatz oder nur eine Spalte der Tabelle gesperrt wird, hängt von dem Datenbankhersteller ab. Der nächste Client muss solange warten, bis die Transaktion committed oder rollbacked wurde. Es handelt sich hier um eine echte Sperre des Datensatzes. Für die Sperre ist hier die jeweilige Datenbank verantwortlich und wird durch den JDBC-Treiber bis zu den Clients propagiert.

Da sich bei den Client-Server-Anwendungen meistens um »intelligente« Fat-Clients handelt, kann hier besser auf die Blockierung der Datensätze reagiert werden. Auch kurze Transaktionszeiten lassen sich hier einfacher implementieren.

Ein HTML-Client kann nicht auf die Freigabe von benötigten Datensätzen längere Zeit warten. Diese Sperre kann sich mit »Connection Timouts« des Browsers äußern. Man kann sich also in einer solchen Umgebung nicht auf die Lockingmechanismen der Datenbank verlassen. Ferner sollten die jeweiligen Schichten (Web, EJB, Integration usw.) austauschbar bleiben. Damit kennt der Applikationsserver die Datenbank nur durch ihre JDBC-Schnittstelle. Der Anwendungsentwickler kann sich somit auf die Locking-Strategien der Datenbank nicht verlassen. Hier wird eine datenbankunabhängige Lösung benötigt.

3.1.3 Lösung

Die web-konforme Lösung des Problems heißt hier das »optimistische Locking«. Diese Lösung basiert nicht auf den jeweiligen Fähigkeiten der Datenbanken, sondern vielmehr auf den Design-Entscheidungen des Entwicklers. Dabei kann frei zwischen der EJB 1.1- und der EJB 2.0-Spezifikation gewählt werden. Auch die Container Managed Persistence (CMP) wird hier unterstützt. Auch bei der Wahl der BMP-Strategie kann ein direkter Datenbankzugriff, z. B. über ein Data Access Object (DAO), implementiert werden.

Obwohl diese Lösung mit ihren Namen auch eine Sperrung des Datensatzes suggeriert, werden die Datensätze einer Tabelle nicht für das Lesen gesperrt. Man hofft, dass die meisten Clients nur lesend zugreifen – ein wenig Optimismus (»Optimistic Locking«) gehört schon dazu. Falls irgendein Client sich doch für das Schreiben der geänderten Datensätze entscheiden sollte, wird zuerst seine »Version« mit der Version der Datenbank verglichen. Bei einer Übereinstimmung wird der Datensatz zurückgeschrieben. Falls die Versionsnummern nicht übereinstimmen, wird eine Exception geworfen, die sich normalerweise durch eine Benutzermeldung auf der Client-Seite äußert. Der Anwender erhält eine Möglichkeit den Datensatz neu zu laden. Dabei wird auch die aktuelle Versionsnummer geladen, die dann auch beim nächsten Update mit dem Applikationsserver verglichen wird. Leider muss die Änderung noch mal eingepflegt werden, da diese natürlich nicht in die Datenbank zurückgeschrieben wurde. Obwohl diese Vorgehensweise nicht sehr benutzerfreundlich ist, ermöglicht diese dem zweiten Mitarbeiter die Arbeit mit den aktuellsten Daten.

3.1.4 Praxis

In der Praxis möchte man hier auf die bereits existierende Infrastruktur zurückgreifen – also auf einen Applikationsserver. Dieses Pattern erlaubt sowohl die Verwendung von der EJB 1.1 als auch der EJB 2.0 Spezifikation. Auch beide Persistenzvarianten, nämlich die CMP und die Bean Managed Persistence (BMP) können für die Implementierung verwendet werden. Um die Wiederverwendbarkeit zu erhöhen, wurden alle generischen Klassen in ein gemeinsames Package `com.hotspots.ejb.version` ausgelagert.

Das Interface `VersionIF` ermöglicht die Abfrage der aktuellen Versionsnummer.

```
package com.hotspots.ejb.version;
import java.io.*;
public  interface VersionIF extends Serializable {
    public int getVersion();
}
```

Es wird indirekt sowohl von dem Value Object, als auch von der EntityBean implementiert. Somit sind direkte Vergleiche der Version der Datenbank mit der Version des Value Objects möglich. Das Interfaces `VersionModifierIF` ermöglicht die Veränderung der Version. Sinnvollerweise ist dies nur in der EntityBean möglich.

```
package com.hotspots.ejb.version;
public interface VersionModifierIF {
      void setVersion(int newVersion);
}
```

Allerdings holt sich die Entity diese Funktionalität durch die Vererbung von der Klasse `AbstractVersionCheck`.

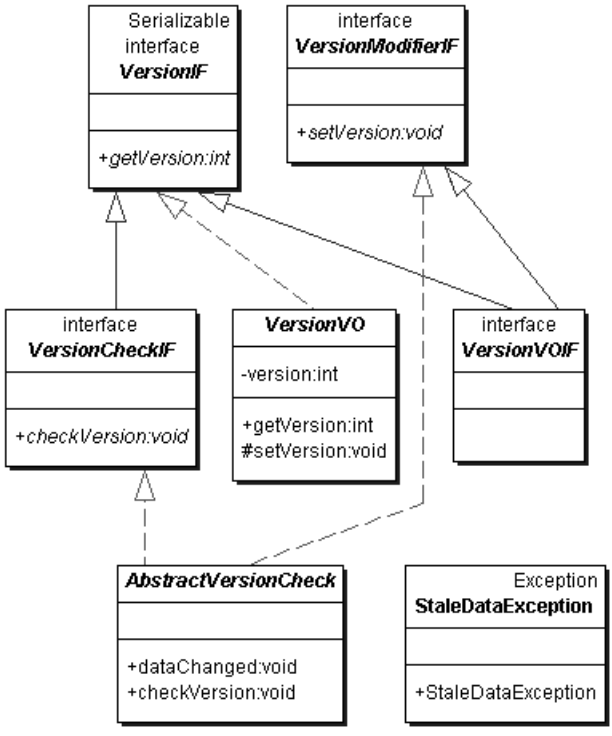


Abbildung 3.3: Das Package com.hotspots.ejb.version.

```
package com.hotspots.ejb.version;
public abstract class AbstractVersionCheck implements VersionCheckIF,
VersionModifierIF {
  public void dataChanged(){
        int currentVersion = getVersion();
        this.setVersion(++currentVersion);
  }
  public void checkVersion(VersionIF versionInfo) throws StaleDataException{
            if(this.getVersion()!=versionInfo.getVersion())
            throw new StaleDataException(this,"checkVersion",this,versionInfo);
    }
}
```

Diese abstrakte Klasse liefert bereits die benötigte Grundfunktionalität, die für eine einfache Versionsverwaltung ausreichen sollte.

```
public abstract class BookBean extends AbstractVersionCheck implements EntityBean,
BookIF  {
    private EntityContext entityContext = null;
```

```
    public String ejbCreate(String isbn,String title,String author,int
price,String additionalInfo)throws CreateException{
        this.setISBN(isbn);
        this.setTitle(title);
        this.setAuthor(author);
        this.setPrice(price);
        this.setVersion(0);
        this.setAdditionalInfo(additionalInfo);
        return null;
    }
    public void ejbPostCreate(String isbn,String title,String author,int
price,String additionalInfo) throws CreateException{
    }
   public BookVO getBookVO(){
     return new
BookVO(this.getISBN(),this.getAuthor(),this.getTitle(),this.getPrice(),this.getAdd
itionalInfo(),this.getVersion());
    }
  public void setBookVO(BookVO book) throws StaleDataException{
   System.out.println("[BookBean.setBookVO] ValueObject with version " +
book.getVersion() + " passed !");
this.checkVersion(book);
   System.out.println("[BookBean.setBookVO] The same version => updating the data
!");
       this.setAdditionalInfo(book.getAdditionalInfo());
       this.setAuthor(book.getAuthor());
       this.setPrice(book.getPrice());
       this.setTitle(book.getTitle());
       dataChanged();
       System.out.println("[BookBean.setBookVO] Data updated, versionnumber
increased, current version: " + this.getVersion() + " old version " +
book.getVersion());
    }
    public void ejbStore() { }
    public void ejbPassivate(){  }
    public void ejbLoad() { }
    public void setEntityContext(EntityContext entityContext) {
        this.entityContext = entityContext;
    }
    public void ejbRemove(){ }
    public void unsetEntityContext(){}
}
```

Die Änderung des Zustandes der Entity `com.hotspots.ejb.data.BookBean` erfolgt durch die Methode `setBookVO(BookVO book)`. Die feingranulären Setter der Bean sollten nicht direkt aufgerufen werden können.

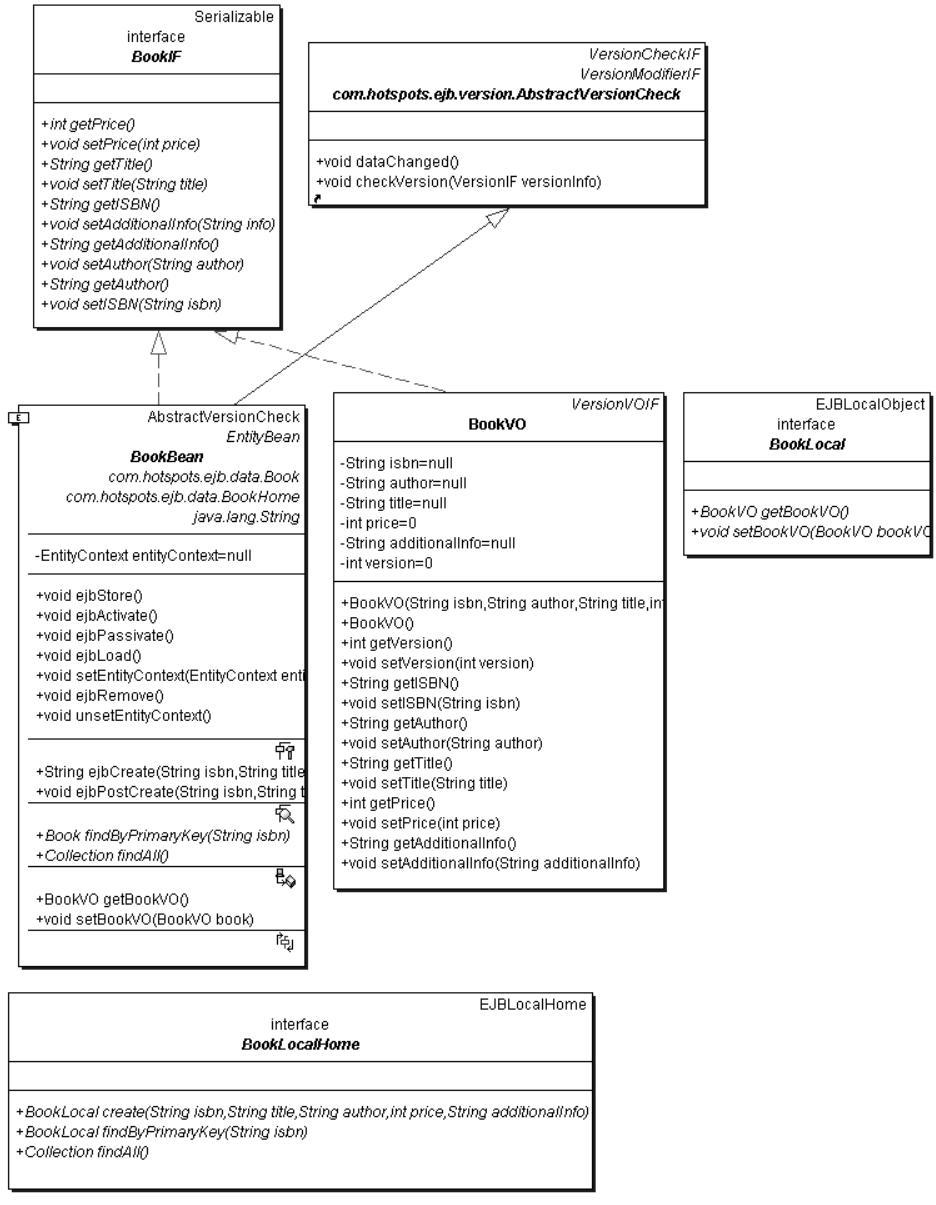


Abbildung 3.4: Das Package com.hotspots.ejb.data

Dies kann durch die »Freigabe« nur ausgewählter Methoden in den Remote- und Local-Interfaces der Bean erreicht werden. Das Remote-Interface erlaubt lediglich das Lesen des Zustands der Bean mit Hilfe eines Value Objects.

```
package com.hotspots.ejb.data;
import javax.ejb.*;
import java.rmi.*;
public interface Book extends EJBObject {
    public BookVO getBookVO() throws RemoteException;
}
```

Das Local-Interface ist nicht ganz so restriktiv – hier können auch alle Attribute von »außen« gesetzt werden. Dieses Interface kann natürlich nur von innerhalb eines Containers aufgerufen werden, was die Verwendung einer Session Facade nötig macht.

```
public interface BookLocal extends EJBLocalObject{
    public BookVO getBookVO();
    public void   setBookVO(BookVO bookVO) throws StaleDataException;
}
```

Das für den Datentransport zuständige Value Object implementiert das Interface `BookIF`, das die benötigten Methoden definiert.

```
public interface BookIF extends Serializable{

    int  getPrice();
    void setPrice(int price);

    String getTitle();
    void   setTitle(String title);

    String getISBN();
    void   setAdditionalInfo(String info);

    String getAdditionalInfo();
    void setAuthor(String author);

    String getAuthor();
    void   setISBN(String isbn);
}
```

Das gleiche Interface implementiert die Entity `BookBean` – somit können alle Attribute der Entity mit dem Value Object `BookVO` transportiert werden. Der Client »sieht« allerdings diese Methoden nicht, diese stellen lediglich die CMP-Felder der EntityBean ab.

Beim Aufruf der sichtbaren Methode `setBookVO` wird ein kompletter Datensatz übergeben. Zuerst wird die Version des übergebenen Value Object gelesen. Diese wird mit der aktuell gültigen Version der Entity verglichen. Die Übereinstimmung der beiden Versionsnummern wird in der Methode `checkVersion` der Superklasse überprüft. Falls diese nicht übereinstimmen sollten, wird eine Exception `StaleDataException` erzeugt und geworfen. Dabei wird sowohl das Value Object als auch die Entity übergeben, um mehr Informationen über den Versionskonflikt zu erhalten.

Bei einer Übereinstimmung der Versionsnummern wird innerhalb einer Transaktion zuerst der Zustand des Value Objects mit den Getter-Methoden gelesen und mit den Set-Aufrufen der Entity an diese »übertragen«. Anschließend wird die Methode `dataChanged()` aufgerufen, die für die Änderung der Versionsnummer der Bean zuständig ist.

```
public void dataChanged(){
        int currentVersion = getVersion();
        this.setVersion(++currentVersion);
  }
```

Unmittelbar nach diesem Vorgang sollte auch die Transaktion committed werden. Somit werden alle Daten konsistent und dauerhaft in einer Datenbank (nicht unbedingt in einer relationalen) abgelegt.

Falls zwei Clients gleichzeitig den gleichen Datensatz modifiziert haben und der erste bereits seine Daten zurückgeschrieben hat, kann sein Nachfolger seine Änderung nicht mehr »einspielen«. Die Version der Bean hat sich bereits bei dem ersten Update der Entity geändert (typischerweise erhöht) – eine `StaleDataException` wird geworfen, um die Änderung zu signalisieren. Dem zweiten Client bleibt nichts anders, als die Daten noch mal, mit der aktuellen Version, zu lesen. Somit sind die Daten des Clients wieder aktuell und können in die Entity zurückgespielt werden.

Es werden jeweils die Versionsnummern der Entity geändert und nicht des Value Objects. Das Value Object kann hier als ein Snapshot der Daten der Entity verstanden werden.

Die sichtbare Schnittstelle des Clients wird durch das Interface `BookMgrLocalIF` vorgegeben. Es handelt sich dabei um eine Geschäftsschnittstelle, d.h. mit Aufrufen dieser Methoden werden die Anwendungsfälle (Use Cases) abgebildet.

```
public interface BookMgrLocalIF {
  public void createBook(BookVO book) throws BookAlreadyExistsException;
 public void deleteBook(String isbn) throws BookDoesNotExistException;
 public void updateBook(BookVO book) throws
BookDoesNotExistException,StaleDataException;
 public BookVO getBook(String isbn)  throws BookDoesNotExistException;
 public BookVO[] getAll();
}
```

Die Infrastruktur und die eingesetzte Technologie sollten dem Client an dieser Stelle egal sein. Wichtig ist es hier, dass die benötigte Geschäftslogik »benutzt« werden kann. Aus dem Grund wurden auch eigene Exceptions definiert, die nichts mit der EJB oder JDBC Technologie zu tun haben. Auch hier handelt es sich um »Geschäftslogik-Exceptions«, die lediglich logische- und nicht-technische Fehler signalisieren sollen. Dieses Interface wird auch direkt von der `Session BookMgrBean` implementiert.

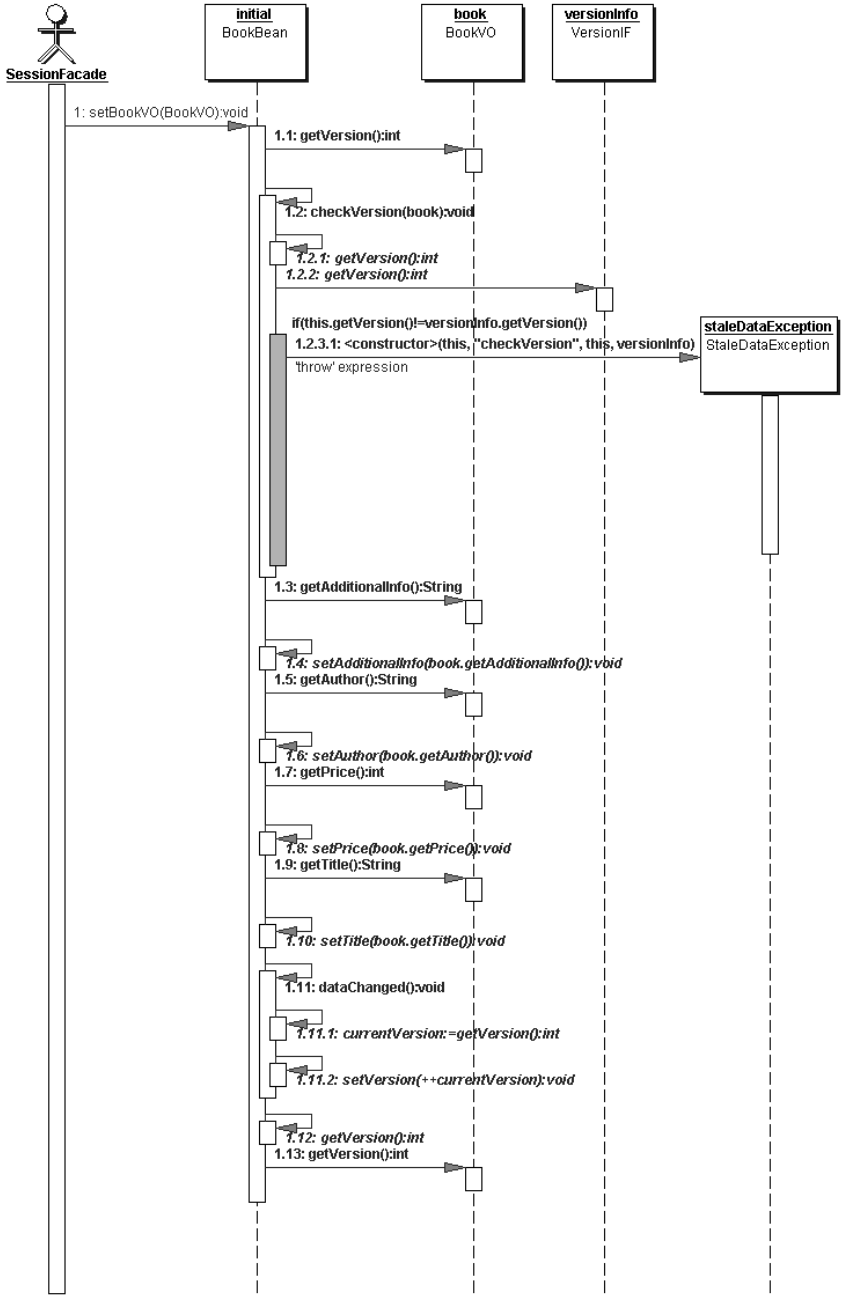


Abbildung 3.5: Die Funktionsweise der Versionsverwaltung der BookBean

```
public class BookMgrBean implements SessionBean, BookMgrLocalIF {
  private SessionContext ctx = null;
  private final String  BOOK_LOCAL_HOME         = "java:comp/env/ejb/
BookBeanLocal";
  private ServiceLocatorEJB serviceLocator      = null;
  private BookLocalHome home  = null;
  private BookLocal     local = null;
    public void setSessionContext(SessionContext param) {
            this.ctx = param;
    }
    private void init() throws RemoteException{
        try{
            this.serviceLocator = new ServiceLocatorEJB();
            this.home    =
(BookLocalHome)this.serviceLocator.getLocalHome(BOOK_LOCAL_HOME);
        }catch(Exception e){
            throw new RemoteException("[BookMgrBean.setSessionContext] " + e);
        }
    }
    public void ejbCreate() throws RemoteException{
        this.init();
    }
    public void ejbActivate(){    }
    public void ejbPassivate(){    }
    public void ejbRemove() {    }
   public void createBook(BookVO book) throws BookAlreadyExistsException{
     try{
this.home.create(book.getISBN(),book.getTitle(),book.getAuthor(),book.getPrice(),b
ook.getAdditionalInfo());
       }catch(CreateException e){
                    throw new BookAlreadyExistsException(e);
       }
    }
    public void deleteBook(String isbn) throws BookDoesNotExistException{
      try{
             getLocal(isbn).remove();
       }catch(FinderException e){
              throw new BookDoesNotExistException(isbn,e);
       }catch(RemoveException e){
 throw new EJBException("[BookMgrBean.deleteBook] Error removing book with PK: " +
isbn + " Exception: " + e);
       }
    }
    public void updateBook(BookVO book) throws
BookDoesNotExistException,StaleDataException{
       try{
                getLocal(book.getISBN()).setBookVO(book);
       }catch(FinderException e){
                    throw new
                    BookDoesNotExistException(book.getISBN(),e);
       }
    }
    public BookVO getBook(String isbn)  throws BookDoesNotExistException{
        BookLocal local = null;
```

```
        try{
         return this.localToVO(getLocal(isbn));
        }catch(FinderException e){
                    throw new BookDoesNotExistException(isbn,e);
        }
    }
        public BookVO[] getAll(){
        try{
              return this.collectionToVo(this.home.findAll());
        }catch(FinderException exception){
                    throw new EJBException("[BookMgrBean.getAll()]  Problem
finding all books ! Exception:  " + exception.toString());
        }
    }
private BookLocal getLocal(String isbn) throws FinderException{
      if(isbn==null)
    throw new FinderException("[BookMgrBean.getLocal()] passed parameter (isbn) is
null !");
      if(this.home!=null)
local = this.home.findByPrimaryKey(isbn.trim());
        else
 throw new EJBException("[BookMgrBean.getLocal()] BookLocalHome is null!");
             return local;
}
    private BookVO localToVO(BookLocal local){
       return local.getBookVO();
    }
  private BookVO[] collectionToVo(Collection collection){
        if(collection == null)
            return null;
        BookVO value[] = null;
        Iterator iterator = collection.iterator();
        ArrayList list = new ArrayList();
        while(iterator.hasNext()){
            list.add(((BookLocal)iterator.next()).getBookVO());
        }
        value = new BookVO[list.size()];
        for(int i=0;i<list.size();i++){
            value[i] = (BookVO)list.get(i);
        }
        return value;
    }
}
```

Diese SessionBean implementiert das J2EE Pattern Session Facade, sie überwacht den Zugriff auf die `BookBean` und implementiert zusätzliche Funktionalitäten. Interessanterweise ist das Local-Interface der Bean »leer«. Es werden alle Methoden des Business-Interfaces `BookMgrIF` übernommen.

```
package com.hotspots.ejb.sessionfacade;
import javax.ejb.EJBLocalObject;
public interface BookMgrLocal extends EJBLocalObject, BookMgrLocalIF {
}
```

Dass die Erzeugung dieser Bean keine Parameter benötigt, ist auch der Aufbau des `LocalHome`-Interfaces ziemlich einfach.

```
package com.hotspots.ejb.sessionfacade;
import javax.ejb.*;
import javax.ejb.EJBLocalHome;

public interface BookMgrLocalHome extends EJBLocalHome {
    public BookMgrLocal create() throws CreateException;
}
```

Diese Tatsache erlaubt die Benutzung einer parameterlosen Methode `create()`. Somit lässt sich diese Bean sowohl als stateless als auch als stateful Variante deployen. In unserem Beispiel sind wir auf das Konversationsgedächtnis einer stateful Bean nicht angewiesen und verwenden deshalb die stateless SessionBean.

Da es sich hier um eine einfache Anwendung handelt, ist die Verwendung des Business Delegate Patterns nicht notwendig. Dieses Pattern würde eine weitere Abstraktionsschicht definieren, die für die weitere Verallgemeinerung des Business-Interfaces der SessionBean zuständig wäre. Dabei handelt es sich um ein weiteres Interface, dessen Implementierung direkt mit dem Local- oder Remote-Interface der Session Facade (SessionBean) arbeitet.

Auf die Verwendung des ServiceLocators wird hier allerdings nicht verzichtet, da dieser für eine komfortable Arbeit sowohl mit der JNDI Technologie als auch für die Beschaffung des Home- und LocalHome- Interfaces zuständig ist.

3.2 Was »kosten« unnötige Transaktionen?

3.2.1 Problemstellung

Der große Vorteil bei der Entwicklung von EJB-Komponenten ist die deklarative Programmierung. Diese erlaubt die Modifikationen von Deployment Descriptoren, die wiederum das »Verhalten« der Komponente beeinflusst. Wegen der Auslagerung dieser Konfigurationen in XML-Dateien (Deployment Deskriptoren), ist eine Neukompilierung der Geschäftslogik für die gewünschte Änderung des Zustandes nicht notwendig. Von diesen Einstellungen hängt aber die Implementierung von automatisch generierten Klassen ab. Diese repräsentieren den EJB-Container und bieten die Laufzeitumgebung für unsere EJBs an. Innerhalb dieser Umgebung sind erst die Geschäftslogikkomponenten (also EJBs) lauffähig. Die Einstellungen des Deployment Deskriptors sind umfassend und können feingranulär für jede Komponente und sogar ihre Methoden vorgenommen werden. Beispielsweise kann das gewünschte Transaktionsverhalten für jede Methode des Remote-Interfaces eingestellt werden.

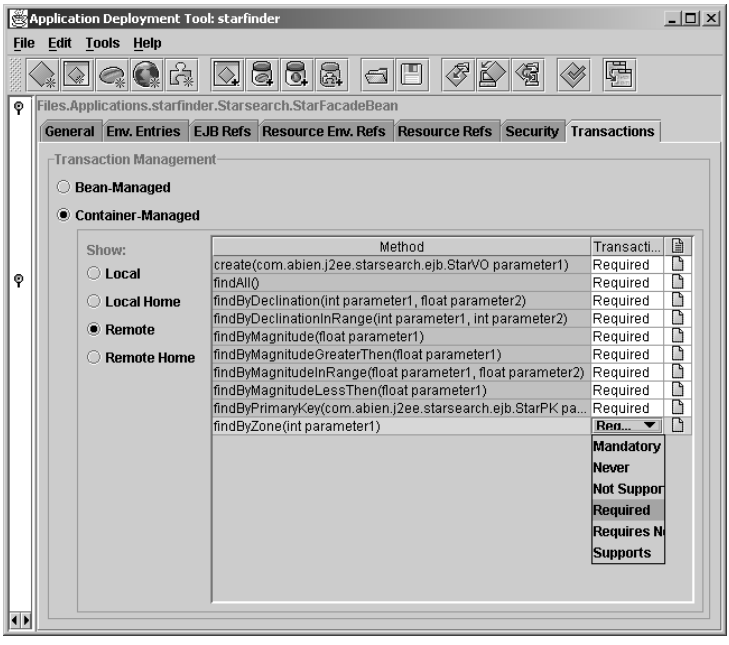


Abbildung 3.6: Die Einstellungen des Transaktion-Levels einer SessionBean

Der Inhalt des Deployment Deskriptors ist dementsprechend komplex. Jede Methode wird nochmal mit ihren Parametern, ihren Namen, der Zugehörigkeit zu dem Interface (Home, Remote, Local) und dem Transaktionslevel aufgeführt.

```
<container-transaction>
    <method>
      <ejb-name>StarFacadeBean</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>findByMagnitudeInRange</method-name>
      <method-params>
        <method-param>float</method-param>
        <method-param>float</method-param>
      </method-params>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
  <container-transaction>
    <method>
      <ejb-name>StarFacadeBean</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>findByMagnitudeLessThen</method-name>
      <method-params>
        <method-param>float</method-param>
      </method-params>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
```

Da diese Einstellungen einen gewissen Aufwand bedeuten, kann in der Praxis mit etwas anderer Schreibweise der Deployment Deskriptor entscheidend verkürzt werden.

```
<assembly-descriptor>
   <container-transaction>
      <method>
        <ejb-name> StarFacadeBean </ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute> Required </trans-attribute>
    </container-transaction>
  </assembly-descriptor>
```

Dabei kann man mit dem Stern »*« angeben, dass alle Methoden der Bean, unabhängig von ihren Parametern, gewisse Transaktionslevel (hier »Required«) erhalten sollen. Da man oft sicher gehen möchte, wird meistens der Transaktionslevel »Required« oder sogar »RequiresNew« gewählt. Dabei wird bei jedem Methodenaufruf, wenn auch unnötig, eine neue Transaktion gestartet.

3.2.2 Technischer Hintergrund

Grundsätzlich sind folgende Transaktionslevels möglich:

Transaktionslevel	**Beschreibung**
Required	Der Container startet eine neue Transaktion oder verwendet die bestehende. Man kann sich darauf verlassen, dass eine Methode mit diesem Level immer innerhalb einer Transaktion ausgeführt wird.
RequiresNew	Der Container startet immer eine neue Transaktion. Falls die Methode bereits innerhalb einer Transaktion aufgerufen wurde, wird die bestehende Transaktion suspendiert. Man kann sich darauf verlassen, dass eine Methode mit diesem Level immer innerhalb einer **neuen** Transaktion gestartet wird.
NotSupported	Eine Methode mit diesem Level wird nicht innerhalb einer Transaktion ausgeführt. Falls diese Methode mit einer Transaktion bereits aufgerufen wird, »unterbricht« der Container die bestehende Transaktion, damit die Methode ohne einen Transaktionskontext ausgeführt wird. Dieser Level sollte verwendet werden, wenn man z.B. auf eine nichttransaktionsfähige Ressource ohne eine Transaktion zugreifen möchte, aber trotzdem den Zugriff auf diese Methode innerhalb einer Transaktion erlaubt.
Supports	Da dieser Level unberechenbar ist, sollte dieser nicht verwendet werden. Falls eine Methode mit diesem Level bereits innerhalb einer Transaktion aufgerufen wird, übernimmt der Container den Transaktionskontext, ansonsten wird diese Methode ohne eine Transaktion aufgeführt. Dieser Level ist, wenn überhaupt, nur für schnelle »hacks« einsetzbar.

Transaktionslevel	Beschreibung
Mandatory	Falls man eine Methode mit diesem Level deployed, muss diese Methode bereits immer innerhalb einer Transaktion aufgerufen werden. Bei einem Versuch, diese Methode ohne eine Transaktion aufzurufen, wird eine Exception (die `IllegalStateException`) geworfen. Beim Aufruf dieser Methode wird eine Transaktion vorausgesetzt.
Never	Bei diesem Level handelt es sich genau um das Gegenteil zu dem Level »Mandatory«. Eine Methode darf nicht innerhalb einer Transaktion aufgerufen werden. Ein Versuch wird auch hier mit der `IllegalStateException` »bestraft«. Dieser Level sollte für Methoden eingestellt werden, die nicht in der Lage sind, das Ergebnis zu »rollbacken« – also wenn man innerhalb dieser Methoden auf nicht transaktionsfähige Ressourcen zugreifen möchte. Beim Verschicken einer E-Mail ist es unmöglich, bei einem »Rollback« das Verschicken ungeschehen zu machen. Ferner könnte dieser Level eingestellt werden, wenn man innerhalb der Methoden auf nichttransaktionsfähige Datenbanken zugreift.

Wie bereits erwähnt, werden diese Level in dem Deployment Deskriptor eingestellt. Beim Deployen generiert der Container die entsprechenden Wrapper-Klassen, die für die Transaktionssteuerung zuständig sind. Der Source-Code der Wrapper oder Proxies hängt von der vorgenommenen Einstellung ab.

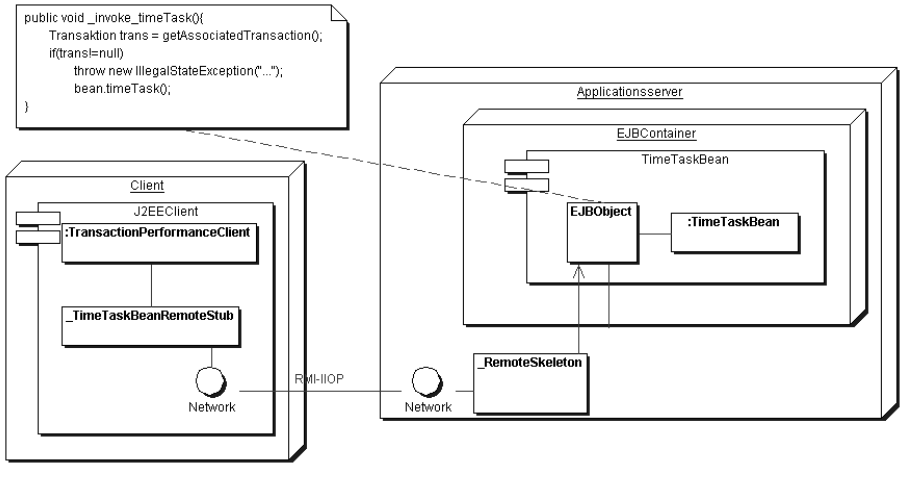


Abbildung 3.7: Eine EJB mit den Wrapper-Klassen

Der generierte Code für den Wrapper, für den der Transaktionslevel »RequiresNew« generiert wurde, könnte so aussehen:

```
public void _invoke_timeTask(){
      getAssociatedTransaction().suspend();
      Transaktion newTrans = getNewTransaction();
```

```
        try{
        newTrans.begin();
                bean.timeTask();
                newTrans.commit();
        }catch(Exception e){
                newTrans.rollback();
        }
        getAssociatedTransaction().resume();
}
```

Für den Level »Supports« wäre der Code wesentlich einfacher:

```
public void _invoke_timeTask(){
                bean.timeTask();
}
```

Da man aber nicht mit »Supports« arbeiten sollte, könnte man hier auch »Never« verwenden, um zu signalisieren, dass man an Transaktionen nicht interessiert ist:

```
public void _invoke_timeTask(){
        Transaktion trans = getAssociatedTransaction();
        if(trans!=null)
                throw new IllegalStateException("...");
                bean.timeTask();
}
```

Grundsätzlich lässt sich feststellen, dass der Aufwand für »Required« und »RequiresNew« wesentlich höher ist, als der Aufwand für die »einfacheren« Level. Auch das Starten einer Transaktion kostet Zeit.

3.2.3 Lösung

Aus den oben genannten Gründen sollten die Transaktionen immer nur dann gestartet werden, wenn diese auch tatsächlich benötigt werden. Bei Methoden, die lediglich auf nichttransaktionsfähige Ressourcen zugreifen oder keine Transaktionen benötigen, ist dieser Mehraufwand unnötig. Auch wenn man auf eine transaktionelle Ressource lesend zugreift, wie z.B. eine relationale Datenbank, ist eine Transaktion meistens unnötig. Die Frage, die sich hier stellt, lautet: »Wie viel Zeit kostet eine unnötige Transaktion?«

3.2.4 Praxis

Um die Frage zu beantworten schauen wir uns mal eine Stateless SessionBean `TimeTaskBean` an. Diese implementiert eine Geschäftslogikmethode `timeTask`. Bei jedem Aufruf wird die aktuelle Systemzeit des Servers an den Client zurückgegeben.

```
public class TimeTaskBean implements SessionBean {
    private SessionContext ctx;
    public long timeTask(){return System.currentTimeMillis();   }
    public void ejbCreate(){    }
    public void ejbActivate(){    }
    public void ejbPassivate(){    }
    public void ejbRemove() {    }
    public void setSessionContext(SessionContext param0) {    }
}
```

Da hier weder eine transaktionsfähige Resource beteiligt ist, noch schreibend auf externe Systeme geschrieben wird, ist eine Transaktion offensichtlich unnötig.

3.2.5 Performance

Für die Messung der Performance wurde ein Client implementiert, der die angegebene Methode 1.000-Mal aufruft.

```
public class TransactionPerformanceClient {
    public static void main(String[] args) throws Exception{
        String argument = null;
        if(args == null || args.length != 1){
        System.out.println("TransactionPerformanceClient: Please provide at least
one parameter !");
            return;
        }
      Context context = new InitialContext();
        Object ref = context.lookup("TimeTaskBean");
        TimeTaskHome timeTaskHome =
(TimeTaskHome)PortableRemoteObject.narrow(ref,TimeTaskHome.class);
        TimeTask timeTask     = timeTaskHome.create();
        int loops = Integer.parseInt(argument);
        for(int i=0;i<loops;i++){
                  timeTask.timeTask();
        }
    }
}
```

Dabei wurde dieser Client innerhalb der JProbe Suite 3.0 Server Side Edition der Firma Sitraka (*www.sitraka.com*) ausgeführt. Unsere `TimeTaskBean` wurde auf dem Weblogic Server 7.0 deployed. Zuerst wurde die Methode `timeTask` der Bean mit dem Transaktionslevel `NEVER` aufgerufen.

```
  <container-transaction>
      <method>
        <ejb-name>TimeTaskBean</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>timeTask</method-name>
        <method-params />
```

```
    </method>
    <trans-attribute>Never</trans-attribute>
  </container-transaction>
```

Es sollte also keine neue Transaktion gestartet werden. Mit diesem Level könnte auch diese Bean in die »Produktion« deployed werden. Bei dieser Methoden werden keine Transaktionen benötigt.

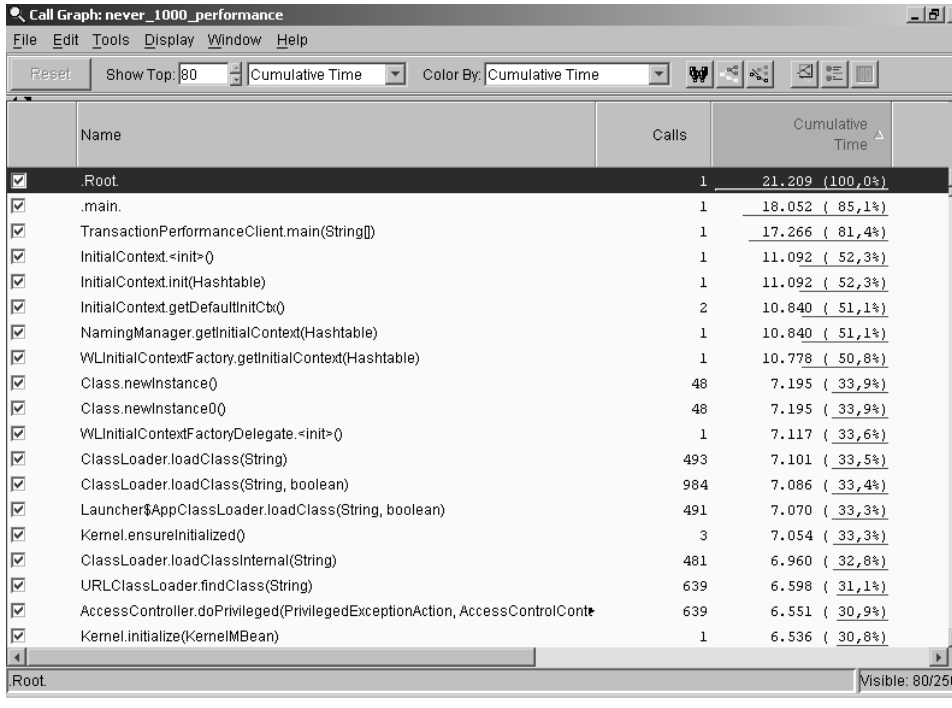


Abbildung 3.8: Die Aufrufe der Methode timerTask mit dem Transaktionslevel NEVER

Die Abbildung 1.1 zeigt, dass ein Löwenanteil der Gesamtzeit für die Initialisierung der JNDI-Umgebung benötigt wird. Für die 1.000 Methodenaufrufe werden insgesamt 21.209 msec benötigt.

Wie die Abbildung 1.9 zeigt, werden für die 1.000-Aufrufe auf der Clientseite 3.928 msec benötigt. Nun wird die gleiche `TimeTaskBean` noch einmal mit dem Transaktionslevel `REQUIRED` deployed.

```
<container-transaction>
   <method>
     <ejb-name>TimeTaskBean</ejb-name>
     <method-intf>Remote</method-intf>
```

```
      <method-name>timeTask</method-name>
      <method-params />
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
```

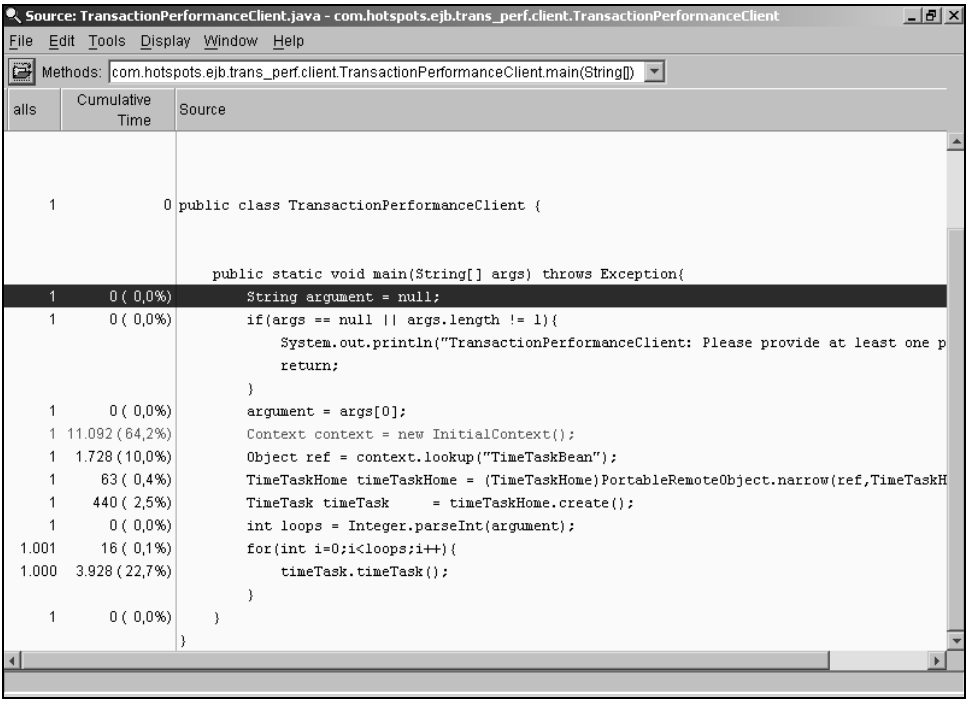


Abbildung 3.9: Aufrufe der Methode timerTask mit dem Transaktionlevel NEVER

Die Gesamtzeit der Anwendung beläuft sich hier auf 21.806 msec. Mit dieser Einstellung benötigt der Client ca. 0,5 sec länger.

Mit Hilfe der Abbildung 1.11 lässt sich einfach feststellen, dass für die 1.000 Methodenaufrufe 4.399 msec benötigt wurden. Für diese Aufrufe mit dem Transaktionslevel `REQUIRED` wurden also rund 0,5 sec mehr benötigt.

Bei den Einstellungen des Launchpads wurde die Einstellung »CPU Time« vorgenommen. Die Konfiguration ermöglicht die Messung der Zeit in Prozessorzyklen. Aus diesem Grund kann angenommen werden, dass diese Zeit nicht durch »Nebenprozesse« verfälscht werden kann. Die Zeitdifferenz von rund einer halben Sekunde wird für den Mehraufwand benötigt, der für die Verwaltung der Transaktionen notwendig ist.

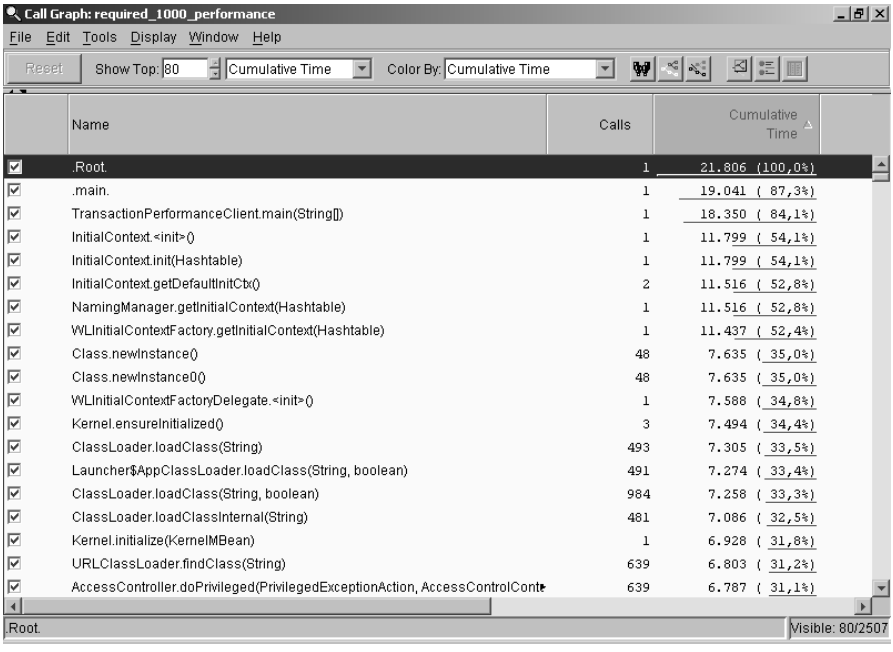


Abbildung 3.10: Die Aufrufe der Methode timerTask mit dem Transaktionslevel REQUIRED

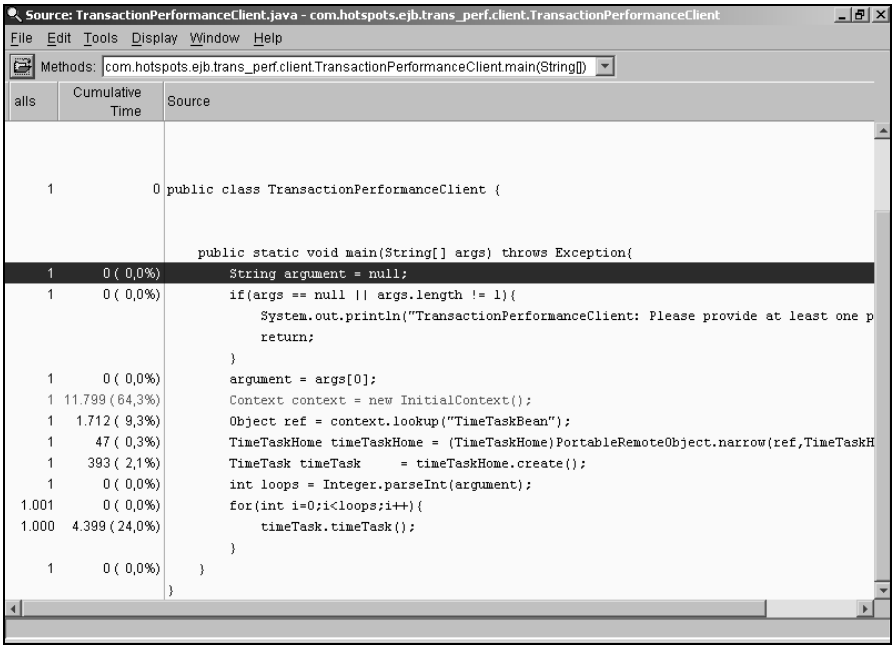


Abbildung 3.11: Die Aufrufe der Methode timerTask mit dem Transaktionslevel REQUIRED

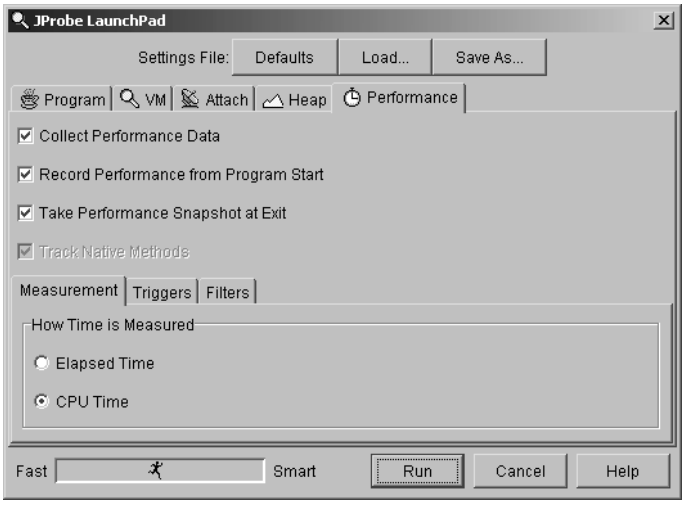


Abbildung 3.12: JProbe Projekteinstellungen

Obwohl diese Zeit von einer halben Sekunde bei der Betrachtung der Gesamtlaufzeit (ca. 22 sec) nicht ins Gewicht fällt, sollte diese trotzdem berücksichtigt werden. Die Gesamtzeit lässt sich durch das Caching des `InitialContext` und der Home- sowie Remote-Interfaces deutlich reduzieren. Dies ist mit der Verwendung des Service-Locator-Patterns möglich. Der Service-Locator cached teure Ressourcen und entspricht selber dem Singleton GoF-Pattern. In einer JVM existiert lediglich eine einzige Instanz dieser Komponente.

Allein bei der Betrachtung der Aufrufe ergibt sich ein Performance-Unterschied von rund 10,7%. Rein aus der Performancesicht lohnt sich also die Differenzierung zwischen notwendigen und überflüssigen Transaktionen. Wenn man noch berücksichtigt, dass auch der Applikationsserver mit überflüssigen Transaktionen »belastet« wird, lohnt sich die Optimierung der Deklarativen Programmierung auf jeden Fall. Ferner spielt hier die Umgebung eine große Rolle. Mit der J2EE 1.2.1 RI (Reference Implementation) betrug der Performance-Unterschied mehr als 40%.

3.3 Wie kritisch sind die Stateful SessionBeans für die Performance wirklich?

3.3.1 Problemstellung

Für die Modellierung von Prozessen oder Aktivitäten sollten die SessionBeans verwendet werden. Die EntityBeans sollten nur für die persistente Speicherung von Objektinstanzen dienen. Die SessionBeans können in folgende Kategorien eingeteilt werden:

Kategorie	**Merkmale**
Stateless	Diese Variante ist zustandslos. Aus diesem Grund sind die einzelnen Instanzen, auch zwischen den Methodenaufrufen, komplett austauschbar. Die Zustandslosigkeit dieser Komponenten ermöglicht auch eine effiziente Wiederwendung bereits erzeugter Instanzen. Mit dem »Pooling« lässt sich die unnötige Erzeugung (»new«-Aufrufe) und die Zerstörung (Garbage Collector) minimieren. Bei diesem Zustand wird der Zustand des Clients verstanden. Eine Stateless SessionBean, ähnlich wie ein Servlet, kann über einen eigenen, technischen Zustand verfügen. Ihre Attribute (z.B. Konfiguration, Connection usw.) entsprechen zwar dem Zustand der Bean, nicht aber dem Zustand des Aufrufers (Clients).
Stateful	Die Stateful SessionBeans sind in der Lage den Zustand der Clients auf der Server-Seite zu speichern. Diese Eigenschaft wird mit der 1:1-Beziehung zwischen dem Client und dem Server ermöglicht. Durch die Zustandsbehaftung und die Abhängigkeit jeder Instanz von dem Client ist diese Variante nicht »poolbar«. Jede Anforderung einer Bean entspricht einer Erzeugung einer neuen Instanz. Nach dem Aufruf `remove` wird diese gerade erzeugte Instanz nicht um eine Wiederverwendung zu ermöglichen, in den Pool zurückgelegt, sondern zerstört.
Message Driven	Diese Unterkategorie der SessionBeans verleiht einem Server asynchrone Fähigkeiten. Nur mit einer MDB ist es möglich, asynchron mit dem Applikationsserver zu kommunizieren. Technisch gesehen handelt es sich um eine zustandslose EJB, die das Interface `javax.jms.MessageListener` und `javax.ejb.MessageDrivenBean` implementieren muss. Eine Message Driven Bean wird ohne das Remote- und Home-Interface deployed. Die Methode `onMessage()` wird asynchron von dem EJB-Container beim Eintreffen einer Message aufgerufen. Die Performance dieser Bean ist sehr gut. Da diese über kein Remote-Interface verfügt, eignet sie sich nur bedingt für die Implementierung von Business-Schnittstellen.

Bevor größere J2EE-Projekte gestartet werden, wird oft versucht, eine Vorgehensweise oder Architektur zu definieren. Eine der Fragen, die sich in dem Moment stellen, ist: Sollen Stateful SessionBeans verwendet werden, oder soll ausschließlich mit Stateless SessionBeans gearbeitet werden?

3.3.2 Technischer Hintergrund

Nun, diese Frage hängt natürlich maßgeblich von der zu implementierenden Anwendung ab. Bei zustandslosen Anwendungen, wie z.B. einem Aktienticker, gibt es keine Notwendigkeit für die Benutzung der Stateful SessionBeans. Allerdings finden wir solche Anwendungen in der Praxis nur sehr selten. Eine typische J2EE-Anwendung ist personalisierbar, sodass der Applikationsserver genau wissen muss, mit welchem Benutzer gerade kommuniziert wird. Ab diesem Zeitpunkt muss auch der Zustand des Benutzers auf dem Applikationsserver verwaltet werden. Ohne die Benutzung einer Stateful SessionBean ist die Ablage des Benutzerzustands nur in der `HttpSession` des Web-Containers, oder direkt in einer Datenbank möglich. In solchen Fällen sollte

auf Stateful SessionBeans nicht verzichtet werden, da ansonsten bereits vorhandene Standardfunktionalität von dem Entwickler nachgebildet wird. Die Entscheidung, ob mit Stateful oder Stateless Beans modelliert wird, sollte also nicht projektübergreifend, sondern abhängig von Use Case getroffen werden.

Wenn man die reine »Architektensicht« außer Acht lässt und sich wieder auf die technischen Gegebenheiten konzentriert, spielt die Skalierbarkeit für die Strategieentscheidung eine große Rolle. Durch die Zustandsbehaftung der Stateful SessionBeans ist ihre Skalierbarkeit wesentlich schlechter als die der Stateless SessionBeans. Für jeden Client, im Gegensatz zu den Stateless SessionBeans, muss eine Instanz der benötigten Bean erzeugt werden. Bei den Stateless SessionBeans entscheidet der Applikationsserver, ob die aktuelle Größe des Instanzenpools ausreichend ist oder nicht. Es können jederzeit neue Instanzen von dem Applikationsserver erzeugt werden. Im Allgemeinen lässt sich feststellen, dass mit der gleichen Anzahl von SLSB wesentlich mehr Client-Requests abgearbeitet werden können, als mit der gleichen Anzahl der SFSB.

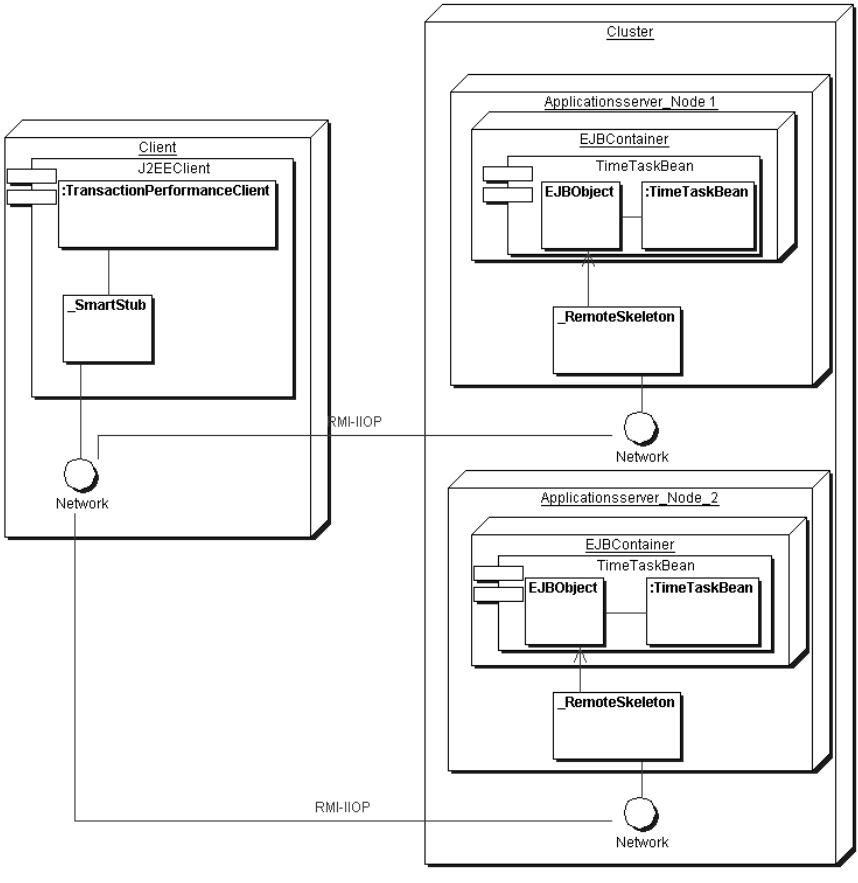


Abbildung 3.13: Das Deployment einer SessionBean in einer geclusterten Umgebung

Ein weiterer Nachteil der SFSB ist die Replizierungsproblematik im Cluster-Betrieb. Wie die Abbildung 1.13 zeigt, werden die Beans redundant auf jedem vorhandenen Cluster-Knoten deployed. Der Einsatz der sog. »Smart Stubs« ermöglicht eine echtes Load Balancing bereits auf der Client-Seite. Auf eine SLSB kann wahllos zugegriffen werden, weshalb die Lastverteilung die meisten Performancevorteile bringt und sich am einfachsten implementieren lässt.

Das Clustering einer SFSB gestaltet sich wesentlich komplizierter, da diese Komponente über client-bezogenen Zustand verfügt und somit nicht mit einer anderen Instanz austauschbar ist. Grundsätzlich bleiben dem Applikationsserverhersteller lediglich folgende Alternativen übrig:

- Replizierung des Zustands der SFSB über alle Clusterknoten hinweg,
- Session-Affinität, also die Rückkehr des Clients zu vorher besuchten Knoten.

Da die Session Affinität bei einem Hardware-Ausfall eines Knotens auch zu einem Ausfall der kompletten Anwendung führen kann, wird diese Strategie oft nur als ein zusätzliches Mittel für die Performance-Steigerung des Clusters verwendet. Eine echte Ausfallsicherheit kann nur mit der Replizierung des Zustandes der Bean erreicht werden. Obwohl dieses Verfahren mit weiteren »Features« optimiert werden kann, bedeutet eine Replizierung des Zustands einer SFSB zusätzlichen Mehraufwand für den Applikationsserver und somit eine schlechtere Performance.

3.3.3 Performance

Für die Performance-Messung lässt sich die bereits für Testzwecke installierte `TimeTaskBean` wieder verwenden. Bei der ersten Performancemessung werden die Aufrufe `create` und `remove` mitgemessen, um den Unterschied zwischen der Stateless- und Stateful-Variante zu testen. Dabei wird die `TimeTaskBean` zuerst als Stateless und dann als Stateful Variante deployed. Die Performance kann beide Male mit einem Client gemessen werden.

```
public class StatefulStatelessPerformance {
    public static void main(String[] args) throws Exception{
       String argument = null;
       //…
       argument = args[0];
       Context context = new InitialContext();
       Object ref = context.lookup("TimeTaskBean");
       TimeTaskHome timeTaskHome =
(TimeTaskHome)PortableRemoteObject.narrow(ref,TimeTaskHome.class);
       TimeTask timeTask     = null;
       int loops = Integer.parseInt(argument);
       long start = System.currentTimeMillis();
       for(int i=0;i<loops;i++){
```

```
            timeTask = timeTaskHome.create();
        timeTask.timeTask();
            timeTask.remove();
         }
    System.out.println(loops + " loops in " + (System.currentTimeMillis()-start) +
" milliseconds !");
    }
}
```

Auch in diesem Beispiel wird die Performance des Clients mit JProbe 3.0 Server Side Edition gemessen. Nach jeder Messung wird der Applikationsserver (BEA WebLogic 6.1) heruntergefahren, um die Optimierungsversuche des Servers (z. B. Pooling, Verbesserung der Laufzeit-Performance mit Hotspot usw.) zu verhindern.

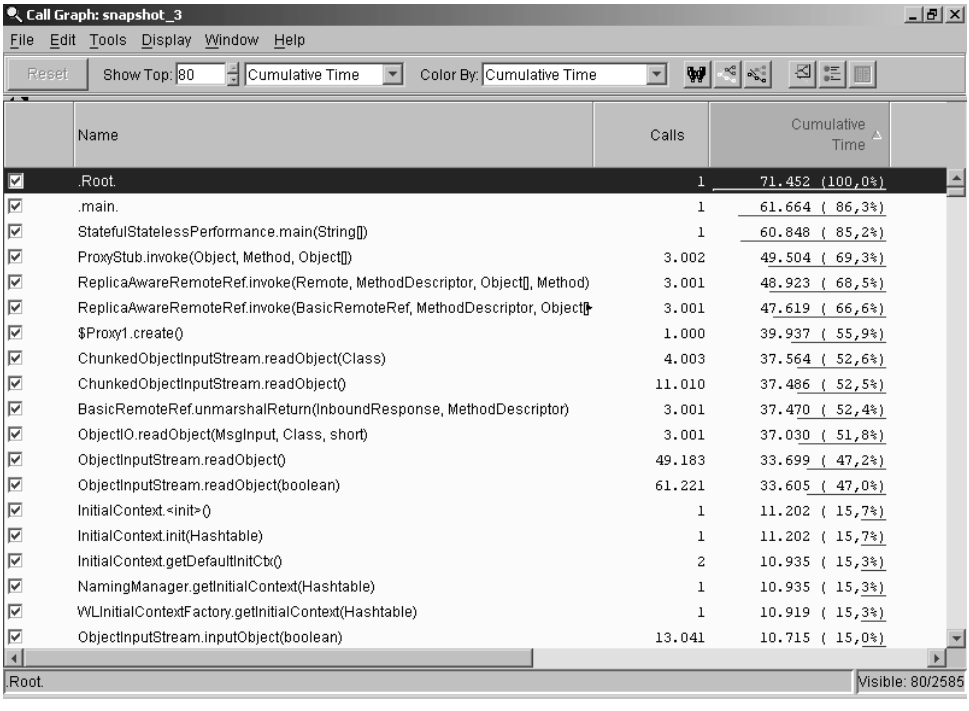


Abbildung 3.14: Gesamtübersicht der Aufrufe der SLSB

Der Client ruft die Methode `timeTask` beim ersten Versuch 1.000-Mal auf. Die Gesamtlaufzeit beträgt 71.452 msec. Die Abbildung 1.15 zeigt hier auch sehr deutlich, dass die Initialisierung des `InitialContext` mit 11.202 msec ganze 18,4% der Gesamtzeit für sich beansprucht. Der Einsatz des Service-Locator-Patterns könnte hier die Performance der Anwendung deutlich erhöhen.

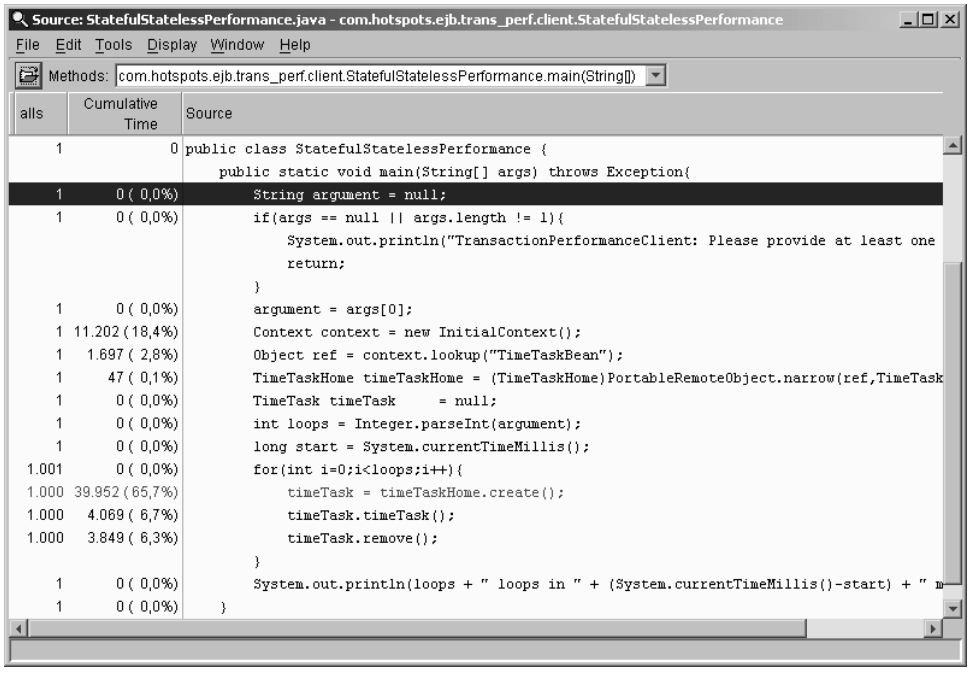


Abbildung 3.15: Source-Code-Ansicht des Clients der SLSB

Nachdem die Performance einer SLSB gemessen wurde, wird die `TimeTaskBean` noch einmal als eine SFSB deployed.

```
<enterprise-beans>
    <session>
      <display-name>TimeTaskBean</display-name>
      <ejb-name>TimeTaskBean</ejb-name>
      <home>com.hotspots.ejb.trans_perf.TimeTaskHome</home>
      <remote>com.hotspots.ejb.trans_perf.TimeTask</remote>
      <ejb-class>com.hotspots.ejb.trans_perf.TimeTaskBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
      <security-identity>
        <description></description>
        <use-caller-identity/>
      </security-identity>
    </session>
  </enterprise-beans>
```

Der gleiche Client greift erneut auf diese Bean zu. Auch hier wird die Performance mit JProbe gemessen. Nach der genauen Betrachtung der Gesamtübersicht fällt sofort auf, dass die Ausführung der SFSB deutlich performanter ist als die der SLSB Variante. Die Gesamtzeit betrug nur 63.471 msec.

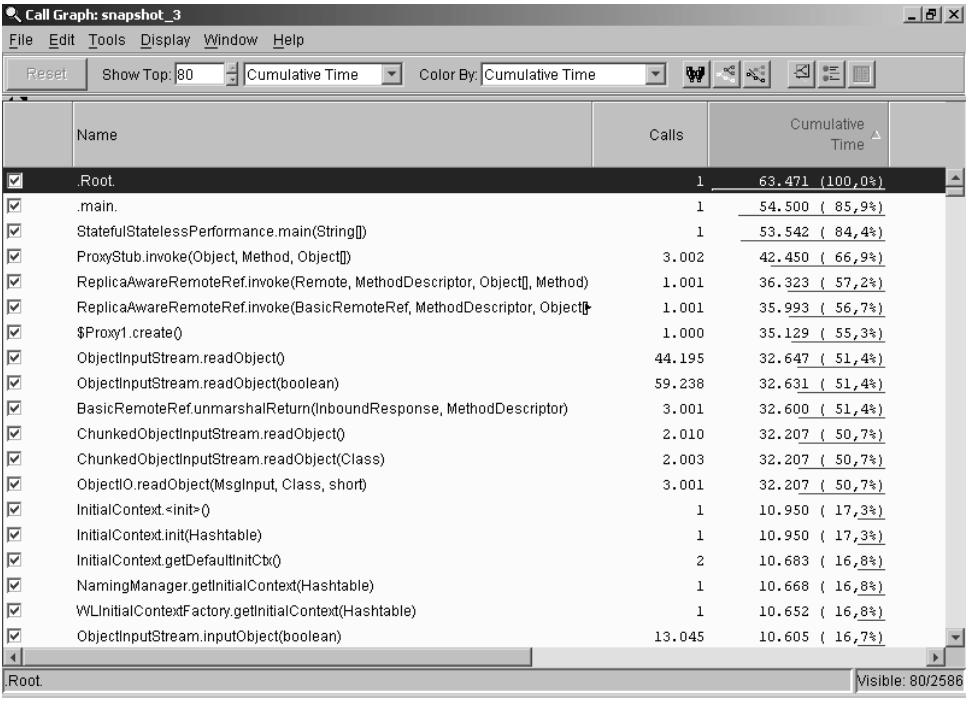


Abbildung 3.16: Gesamtübersicht der Aufrufe der SFSB

Für die Suche nach möglichen Ursachen bietet sich die Quellcode-Übersicht des Clients an. Hier sieht man deutlich, dass sowohl die Erzeugung (35.161 msec), als auch die Geschäftslogikaufrufe (3.079 msec) und die Zerstörung (2.702 msec) deutlich schneller sind als die der SLSB.

Obwohl die bisherigen Untersuchungen sehr interessant sind, entsprechen diese nicht einer »real-World« Anwendung. Der Client geht hier ziemlich verschwenderisch mit den teuren Ressourcen wie den Remote-Interfaces der EJB um. Die Performance der Gesamtanwendung lässt sich auf einfache Art und Weise erhöhen, in dem die Instanz des Remote-Interfaces auf dem Client gecached wird. Die Aufrufe der Methode `create` und `remove` werden aus der `for`-Schleife entfernt. Lediglich die Geschäftslogik der Bean wird nun mehrmals (1.000-Mal) aufgerufen.

```
public class StatefulStatelessPerformanceReal {
    public static void main(String[] args) throws Exception{
        String argument = null;
//…
        argument = args[0];
             Context context = new InitialContext();
        Object ref = context.lookup("TimeTaskBean");
```

```
        TimeTaskHome timeTaskHome =
(TimeTaskHome)PortableRemoteObject.narrow(ref,TimeTaskHome.class);
        TimeTask timeTask     = null;
        int loops = Integer.parseInt(argument);
        long start = System.currentTimeMillis();
        timeTask = timeTaskHome.create();
        for(int i=0;i<loops;i++){
                    timeTask.timeTask();
        }
        timeTask.remove();
        System.out.println(loops + " loops in " + (System.currentTimeMillis()-
start) + " milliseconds !");
    }
}
```

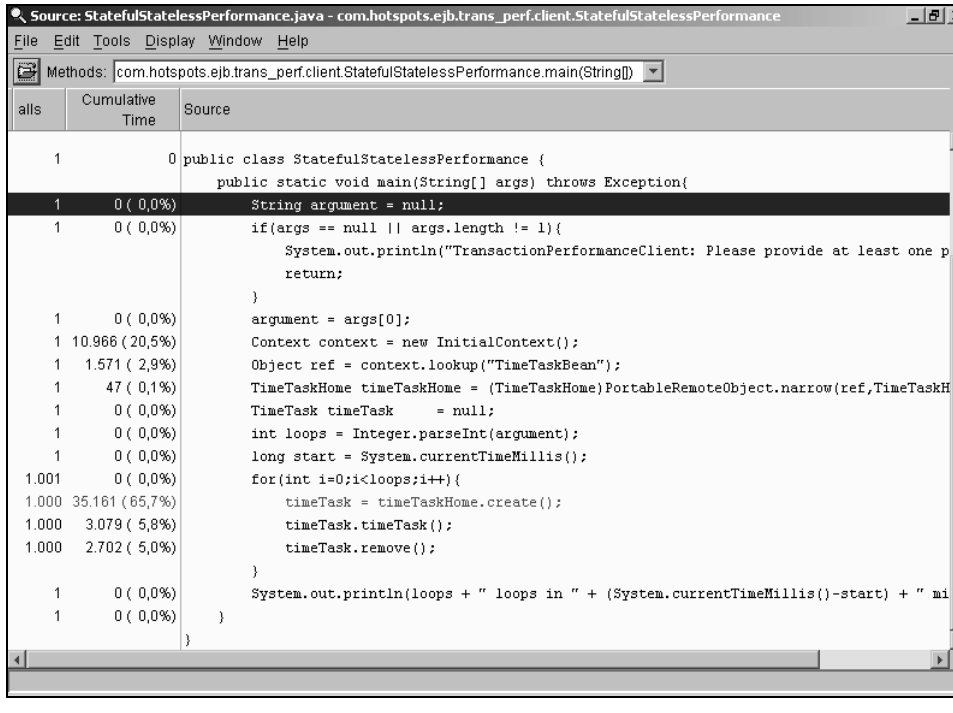


Abbildung 3.17: Source-Code-Ansicht des Clients der SFSB

Zuerst wird die Performance der SLSB untersucht. Wie erwartet konnte man mit dieser einfachen Maßnahme die Geschwindigkeit der Anwendung deutlich erhöhen. Nun beträgt die Gesamtzeit des SLSB-Clients 20.942 msec.

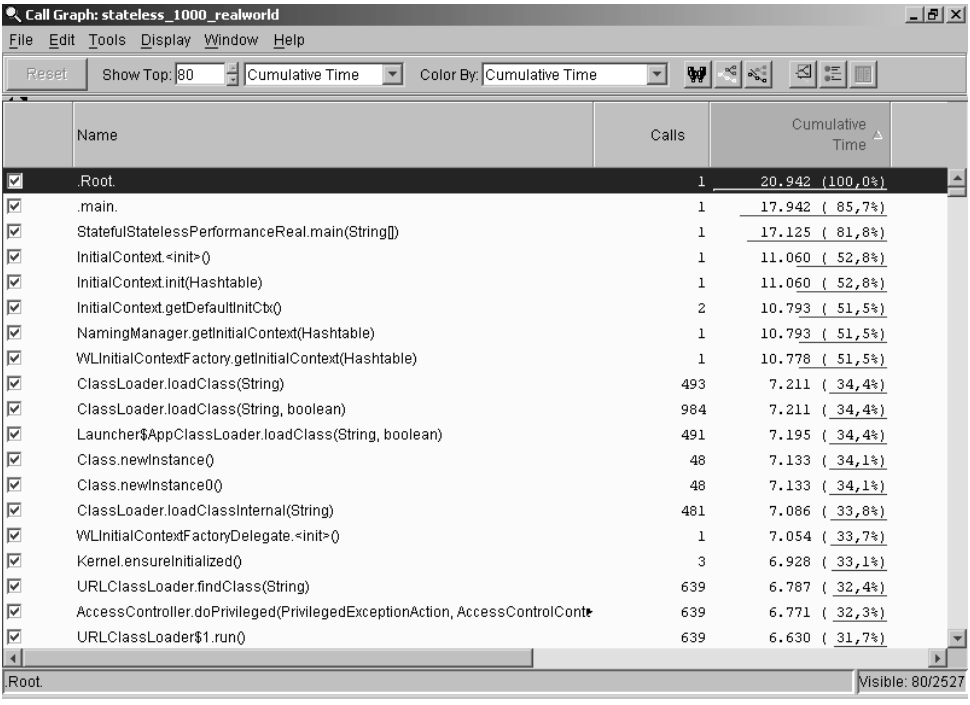


Abbildung 3.18: Gesamtübersicht der Aufrufe der SLSB

Bei der genauen Betrachtung der Laufzeiten einzelner Methoden fällt auch hier auf, dass die Erzeugung des `InitialContext` sehr lange dauert. Die »Erzeugung« des Remote-Interfaces ist vergleichsweise schnell. Es dauert ca. 471 msec. Die 1.000 Aufrufe der Methode `timeTask` dauern hier 3.833 msec.

Nun betrachten wir mal näher die Performance der Stateful-Variante. Die Gesamtzeit beträgt 20.298 msec, also entgegen den Erwartungen etwas schneller als die der SLSB.

Sowohl die Erzeugung der Bean als auch die Aufrufe der Methode `timeTask` sind schneller als die der SLSB. Die Performance einer SFSB ist also mindestens so gut wie die einer SLSB.

Um die Performance der Bean unter reellen Bedingungen zu testen, wurde ein multithreaded Client entwickelt. Somit können mehrere virtuelle Clients »emuliert« werden. Beim Start des Clients kann sowohl die Anzahl der Business-Aufrufe als auch die Anzahl der gleichzeitigen Zugriffe (Threads) mitübergeben werden.

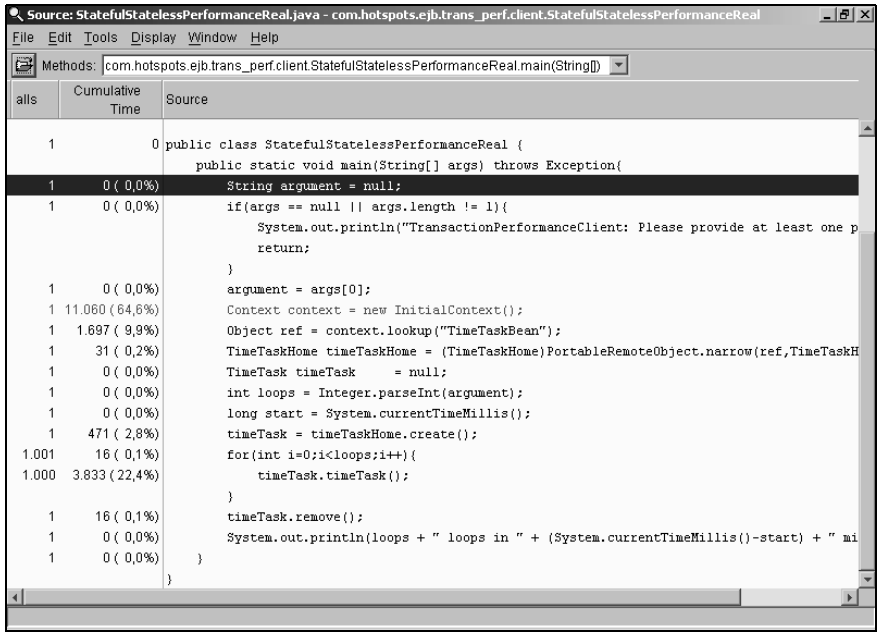


Abbildung 3.19: Source-Code-Ansicht des Clients der SLSB

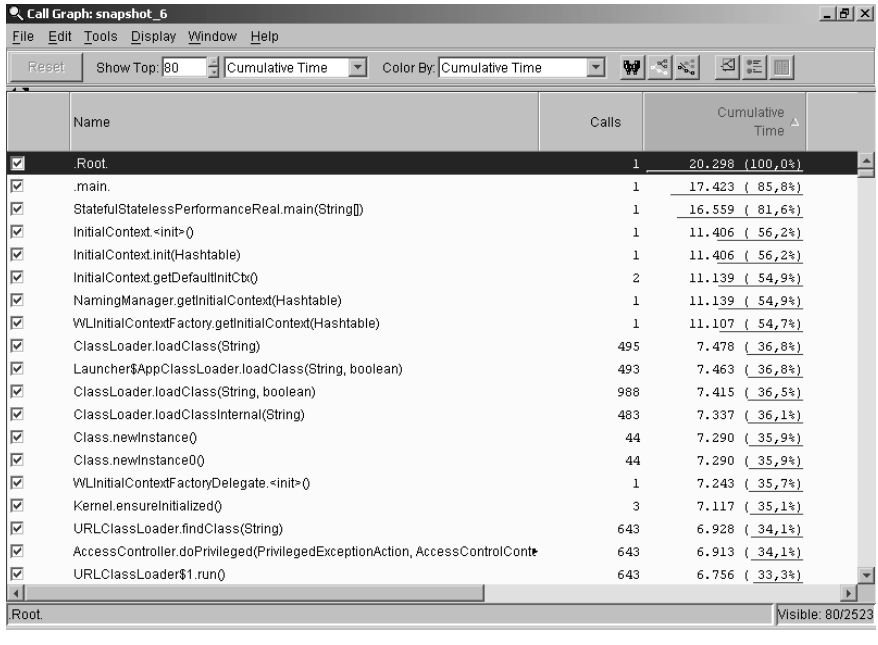


Abbildung 3.20: Gesamtübersicht der Aufrufe der SFSB

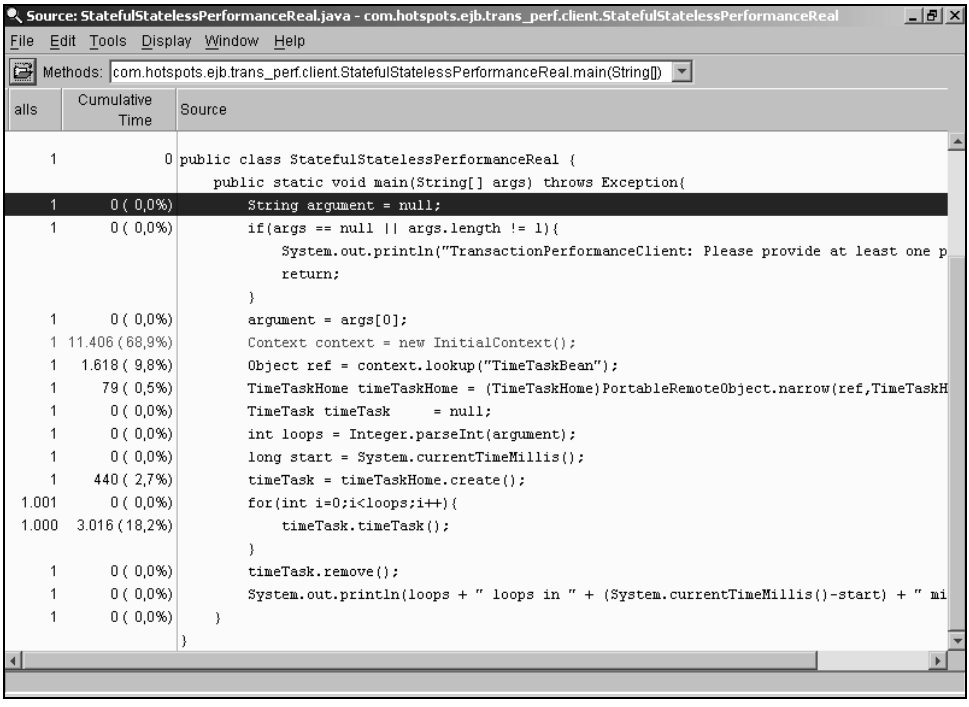


Abbildung 3.21: Source-Code-Ansicht des Clients der SFSB

```
public class StatefulStatelessPerformanceRealThreaded implements Runnable {
    private TimeTask timeTask         = null;
    private TimeTaskHome timeTaskHome = null;
    private int loops                 = -1;
    private int threadnumber          = -1;
    public StatefulStatelessPerformanceRealThreaded(int loops, int threadnumber)
throws Exception {
        this.loops        = loops;
        this.threadnumber = threadnumber;
        Context context = new InitialContext();
        Object ref = context.lookup("TimeTaskBean");
        this.timeTaskHome = (TimeTaskHome)PortableRemoteObject.narrow(ref,
TimeTaskHome.class);
        System.out.println("Thread " + threadnumber + " initialized !");
    }
    public void run() {
        try {
            timeTask = timeTaskHome.create();
            for (int i = 0; i < this.loops; i++) {
                timeTask.timeTask();
                Thread.currentThread().yield();
            }
            timeTask.remove();
```

```
        } catch (Exception e) {
    System.err.println("[….run()] " + e);
        }
     }
    public static void main(String[] args) throws Exception {
        String loopsArg   = null;
        String threadArg  = null;
//…
        threadArg = args[0];
        loopsArg = args[1];
        System.out.println("Number of threads: " + threadArg + " number of loops "
+ loopsArg);
        int loops   = Integer.parseInt(loopsArg);
        int threads = Integer.parseInt(threadArg);
        for (int i = 0; i < threads; i++) {
            Thread thread = new Thread(new
StatefulStatelessPerformanceRealThreaded(loops,i));
            thread.start();
        }
    }
}
```

Die Performance der stateless- und der stateful-Variante wurde erneut getestet. Der erste Test wurde mit 33 Threads und 33 Aufrufen der Methode `timeTask` durchgeführt. Die Aufrufe konnten von einer SLSB-Instanz abgearbeitet werden. Diese Aufgabe konnte in 59.355 msec erledigt werden. Lediglich bei 100 gleichzeitigen Zugriffen hat sich der Applikationsserver für eine weitere Instanz der `TimeTaskBean` entschieden.

Threads	Schleifendurchgänge	Gleichzeitig aktive SLSB	Benötigte Zeit
33	33	1	59.355
10	100	1	43.377
100	10	2	82.057
20	50	1	54.972
50	20	1	59.261

Die `TimeTaskBean` wurde erneut deployed, diesmal als eine SFSB. Auch hier wurde die Performance mit 33 Threads und 33 Aufrufen gemessen. Für diese Aufgabe musste der Applikationsserver 33 Instanzen erzeugen, um das Konversationsgedächtnis der Clients zu verwalten. Jeder Client kommuniziert also mit einer SFSB-Instanz. Interessanterweise ist die Performance der SFSB in allen Fällen deutlich besser als die der SLSB.

Threads	Schleifendurchgänge	Benötigte Zeit
33	33	49.190
10	100	34.124
100	10	60.643
20	50	41.570
50	20	45.341

Um die Tests noch realistischer zu gestalten, wird die `TimeTaskBean` künstlich »belastet«. In der `timeTask` Methode wird ein `String` 2.000-Mal addiert. Somit dauert jeder `timeTask`-Aufruf deutlich länger.

```
public long timeTask(){
    String testString ="dummy";
    for(int i=0;i<2000;i++){
                testString+="g";
    }
   return System.currentTimeMillis();
}
```

Interessanterweise werden jetzt deutlich mehr gleichzeitige Instanzen von dem Applikationsserver gestartet. Die 33 Threads werden mit 33 SLSB-Instanzen parallel abgearbeitet. Der Test mit den 100 gleichzeitigen Clients konnte allerdings mit 45 Instanzen bewältigt werden.

Threads	Schleifendurchgänge	Gleichzeitig aktive SLSB	Benötigte Zeit
33	33	33	35.962
10	100	10	31.594
100	10	45	53.998
20	50	20	48.896
50	20	46	37.501

Auch jetzt ist der Applikationsserver gezwungen, für jeden Client eine SFSB zu erzeugen. Die 33 Durchgänge dauern hier 38.460 msec, also etwas länger als die der SLSB. Alle anderen Tests benötigen etwas weniger Zeit, wobei der Performance-Unterschied nicht mehr so groß ist.

Threads	Schleifendurchgänge	Benötigte Zeit
33	33	38.460
10	100	29.065
100	10	47.446

Threads	Schleifendurchgänge	Benötigte Zeit
20	50	28.515
50	20	37.313

3.3.4 Fazit

Die Performance-Test mit JProbe ServerSuite 3.0 und WebLogic Applikationsserver 6.1 haben gezeigt, dass die SFSB mindestens so »schnell« sind wie ihre SLSB Konkurrenz. Dies widerspricht der weit verbreiteten Meinung, dass man nur mit SLSB hohe Performance erzielen kann. In vielen Projekten wird aus Performance-Gründen nur mit SLSB gearbeitet. Allerdings gibt es nur wenige Internet-Anwendungen, die tatsächlich auch zustandslos sind. Oft wird der Zustand mit einigen Workarounds mit den SLSB emuliert, um den Projektvorgaben gerecht zu werden, was die Performance verschlechtert bzw. viel langsamer ist als der Einsatz der SFSB.

Diese Tests bestätigen, dass die Architektur der Anwendung im Vordergrund stehen sollte. Falls die nichtfunktionalen Anforderungen (Wiederverwendbarkeit, Modularität, Austauschbarkeit der Komponenten) der Anwendung richtig umgesetzt wurden, kann man die Performance auch nachträglich verbessern, in dem die kritischen »Hotspots« einfach ausgetauscht werden.

3.4 Benutzung von WebServices für den Zugriff auf eine Stateless SessionBean statt eines konventionellen Zugriffs

3.4.1 Problemstellung

Die Kommunikation mit den EJBs erfolgt in den meisten Fällen über das IIOP-Protokoll (Internet InterOrb Protocol). Genauer betrachtet handelt es sich dabei um die RMI-IIOP-Variante. Dabei wird das ursprüngliche JRMP-Protokoll (Java Remote Method Invocation Protocol) mit dem IIOP-Protokoll ausgetauscht. Bei beiden handelt es sich um ein Objektprotokoll zwischen Stubs und Skeletons. Bei der Kommunikation durch eine Firewall kann es zu Problemen kommen, da die Stubs die Ports dynamisch verwalten. Wie bei den meisten Firewalls müssen die Ports aber fest eingetragen werden.

3.4.2 Technischer Hintergrund

Für die dynamische Vergabe der Ports sind im Wesentlichen die Stubs und Skeletons verantwortlich. Diese abstrahieren das Netzwerk und bieten dem Client einen transparenten Zugriff auf den Server. Diese Konstellation entspricht dem Proxy-Pattern (GoF).

Der Stub repräsentiert auf dem Client die Instanz des `RealSubjects` und ist quasi der Stellvertreter dieser Instanz. Der Client weiß allerdings nicht, dass eigentlich über diese Proxy-Instanz kommuniziert wird. Der Proxy kennt das jeweilige Protokoll sehr gut und weiß, wie man am besten mit dem Skeleton kommunzieren muss. Dieser »hält« die `RealSubject`-Instanz und ruft die von dem Stub geforderten Methoden auf. Dabei wartet diese, bis die `RealSubject`-Instanz aus dem Aufruf zurückkehrt und gibt den Rückgabewert an den Stub zurück. Dieser hat den Client auch bis zu diesem Zeitpunkt blockiert. Der Stub gibt wiederum die Ergebnisse des Aufrufs und ggf. alle Exceptions, die auf dem Server aufgetreten sind, an den Client zurück.

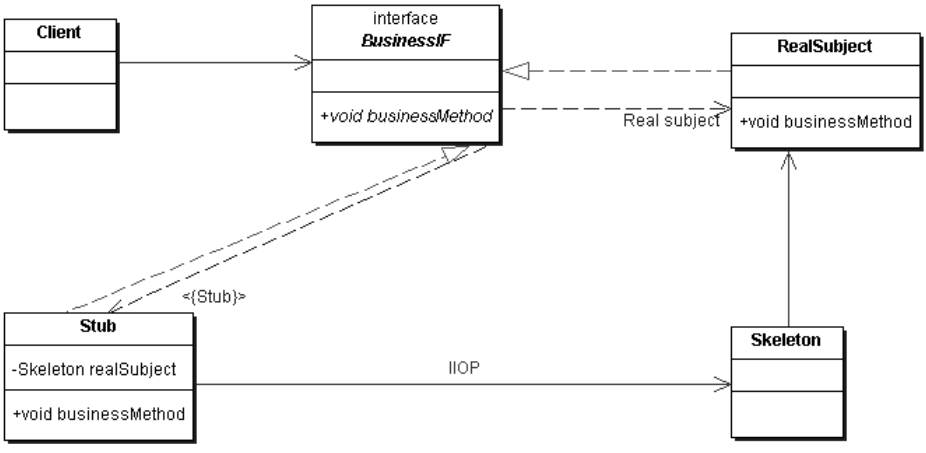


Abbildung 3.22: Das Proxy-Pattern im RMI/CORBA Umfeld

Abhängig vom Protokoll können allerdings weitere Ports zwischen dem Stub und dem Skeleton geöffnet werden. Diese dienen meistens administrativen Zwecken. So benachrichtigt der Client den Server, dass er noch aktiv ist. Mit den einfachen »Pings« wird beispielsweise die verteilte Garbage Collection gesteuert. Diese DGC (Distributed Garbage Collection) wird beispielsweise bei dem klassischen RMI-Protokoll auf diese Art und Weise erledigt. Allerdings »gefällt« diese Vorgehensweise den meisten Firewalls nicht, da diese hier unerlaubte Zugriffe auf den Server vermuten. Außerdem wird ein Server mit der wachsenden Anzahl der Clients unnötig mit der Abarbeitung der »Is-Alive« Ereignisse belastet.

3.4.3 Lösung

Die Lösung des Problems ist relativ einfach. Man versucht die Firewall-Problematik zu umgehen, indem nur feste, bekannte Ports verwendet werden. Dabei verlässt man sich auf die bereits freigeschalteten Ports, wie die Portnummer 80. Streng genommen versucht man an dieser Stelle die Security-Policy zu umgehen, indem man doch schafft,

die hinter einer Firewall befindliche Geschäftslogik aufzurufen. Der Client schickt über die Port 80 eine Textnachricht, deren Inhalt den Server über die aufzurufende Methode informiert. Der Stub ruft den Skeleton bereits auf dem Applikationsserver auf. Für das Tunneling wird meistens das Simple Object Access Protocol (SOAP) verwendet, das wiederum eine Untermenge (Protokoll) der WebServices ist. Bei den »Textnachrichten« handelt es sich eigentlich um XML-Nachrichten, die zwischen dem Client und einem Servlet ausgetauscht werden. Diese Nachrichten wurden weitgehend standardisiert, sodass auch unterschiedliche Plattformen wie z. B. .NET und J2EE mit diesem Medium miteinander kommunizieren können. Bei dem SOAP-Protokoll handelt es sich keinesfalls um ein Objektprotokoll – es werden hier lediglich die Methoden oder Prozeduren aufgerufen (deswegen der Name RPC). Ferner ist SOAP zustandslos, sodass bei jedem Aufruf der ganze Kontext mitübergeben werden muss.

Da jeder Parameter und Rückgabewert in eine XML-Nachricht umgewandelt werden muss, entsteht hier ein gewisser Kommunikationsmehraufwand. Wie gut ist die Performance dieser Kommunikation? Ist SOAP für die Kommunikation innerhalb einer Komponente sinnvoll?

3.4.4 Praxis

Um die Performance zu testen, wurde eine einfache Stateless SessionBean entwickelt. Diese Bean besteht aus einer einzigen Geschäftslogikmethode `sayHello`, die von einem EJB- und einem Webservice-Client aufgerufen wird. Diese Methode erwartet zwei `Strings` als Parameter, und gibt auch eine `java.lang.String`-Instanz zurück. Die Parameter sind nicht unbendingt für die Abbildung der Geschäftslogik notwendig, sie wurden lediglich für die Messung des Serialisierungsaufwandes gemessen.

```
public class WebserviceTestBean implements SessionBean {
       public void ejbCreate() { }
       public void ejbActivate() { }
       public void ejbPassivate() {}
       public void ejbRemove() {}
       public void setSessionContext(SessionContext arg0) { }
       public String sayHello(String date, String message) {
            return "Message received. Date: " + date + " Message: " + message;
 }
}
```

Es wurde zunächst ein standardmäßiges Archiv `ejb-jar` erzeugt. Mit Hilfe des `servicegen` Ant-Tasks (Weblogic 7.0) wurde die notwendige Webservice-Infrastruktur generiert.

```
<project name="buildWebservice" default="ear">
<target name="ear">
<servicegen destEar="webservice_perf.ear" contextURI="WebServices">
```

```
<service ejbJar="ejb-jar-ic.jar" targetNamespace="http://www.bea.com/webservices/
webservice_test" serviceName="WebserviceTest" serviceURI="/WebserviceTest"
generateTypes="True" expandMethods="True">
<client packageName="com.hotspots.webservice.performance.client"/>
</service>
</servicegen>
</target>
</project>
```

Als Ergebnis entstanden ein EAR-Archiv und ein client-seitiges Archiv, das für die SOAP-Aufrufe notwendig ist.

3.4.5 Performance

Zunächst wurde die Performance mit einem gewöhnlichen Client gemessen. Die Methode `sayHello` wurde 1.000-Mal aufgerufen.

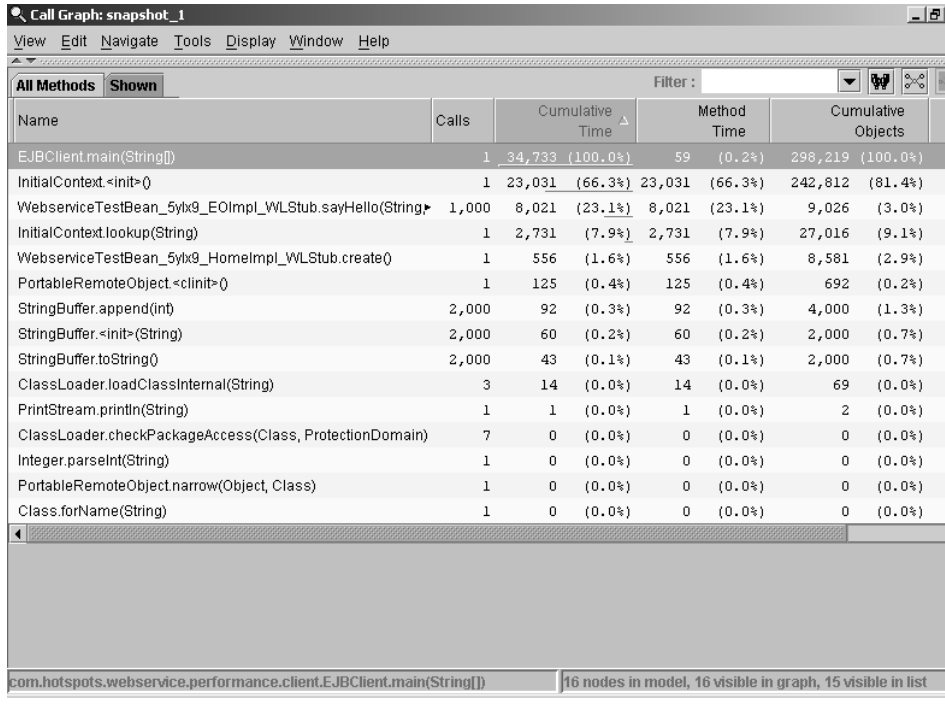
```
package com.hotspots.webservice.performance.client;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import com.hotspots.webservice.performance.WebserviceTest;
import com.hotspots.webservice.performance.WebserviceTestHome;

public class EJBClient {

      public static void main(String[] args) throws Exception{
             if(args.length < 1){
                    System.out.println("Number of loops required: ");
                    System.exit(0);
             }
             nt loops = Integer.parseInt(args[0]);
            Context context = new InitialContext();
            Object ref      = context.lookup("WebserviceTestBean");
            WebserviceTestHome home =
(WebserviceTestHome)PortableRemoteObject.narrow(ref,WebserviceTestHome.class);
             WebserviceTest     test = home.create();
             for (int i = 0; i < loops; i++) {
                  test.sayHello("Hello "+i,"Message: " +i);
             }
             System.out.println("Performance test passed !");

     }
}
```

Die Gesamtzeit der 1.000 Aufrufe betrug 34.733 sec. Die Abbildung 1.23 zeigt, dass auch ein Löwenanteil der Performance an die Initialisierung des `InitialContext` verloren ging. Für diesen Vorgang werden hier ganze 23.031 sec benötigt, also etwa 2/3 der Gesamtzeit.



Name	Calls	Cumulative Time		Method Time		Cumulative Objects	
EJBClient.main(String[])	1	34,733	(100.0%)	59	(0.2%)	298,219	(100.0%)
InitialContext.<init>()	1	23,031	(66.3%)	23,031	(66.3%)	242,812	(81.4%)
WebserviceTestBean_5ylx9_EOImpl_WLStub.sayHello(String	1,000	8,021	(23.1%)	8,021	(23.1%)	9,026	(3.0%)
InitialContext.lookup(String)	1	2,731	(7.9%)	2,731	(7.9%)	27,016	(9.1%)
WebserviceTestBean_5ylx9_HomeImpl_WLStub.create()	1	556	(1.6%)	556	(1.6%)	8,581	(2.9%)
PortableRemoteObject.<clinit>()	1	125	(0.4%)	125	(0.4%)	692	(0.2%)
StringBuffer.append(int)	2,000	92	(0.3%)	92	(0.3%)	4,000	(1.3%)
StringBuffer.<init>(String)	2,000	60	(0.2%)	60	(0.2%)	2,000	(0.7%)
StringBuffer.toString()	2,000	43	(0.1%)	43	(0.1%)	2,000	(0.7%)
ClassLoader.loadClassInternal(String)	3	14	(0.0%)	14	(0.0%)	69	(0.0%)
PrintStream.println(String)	1	1	(0.0%)	1	(0.0%)	2	(0.0%)
ClassLoader.checkPackageAccess(Class, ProtectionDomain)	7	0	(0.0%)	0	(0.0%)	0	(0.0%)
Integer.parseInt(String)	1	0	(0.0%)	0	(0.0%)	0	(0.0%)
PortableRemoteObject.narrow(Object, Class)	1	0	(0.0%)	0	(0.0%)	0	(0.0%)
Class.forName(String)	1	0	(0.0%)	0	(0.0%)	0	(0.0%)

Abbildung 3.23: Direkter Zugriff auf eine SLSB TimeTaskBean

In der Praxis könnte man sich den Mehraufwand für die Initialisierung des `InitialContext` sparen, indem dieser zwischengespeichert wird. Das J2EE-Pattern `ServiceLocator`, »kümmert« sich um den performanten Zugriff auf die in dem JNDI-Namensraum abgelegten Ressourcen. Die 1.000 Aufrufe der Geschäftslogigmethode `sayHello` konnten in 8.260 sec abgearbeitet werden. Da der Client aus einer anderen JVM gestartet wurde (JProbe 4.0), musste die Kommunikation des Clients mit dem Server mit Hilfe der Remote-Interfaces stattfinden. Wenn man bedenkt, dass hier über das Protokoll RMI-IIOP kommuniziert wurde, ist die Performance der Aufrufe erstaunlich gut ausgefallen (ca. 8 msec pro Aufruf).

Nach der Untersuchung der Performance eines EJB-Clients, wurde die Performance eines Webservice-Clients gemessen. Der Applikationsserver wurde nach jeder Messung neu gestartet, um die Optimierungsversuche des Applikationsservers zu unterbinden (Instanzen Pooling, Stub-Caching, Optimierungen der Hotspots JVM). Die Geschäftslogikmethode des Webservice-Client wurde hier ebenfalls 1.000-Mal aufgerufen.

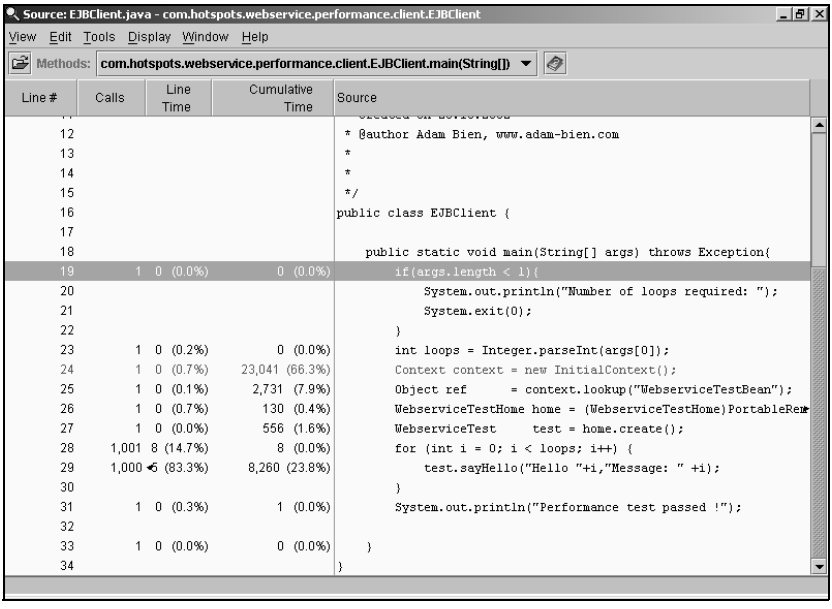


Abbildung 3.24: Detailansicht des SLSB-Clients

```
public final class WebserviceClient {
       public static void main(String[] args) throws Exception{
             if(args.length < 2){
                   System.out.println("Number of loops required: ");
                   System.exit(0);

             int loops = Integer.parseInt(args[1]);
             // Setup the global JAXM message factory
             System.setProperty(
                  "javax.xml.soap.MessageFactory",
            "weblogic.webservice.core.soap.MessageFactoryImpl");
             // Setup the global JAX-RPC service factory
             System.setProperty(
                  "javax.xml.rpc.ServiceFactory",
            "weblogic.webservice.core.rpc.ServiceFactoryImpl");
      WebserviceTest_Impl ws = new WebserviceTest_Impl("http://localhost:7001/
WebServices/WebserviceTest?WSDL");
      WebserviceTestPort port = ws.getWebserviceTestPort();
            for (int i = 0; i < loops; i++) {
                   port.sayHello("Hello "+i,"Message: " +i);
            }
            System.out.println("Performance test passed !");
       }
}
```

Die Gesamtlaufzeit des Clients betrug hier ganze 104.592 sec. Die Performance dieser Variante ist also erheblich schlechter als die Performance einer reinen EJB-Lösung.

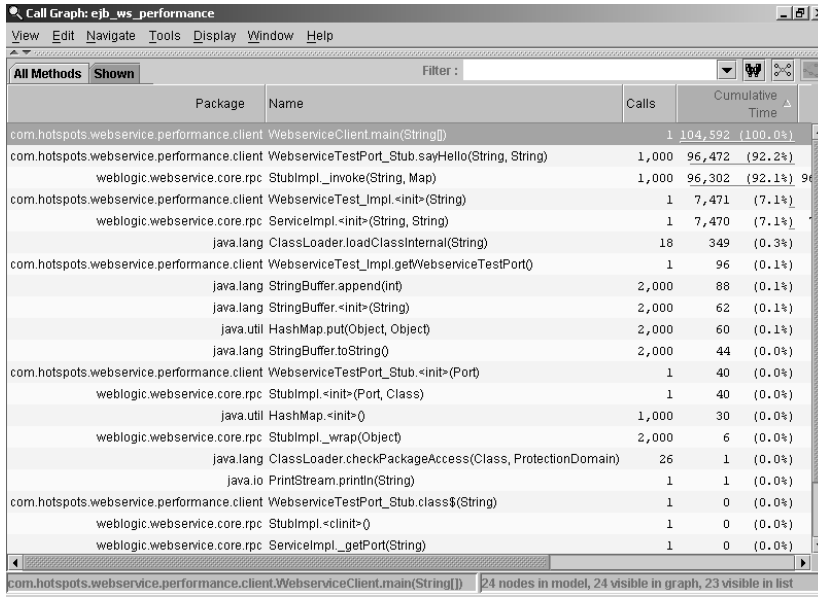


Abbildung 3.25: Zugriff auf die SLSB über RPC

Für die Bereitstellung der Infrastruktur, wurden hier lediglich 7.775 sec benötigt. Ganze 96.707 sec dauerte die Abarbeitung der 1.000 Aufrufe der Methode `sayHello`.

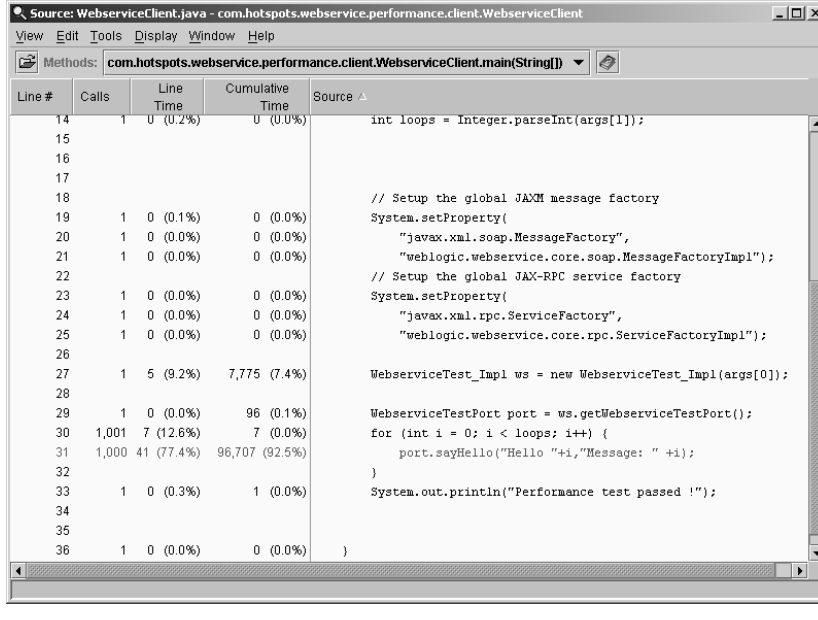


Abbildung 3.26: Detailansicht des RPC-Clients

3.4.6 Fazit

Bei der Betrachtung der Ergebnisse fällt auf, dass die Performance eines Webservices beinahe 12-Mal langsamer ist als ein direkter Zugriff auf die EJB. Der Initialisierungsaufwand eines reinen EJB-Clients ist jedoch fast 3-Mal höher als der eines Webservice-Client.

	EJB-Client	**Webservice-Client**
Gesamtlaufzeit	34,733	104,592
Methodenaufrufe (1000)	8,260	96,707
Initialisierungsaufwand	23,031	7,775

Die Ergebnisse der Performanceanalyse bestätigen, dass sich die Webservices für die Kommunikation zwischen feingranulären Komponenten nicht eignen. Vielmehr sollten diese für die Kommunikation zwischen Gesamtsystemen benutzt werden. Nur »Coarse-Grained« Komponenten sollten mit einer Webservice-Schnittstelle ausgestattet werden, um noch eine akzeptable Performance zu erzielen.

Die Intension der Webservices deckt sich hier mit der Intension der »Coarse-Grained« Komponenten. Die Komponenten werden mit nur wenigen Methoden, mit vergleichsweise viel Funktionalität ausgestattet – die Webservices rufen diese Methoden dann auf. Da eine »grobe-Komponente« (*coarse* = grob) über mächtige Funktionalität oder Geschäftslogik verfügen sollte, wirkt sich der Kommunikationsaufwand nicht mehr so negativ auf die Gesamt-Performance der Anwendung aus.

3.5 Wie kann der Client mitteilen, dass die gerade benutzte EntityBean nicht mehr benötigt wird?

3.5.1 Problemstellung

Falls ein Client eine SessionBean nicht mehr benötigt, kann er jederzeit die Methode `remove` des Remote-Interfaces aufrufen. Diese Methode wird durch das EJBObject Interface vorgegeben, sodass sich der Client auf ihre Existenz verlassen kann.

```
public interface EJBObject extends Remote{
public EJBHome   getEJBHome()
public Handle   getHandle()
public Object   getPrimaryKey()
public boolean  isIdentical(EJBObject obj)
public void  remove()
}
```

Sowohl die Remote-Interfaces (bzw. die Local-Interfaces) der Session-, als auch der EntityBeans müssen das `EJBObject` bzw. das Interface `EJBLocalObject` implementieren. Somit steht die Methode dem Client zur Verfügung. Bei dem Aufruf dieser Methode bei einer SFSB wird die gerade von dem Client »gehaltene« Instanz sofort zerstört. Der Client kann ab diesem Zeitpunkt nicht mehr mit dem Remote- bzw. Local-Interface arbeiten.

Da die SLSBs zustandslos sind, sind alle gerade gepoolte Instanzen identisch. Ein `remove`-Aufruf des Clients signalisiert lediglich, dass die gerade referenzierte Instanz von dem Client nicht mehr benötigt wird. Der Applikationsserver zerstört diese jedoch nicht, da diese sofort die Anfragen anderer Clients übernehmen kann. Der Client sollte im Fall von SessionBeans immer die Methode `remove` aufrufen, falls diese Instanz nicht mehr benötigt wird. Der EJB-Container zerstört die SFSB, da diese sich nicht wieder verwenden lassen, allerdings nimmt nur `remove`-Aufrufe der SLSB »zur Kenntnis«. Es kann jedoch passieren, dass die `remove`-Aufrufe nicht ganz folgenfrei bleiben. Bei zu vielen gerade aktiven Instanzen kann der EJB-Container die Pool-Größe verkleinern, um so die Ressourcen freizugeben.

Der `remove`-Aufruf einer EntityBean hat jedoch eine vollkommen andere Wirkung. Der Client signalisiert hier nicht, dass er mit der Instanz nicht mehr arbeiten möchte. Bei diesem Aufruf werden die Daten der Bean aus der Datenbank entfernt. Die Bean-Instanz kehrt anschließend in den Pool zurück. Ab diesem Zeipunkt befindet sich diese Instanz in einem nicht definierten Zustand und muss vor ihrer Verwendung wieder »geladen« werden. Dies geschieht mit der Assoziation mit dem Primärschlüssel und dem Aufruf der `ejbLoad` Methode der Bean.

Da die Methode `remove` bei einer EntityBean andere Bedeutung hat, wie kann der Client dem Applikationsserver mitteilen, dass er die Instanz nicht mehr benötigt?

3.5.2 Technischer Hintergrund

Die EntityBeans sind nicht für die Abbildung der Geschäftslogik zuständig. Vielmehr bilden diese die Entitäten oder das Modell der Anwendung ab. Dabei sind sie persistent, d.h. der Zustand bleibt auch nach dem Neustart des Applikationsservers erhalten. Allerdings besitzt eine EntityBean kein »Konversationsgedächtnis«. Eine Instanz kann von vielen Clients gleichzeitig benutzt werden. Abhängig von dem eingestellten Transaktionslevel wird der Zustand der Bean-Instanzen mit dem Zustand der Datenbank synchronisiert. Aus der Sicht des Clients handelt es sich hier um einen »globalen« Zustand, der bei jeder Änderung auch von den anderen Clients »gesehen« werden kann. Aus diesem Grund kann man die EntityBeans mit den SLSB vergleichen, da beide nicht wissen, mit welchem Client gerade kommuniziert wird. Wenn man das Konversationsgedächtnis betrachtet, sind sowohl die SLSBs als auch die EntityBeans zustandslos.

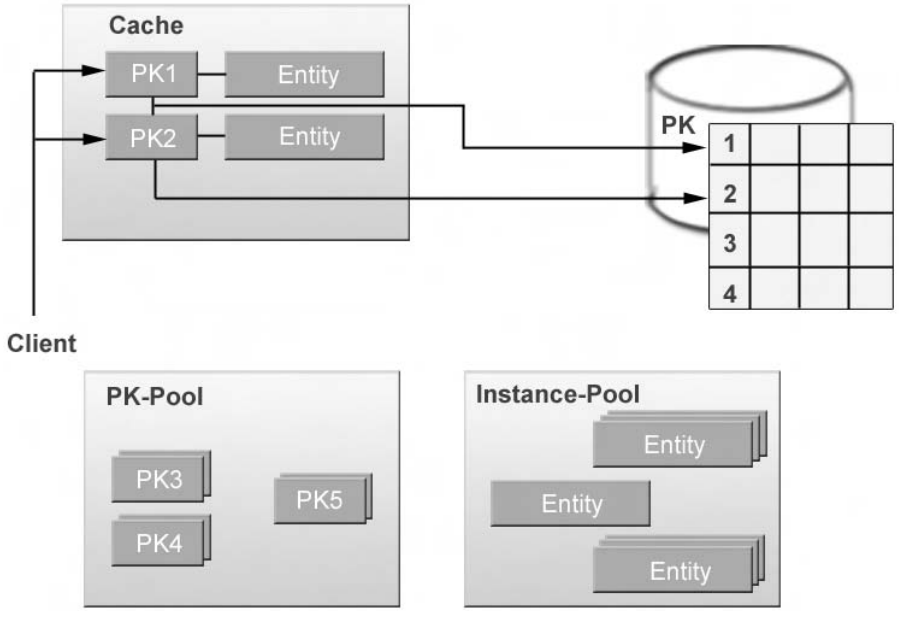


Abbildung 3.27: EntityBean – ein Cache der Datenbank

Allerdings besitzen die EntityBean nach jeder Interaktion mit dem Client einen gültigen, persistenten Zustand. Bei diesem Zustand handelt es sich um einen gültigen Cache der Datenbank. Ähnlich wie bei den SessionBeans lohnt es sich hier nicht, diesen bei jedem Client-Disconnect zu zerstören, da die Daten ohne einen erneuten Datenbankzugriff wieder verwendet werden können. Mit den in der EJB-Spezifikation beschriebenen »Commit Options«, lässt sich dieses Verhalten feingranulär bestimmen. Es können beispielsweise die EJB-Zustände auch über die Transaktionsgrenzen hinweg gecached werden.

3.5.3 Lösung

Da es sich bei den EntityBean Instanzen um eine Abbildung der Datensätze in einer Tabelle handelt, ist der Zustand einer EntityBean-Instanz client-unabhängig. Vielmehr kann man hier von einem globalen Zustand sprechen, der von vielen Clients gleichzeitig »gesehen« wird. Es ist also nicht sinnvoll, dass ein einzelner Client den Lebenszyklus eine Instanz (und somit eines Datensatzes) bestimmt.

Dem EJB-Container steht jedoch jederzeit frei, eine Bean-Instanz aus dem Cache zu entfernen. Wenn sich kein Client mehr für eine EntityBean mit einem bestimmten Schlüssel interessiert, kann der EJB-Container diese aus dem Cache entfernen, um Platz für andere, öfters benutzte Instanzen zu schaffen. Diese Vorgehensweise wird auch von Betriebssystemen benutzt, um nicht mehr so oft benötigte Speicherbereiche

aus dem Haupspeicher zu entfernen und in den Sekundärspeicher abzulegen. Sowohl das Betriebssystem also auch der EJB-Container können hier auch unterschiedlichen Auslagerungstrategien wie LRU (Least Recently Used) oder MRU (Most Recently Used) anwenden.

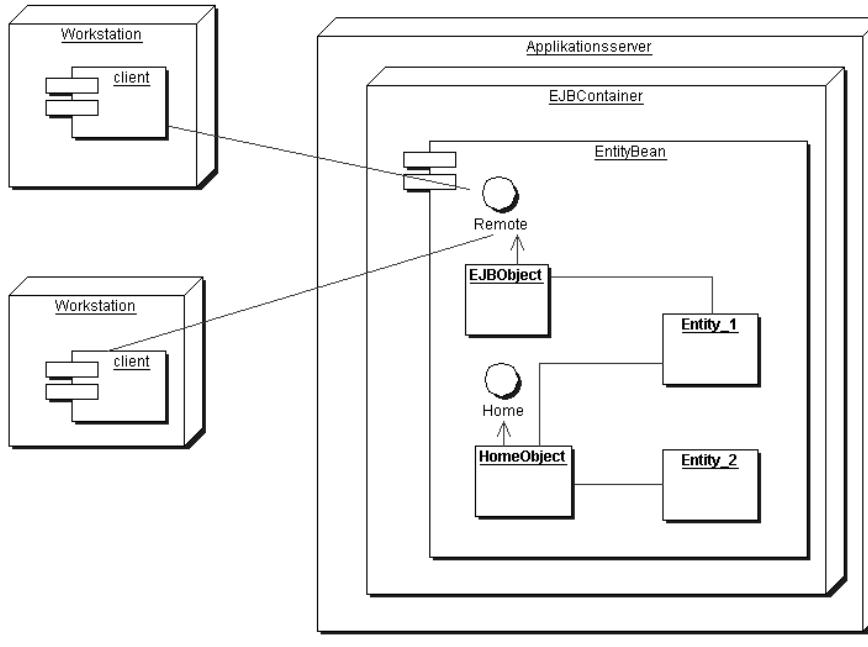


Abbildung 3.28: EntityBean als »shared resource«

Die Funktionsweise eines Caches wurde nicht in der EJB-Spezifikation festgelegt. J2EE-kompatible Applikationsserver, können die Caching-Strategien völlig unterschiedlich implementieren. Für die Performance eine J2EE-Anwendung ist das Caching von gültigen Instanzen jedoch sehr wichtig. Der Zugriff auf eine bereits »geladene« Instanz kann um ein Vielfaches performanter sein als das Laden einer Bean aus der Datenbank.

3.6 Ab wann lohnt es sich, die EJB-Technologie in Projekten einzusetzen?

Diese Frage wird mir oft in meinen Kursen und Beratungen gestellt. Viele assozieren mit dem Einsatz von EJBs hohe Kosten und einen nicht zu unterschätzenden Lernaufwand der relativ neuen Technologie. Die Befürchtung, dass sich mit EJBs keine performanten Anwendungen entwickeln lassen, ist auch sehr verbreitet.

3.6.1 Problemstellung

Noch vor wenigen Jahren konnte man die Frage folgendermaßen beantworten: Möchte man hochskalierbare Anwendungen bauen, werden EJBs benötigt. Diese Antwort hört sich natürlich sehr gut an, allerdings ist man sich zuerst ein wenig unsicher, was eigentlich »Skalierbarkeit« bedeutet.

3.6.2 Technischer Hintergrund

Obwohl die Skalierbarkeit mit der Performance einer Anwendung in Verbindung gebracht wird, hat sie eigentlich mit dieser Eigenschaft nichts zu tun. Vielmehr handelt es sich um die Anforderung, bei wachsender Last weiterhin eine akzeptable Performance des Dienstes zu erzielen. Die Skalierbarkeit lässt sich mit dem Einsatz einer geclusterten Umgebung auf jeden Fall erhöhen. Die Last kann auf die einzelnen Knoten verteilt werden. Dies sieht zunächst nach Verbesserung der Performance aus. Dies trifft nur zu, wenn der Cluster zustandslos ist, was wiederum in der Praxis nur selten vorkommt. Die meisten Anwendungen sind leider zustandsbehaftet, was den Cluster zwingt, seinen Zustand auf alle Knoten zu replizieren. Bei schlecht konfigurierten Applikationsservern kann der Zustand auch in eine gemeinsame Datenbank abgelegt werden, was die Performance um Faktoren verschlechtert.

Obwohl man mit dem Begriff »Skalierbarkeit« den Kunden auf jeden Fall beeindrucken kann, ist diese bestimmt nicht das einzige Kriterium für die Wahl der EJB-Technologie. Schauen wir uns mal die EJB-Technologie genauer an.

Bei den EJBs handelt es sich um Komponenten, die innerhalb eines EJB-Containers ablaufen. Der EJB-Container überwacht jede EJB-Instanz und sorgt dafür, dass es ihr gut geht. Ferner überwacht er auch, dass nicht eine Instanz »arbeitslos« wird. Eine Stateful SessionBean ist für den Container nicht so wertvoll, da diese nur von einem Client benutzt werden kann. Andere Clients dürfen sich diese Instanz nicht teilen, was natürlich auch die »Poolbarkeit« unmöglich macht. Sie wird zuerst passiviert, also aus den Speicher entfernt. Erst eine langzeitarbeitslose Komponente wird zerstört.

Anders sieht hier bei den zustandslosen Komponenten aus. Diese können beliebig von den verschiedenen Clients »wieder verwendet« werden. Der EJB-Container wird mit der Zerstörung dieser Komponenten zögern. Erst beim Herunterfahren des Applikationsservers oder der Reduzierung der Poolgröße, werden die betroffenen zustandslosen Komponenten zerstört.

Neben der effizienten Instanzenverwaltung übernimmt der Applikationsserver auch andere Dienste. Diese lassen sich durch die entsprechenden Einträge im Deployment Deskriptor einstellen.

3.6.3 Antwort

Die meisten J2EE-Anwendungen benötigen Transaktionen, um ihre Daten konsistent zu halten. Bei den EJBs handelt es sich standardmäßig um transaktionelle Ressourcen. Das Transaktionsverhalten kann entweder deklarativ oder programmatisch bestimmt werden. Bei der deklarativen Variante spricht man von CMT (Container Managed Transactions), bei der programmatischen von BMT (Bean Managed Transactions). Meistens wird die CMT-Variante verwendet, da dadurch die Transaktionen bequem im Deployment Deskriptor eingestellt werden können. Bei der Verwendung von EJBs wird also der Umgang mit Transaktionen erheblich erleichtert.

Ein weiterer Vorteil der EJBs ist ihre »Remotefähigkeit«. Auf eine EJB kann ohne Aufwand auch außerhalb des Applikationsservers zugegriffen werden. Die nötige Infrastruktur (Stubs und Skeletons) werden beim Deployment automatisch erzeugt.

Die Container Managed Persistence (CMP) der EntityBeans wurde in der EJB 2.0 Spezifikation modifiziert und erweitert. Dabei lassen sich auch Beziehungen zwischen den Entitäten vom Container verwalten. Die Konsistenz der Container Managed Relations- (CMR) Felder wird von dem Container automatisch überwacht. Auch 1:1, 1:N, N:M, N:1 Beziehungen lassen sich mit der EJB 2.0 CMP ohne größeren Programmieraufwand abbilden.

Dem Entwickler wird also eine Menge an Funktionalität »geschenkt«. Die nötige Infrastruktur wird einfach von dem Applikationsserver bereitgestellt. Der Entwickler kann sich voll auf die Implementierung der Geschäftslogik konzentrieren.

Bei der Entwicklung von server-seitigen Anwendungen stehen dem Entwickler die folgenden Möglichkeiten zur Verfügung:

1. die EJBs einzusetzen und die Infrastruktur des Applikationsservers nutzen und
2. ein eigenes Komponentenmodell zu erstellen – hier muss aber auch die bereits bestehende Infrastruktur nachgebaut werden.

Da es bereits viele opensource Applikationsserver auf dem Markt gibt, spielen die Kosten keine so große Rolle mehr. Zumindest die Entwicklungsphase kann mit »freien« Produkten erfolgen. Die J2EE-Spezifikation stellt hier sicher, dass man nur mit geringem Aufwand bei der Migration auf einen anderen J2EE-konformen Applikationsserver rechnen muss.

Die Frage die man sich stellen sollte, müsste allerdings anders lauten: Möchte ich objektorientierte Komponenten entwickeln oder nicht?

Wenn die Frage mit »ja« beantwortet werden kann, ist auch eine spätere Migration auf andere Technologien (JDO, CORBA) möglich. Die für den Kunden wichtige Geschäftslogik muss nicht in den SessionBeans liegen. Diese lässt sich auch von »normalen« Klassen, sog. Helper-Klassen implementieren. Die EJBs können die Geschäftslogik benutzen und mit zusätzlicher Funktionalität »dekorieren«.

Die Präsentationsschicht sollte auch nicht direkt mit den EJBs kommunizieren. Auch hier ist eine Abstraktion durch eine Adapterschicht (Business Delegate Pattern) möglich. Somit bleibt die EJB-Technologie austauschbar. Bei akuten Performance-Problemen könnte man ggf. diese Schicht durch eigene Implementierung austauschen. Falls beim Profiling der Anwendung Bottlenecks identifiziert werden können, lassen sich diese bequem austauschen – unabhängig davon, ob es sich hier um eine EJB handelt ohne nicht.

3.7 Ist BMP schneller als die CMP 2.0 Persistenz?

Diese Frage wird mir meistens bei meinen Beratungen und Schulungen gestellt, wenn wir die CMP 2.0 Persistenz besprechen. Die Datenbankentwickler, die bereits über fundierte Erfahrung in der SQL-Programmierung verfügen, wählen meistens die BMP-Persistenz. Man glaubt, dass der eigene, optimierte SQL-Code immer performanter sein muss als der generierte und somit sicherlich auch nur »suboptimaler« SQL-Code des EJB-Containers. Es werden oft Vorgaben für Projekte geschrieben, in denen BMP aus Performance-Gründen vorgeschrieben wird. Wie sieht es denn aber mit der Performance der beiden Beans in der Praxis aus? Lohnt sich der höhere Aufwand bei der Entwicklung der BMP-Beans noch? Für die Betrachtung der Performance der beiden Persistenzen wird lediglich die Bean-Klasse ausgetauscht. Die `LocalHome` und Home-Interfaces bleiben identisch.

3.7.1 Problemstellung

Um die Performance der beiden Persistenzen miteinander zu vergleichen, schauen wir uns mal die beiden Implementierungen der Testbeans an.

Die CMP 2.0-Persistenz

Zuerst konzentrieren wir uns mal auf die CMP 2.0 Persistenz Es handelt sich hier um eine EntityBean aus einer bereits bestehenden *http://www.star-finder.com* Anwendung. Diese Anwendung wurde von mir für mein Buch »J2EE Patterns« geschrieben und eignet sich optimal für die Performance-Messungen. Es handelt sich dabei um eine Sternensuchmaschine. Die Sterne werden von einer bereits existierenden Datei in die Datenbank geladen. Für die Messungen wurden in die Datenbank 325.000 Sterne aus dem »Bonner Durchmusterungskatalog« (`ftp://adc.gsfc.nasa.gov/pub/adc/archives/catalogs/1/1122/`) geladen. Die StarBean nimmt die Sterne entgegen und lädt diese in eine JDBC-fähige Datenbank.

Die StarBean wird mit den »Local«-Interfaces ausgestattet. Somit ist es unmöglich, die Entity von »außen« zu erreichen – die `StarBean`-Entity kann lediglich von der gleichen JVM aus, »per Referenz« angesprochen werden. Das Interface `StarLocal` gibt lediglich

das Value Object `StarVO` zurück. Es handelt sich dabei um einen einfachen Datencontainer, der für einen effizienten Datentransport zuständig ist.

```
public interface StarLocal extends EJBLocalObject {
       public StarVO getStarVO();
}
```

Das Interface `StarLocalHome` besteht, wie erwartet, aus einer `create-`Methode und mehreren Findern. Bei einem Local-Home Interface handelt es sich um eine »LocalFactory« – es können hier lediglich die Local–Interfaces zurückgegeben werden.

```
public interface StarLocalHome extends EJBLocalHome{
      public StarLocal  create(String catalogPrefix, String zoneSign, int
declinationZone, int numberWithinZone, String note,float magnitude,int
rightAscensionHours,int rightAscensionMinutes,float rightAscentionSeconds,String
signOfDeclination,int declinationDegrees,float declinationMinutes) throws
CreateException;
 public StarLocal  findByPrimaryKey(StarPK pk) throws FinderException;
 public Collection findAll() throws FinderException;
 public Collection findByZone(int zone) throws FinderException;
 public Collection findByMagnitude(float magnitude) throws FinderException;
 public Collection findByMagnitudeLessThen(float magnitude) throws
FinderException;
 public Collection findByMagnitudeGreaterThen(float magnitude) throws
FinderException;
 public Collection findByMagnitudeInRange(float magnitudeFrom,float magnitudeTO)
throws FinderException;
public Collection findByDeclination(int degrees,float minutes) throws
FinderException;
 public Collection findByDeclinationInRange(int degreesFrom,int degreesTo) throws
FinderException;
}
```

Beide Interfaces gelten sowohl für die CMP 2.0 als auch für die BMP-Variante der Bean. Die Implementierung der Bean und die Wahl der Persistenzstrategie kann also erst beim Deployen festgelegt werden.

Die CMP 2.0 Entity ist relativ einfach. Jede Spalte einer Tabelle, oder jedes Attribut der Bean wird durch abstrakte »Getter« und »Setter« repräsentiert. Alle anderen Callback-Methoden des EJB-Containers sind zwar nicht abstrakt, bleiben aber leer.

```
public abstract class StarBean implements EntityBean {
    private EntityContext entityContext = null;
public StarPK ejbCreate(String catalogPrefix, String zoneSign, int
declinationZone, int numberWithinZone, String note,float magnitude,int
rightAscensionHours,int rightAscensionMinutes,float rightAscentionSeconds,String
signOfDeclination,int declinationDegrees,float declinationMinutes)throws
CreateException{
        this.setCatalogPrefix(catalogPrefix);
        this.setZoneSign(zoneSign);
```

```
        this.setDeclinationZone(declinationZone);
        this.setNumberWithinZone(numberWithinZone);
        this.setNote(note);
        this.setMagnitude(magnitude);
        this.setRightAscensionHours(rightAscensionHours);
        this.setRightAscensionMinutes(rightAscensionMinutes);
        this.setRightAscentionSeconds(rightAscentionSeconds);
        this.setSignOfDeclination(signOfDeclination);
        this.setDeclinationDegrees(declinationDegrees);
        this.setDeclinationMinutes(declinationMinutes);
        return null;
    }
    public void ejbPostCreate(String catalogPrefix, String zoneSign, int
declinationZone, int numberWithinZone, String note,float magnitude,int
rightAscensionHours,int rightAscensionMinutes,float rightAscentionSeconds,String
signOfDeclination,int declinationDegrees,float declinationMinutes) throws
CreateException{    }

    public void setEntityContext(EntityContext entityContext) {
        this.entityContext = entityContext;
    }

    public void ejbStore() { }
    public void ejbActivate(){    }
    public void ejbPassivate(){  }
    public void ejbLoad(){ }

    public abstract String getCatalogPrefix();
    public abstract void   setCatalogPrefix(String catalogPrefix);

    public abstract String getZoneSign();
    public abstract void   setZoneSign(String zoneSign);

    public abstract int getDeclinationZone();
    public abstract void   setDeclinationZone(int declinationZone);

    public abstract int getNumberWithinZone();
    public abstract void   setNumberWithinZone(int numberWithin);

    public abstract String getNote();
    public abstract void   setNote(String note);

    public abstract float getMagnitude();
    public abstract void   setMagnitude(float magnitude);

    public abstract int getRightAscensionHours();
    public abstract void   setRightAscensionHours(int rightAscensionHours);

    public abstract int getRightAscensionMinutes();
    public abstract void   setRightAscensionMinutes(int rightAscensionMinutes);
```

```
    public abstract float getRightAscentionSeconds();
    public abstract void   setRightAscentionSeconds(float rightAscentionSeconds);

    public abstract String getSignOfDeclination();
    public abstract void   setSignOfDeclination(String signOfDeclination);

    public abstract int getDeclinationDegrees();
    public abstract void   setDeclinationDegrees(int declinationDegrees);

    public abstract float getDeclinationMinutes();
    public abstract void setDeclinationMinutes(float declinationMinutes);

    public void ejbRemove()  {   }
    public void unsetEntityContext() {   }

    public StarVO getStarVO(){
        return new
StarVO(this.getCatalogPrefix(),this.getZoneSign(),this.getDeclinationZone(),this.
getNumberWithinZone(),this.getNote(),this.getMagnitude(),this.getRightAscension
Hours(),this.getRightAscensionMinutes(),this.getRightAscentionSeconds(),this.getSi
gnOfDeclination(),this.getDeclinationDegrees(),this.getDeclinationMinutes());
    }
   }
```

Eine Besonderheit dieser Bean ist ihr Composite-Key. Jeder einzelne Stern wird durch die folgenden fünf Attribute: `catalogPrefix`, `zoneSign`, `declinationZone`, `numberWithinZone` und `note` identifiziert. Ein zusammengesetzter Schlüssel muss mit einer eigenständigen Klasse implementiert werden. Diese Klasse muss entweder das `java.io.Serializable` oder `java.rmi.Remote` Interface implementieren. Wichtig ist auch die richtige Implementierung der Methoden `equals` und `hashCode`.

```
public class StarPK implements Serializable{
    public String catalogPrefix          = null;
    public String zoneSign         = null;
    public int    declinationZone  = -1;
    public int    numberWithinZone = -1;
    public String note             = null;

    public StarPK(){    }
    public StarPK(String catalogPrefix,String zoneSign,int declinationZone, int
numberWithinZone,String note){
        this.catalogPrefix = catalogPrefix;
        this.zoneSign      = zoneSign;
        this.declinationZone = declinationZone;
        this.numberWithinZone = numberWithinZone;
        this.note             = note;
    }
    public String getCatalogPrefix(){ return this.catalogPrefix; }
    public String getZoneSign(){ return this.zoneSign; }
    public int    getNumberWithinZone(){ return this.numberWithinZone; }
```

```
    public int    getDeclinationZone(){ return this.declinationZone; }
    public String getNote(){ return this.note; }

    public boolean equals(Object object){
        StarPK temp;
        if(!(object instanceof StarPK))
            return false;
        temp = (StarPK)object;

        if(this.getCatalogPrefix().equalsIgnoreCase(temp.getCatalogPrefix()) &&
this.zoneSign.equals(temp.getZoneSign()) && this.declinationZone ==
temp.getDeclinationZone() && this.numberWithinZone == temp.getNumberWithinZone()
&& this.note.equalsIgnoreCase(temp.getNote()) )
            return true;
        else
            return false;
   }
 public int hashCode(){
     return
this.catalogPrefix.concat(this.zoneSign).concat(String.valueOf(this.declinationZon
e)).concat(String.valueOf(this.numberWithinZone)).concat(this.note).hashCode();
    }
 public String toString(){
   return "CP: "+ getCatalogPrefix() + " ZS: " + getZoneSign() + " DZ: " +
getDeclinationZone() + " NWZ: "+ getNumberWithinZone() + " Note: " + getNote();
    }
}
```

Die BMP-Persistenz

Für die Implementierung der BMP-Persistenz wurde eine neue Implementierung der `StarBean` erstellt. Diese Klasse ist nicht mehr abstrakt. Alle abstrakten Methoden werden hier ausimplementiert. Neben den bereits bekannten Methoden müssen noch alle in den Interfaces Home und LocalHome deklarierten »finder« mit dem Prefix »`ejb`« implementiert werden. Hier findet der Zugriff auf die Datenbank und die Suche nach den Primärschlüsseln der Bean statt. Der Entwickler muss ferner auch für Synchronisierung des Zustandes der Bean mit der Datenbank sorgen. In der Praxis wird der SQL-Code aus der Bean ausgelagert, um die Austauschbarkeit der Persistenzschicht zu gewährleisten. Beim Aufruf der »Getter« und »Setter« dieser Bean wird zunächst das `StarVO` modifiziert. Dieses Value-Object kann als interner Cache der Bean angesehen werden.

```
public class StarBeanBMP implements EntityBean {
    private EntityContext entityContext      = null;
    private StarVO        currentData        = null;
    private DAOFactoryIF  daoFactory         = null;
    private BaseDAO       dao                = null;
    private ServiceLocatorEJB serviceLocator = null;
    private final static String MAX_NUMBER_OF_RECORDS = "MAX_NUMBER_OF_RECORDS";
    private final static String DAO_CLASS     = "DAO_CLASS";
```

```
    private final static String DATA_SOURCE   = "DATA_SOURCE";
    private String        daoClass                = null;
    private String        dataSource              = null;
    private int           maxNumberOfConnections  = -1;
    public StarPK ejbCreate(String catalogPrefix, String zoneSign, int
declinationZone, int numberWithinZone, String note,float magnitude,int
rightAscensionHours,int rightAscensionMinutes,float rightAscentionSeconds,String
signOfDeclination,int declinationDegrees,float declinationMinutes)throws
CreateException{
        currentData.setCatalogPrefix(catalogPrefix);
        currentData.setZoneSign(zoneSign);
        currentData.setDeclinationZone(declinationZone);
        currentData.setNumberWithinZone(numberWithinZone);
        currentData.setNote(note);
        currentData.setMagnitude(magnitude);
        currentData.setRightAscensionHours(rightAscensionHours);
        currentData.setRightAscensionMinutes(rightAscensionMinutes);
        currentData.setRightAscentionSeconds(rightAscentionSeconds);
        currentData.setSignOfDeclination(signOfDeclination);
        currentData.setDeclinationDegrees(declinationDegrees);
        currentData.setDeclinationMinutes(declinationMinutes);
        return dao.create(currentData);
    }

    public Collection ejbFindAll(){
       return this.dao.findAll();
    }
    public Collection ejbFindByZone(int zone){
       return this.dao.findByZone(zone);
    }
    public Collection ejbFindByMagnitude(float magnitude){
       return this.dao.findByMagnitude(magnitude);
    }
    public Collection ejbFindByMagnitudeLessThen(float magnitude){
       return this.dao.findByMagnitudeLessThen(magnitude);
    }
    public Collection ejbFindByMagnitudeGreaterThen(float magnitude){
       return this.dao.findByMagnitudeGreaterThen(magnitude);
    }
    public Collection ejbFindByMagnitudeInRange(float magnitudeFrom,float
magnitudeTO){
       return this.dao.findByMagnitudeInRange(magnitudeFrom,magnitudeTO);
    }
    public Collection ejbFindByDeclination(int degrees,float minutes){
       return this.dao.findByDeclination(degrees,minutes);
    }
    public Collection ejbFindByDeclinationInRange(int degreesFrom,int degreesTo){
       return this.dao.findByDeclinationInRange(degreesFrom,degreesTo);
    }
    public StarPK ejbFindByPrimaryKey(StarPK starPK){
       return this.dao.findByPrimaryKey(starPK);
    }
```

```
    public void ejbPostCreate(String catalogPrefix, String zoneSign, int
declinationZone, int numberWithinZone, String note,float magnitude,int
rightAscensionHours,int rightAscensionMinutes,float rightAscentionSeconds,String
signOfDeclination,int declinationDegrees,float declinationMinutes) throws
CreateException{    }

    public void setEntityContext(EntityContext entityContext) {
        this.entityContext = entityContext;
        try{
            this.init();
            this.daoFactory = new
DAOFactory(this.dataSource,this.daoClass,this.maxNumberOfConnections);
            this.dao    = this.daoFactory.getDAO();
        }catch(Exception e){
            throw new EJBException("[BookBean.setEntityContext] Exception:
" + e.toString());
        }

    }
     private void init(){
        try{
            this.serviceLocator = new ServiceLocatorEJB();
            Context env = this.serviceLocator.getEnvironmentContext();
            this.daoClass = (String)env.lookup(DAO_CLASS);
            this.dataSource = (String)env.lookup(DATA_SOURCE);
            this.maxNumberOfConnections =
((Integer)env.lookup(MAX_NUMBER_OF_RECORDS)).intValue();
        }catch(Exception e){
            throw new EJBException("[StarBeanBMP.init()] Problem initializing.
Reason: " + e.toString());
        }
    }
    public void ejbStore() { this.dao.store(this.currentData); }
    public void ejbActivate(){   this.dao.activate(); }
    public void ejbPassivate(){ this.dao.passivate(); }
    public void ejbLoad(){
       this.currentData = this.dao.loadData(getCurrentPK());
    }
    public  String getCatalogPrefix(){
       return this.currentData.getCatalogPrefix();
    }
    public void  setCatalogPrefix(String catalogPrefix){
         this.currentData.setCatalogPrefix(catalogPrefix);
    }
    public String getZoneSign(){
         return this.currentData.getZoneSign();
    }
    public void   setZoneSign(String zoneSign){
       this.currentData.setZoneSign(zoneSign);
    }
    public int getDeclinationZone()
       return this.currentData.getDeclinationZone();
```

```
    }
    public void   setDeclinationZone(int declinationZone){
       this.currentData.setDeclinationZone(declinationZone);
    }
 //die restlichen "Getter" und "Setter" wurden weggelassen…
    public void ejbRemove(){
       this.dao.delete(getCurrentPK());
    }
    private StarPK getCurrentPK(){
       return (StarPK)this.entityContext.getPrimaryKey();
    }
    public void unsetEntityContext() {   }
    public StarVO getStarVO(){
        return this.currentData;
    }
}
```

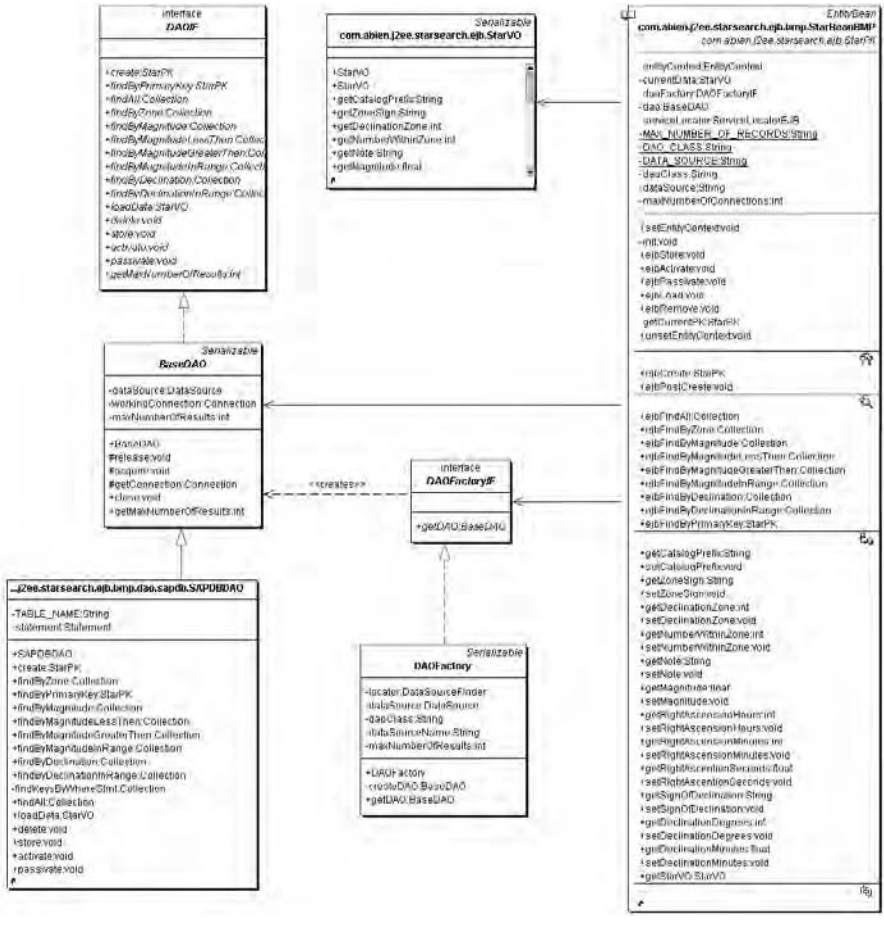


Abbildung 3.29: Die BMP-Variante der StarBean

Erst beim Aufruf der Methode `ejbStore` wird der Zustand der Bean an die aktuelle DAO-Instanz übergeben. Diese kümmert sich um die Speicherung der Daten. Das DAO entspricht der Schnittstelle der Bean in die Integrationsschicht. Es ist wichtig, dass diese Schnittstelle sauber entworfen wird, damit die Koppelung der beiden Schichten möglichst gering ausfällt. In unserem Fall wird das DAO durch eine abstrakte Klasse `com.abien.j2ee.starsearch.ejb.bmp.dao.BaseDAO` repräsentiert. Um die Koppelung der beiden Schichten zu minimieren, kommt zusätzlich eine Factory zum Einsatz. Die Bean benutzt die Factory, um eine Instanz des DAOs zu erhalten.

Die Klasse `com.abien.j2ee.starsearch.ejb.bmp.dao.sapdb.SAPDBDAO` erbt von der Klasse `BaseDAO` und weiß wie mit der SAP-Datenbank (*www.sapdb.org*) zu kommunizieren ist. Diese DAO-Implementierung wurde speziell für die SAP-Datenbank geschrieben und ist somit auch von dieser »abhängig«. Allerdings kennt die `StarBeanBMP` nur die abstrakte und nicht die konkrete `SAPDBDAO` Implementierung. Somit bleibt die Entkoppelung der Geschäftslogik- von der Integrationsschicht gewährleistet.

```
public class SAPDBDAO extends BaseDAO {
       private final String TABLE_NAME = "STARBEAN";
       private Statement statement = null;
       public SAPDBDAO(DataSource dataSource, int maxNumberOfResults) {
           super(dataSource, maxNumberOfResults);
       }
 public StarPK create(StarVO starVO) throws DAOException {
      StarPK starPK = null;
      String catalogPrefix = starVO.getCatalogPrefix();
      String zoneSign = starVO.getZoneSign();
      int declinationZone = starVO.getDeclinationZone();
      int numberWithinZone = starVO.getNumberWithinZone();
      String note = starVO.getNote();
      float magnitude = starVO.getMagnitude();
      int rightAscensionHours = starVO.getRightAscensionHours();
      int rightAscensionMinutes = starVO.getRightAscensionMinutes();
      float rightAscentionSeconds = starVO.getRightAscentionSeconds();
      String signOfDeclination = starVO.getSignOfDeclination();
      int declinationDegrees = starVO.getDeclinationDegrees();
      float declinationMinutes = starVO.getDeclinationMinutes();
      try {
      StringBuffer sql = new StringBuffer();
       this.acquire();
       sql.append("INSERT INTO ").append(TABLE_NAME).append(
       "( catalogPrefix,  zoneSign, declinationZone,  numberWithinZone,
note,magnitude, rightAscensionHours, rightAscensionMinutes,
rightAscentionSeconds,signOfDeclination,declinationDegrees,declinationMinutes) ");
      sql.append("VALUES (\'").append(catalogPrefix).append("\'")
      .append(", \'").append(zoneSign).append("\'").append(", \'")
      .append(" , ").append(declinationZone).append(" , ")
      .append(numberWithinZone).append(" , ").append("\'")
      .append(note).append("\'").append(" , ").append(magnitude
```

```
        .append(" , ").append(rightAscensionHours).append(" , ")
       .append(rightAscensionMinutes).append(" , ")
       .append(rightAscentionSeconds).append(" , ").append("\'")
       .append(signOfDeclination).append("\'").append(" , ")
.append(declinationDegrees).append(" , ")
       .append(declinationMinutes)      .append(")");
       statement = getConnection().createStatement();
       statement.executeUpdate(sql.toString());
       starPK =new StarPK(catalogPrefix, zoneSign,
       declinationZone,numberWithinZone, note);
       this.statement.close();
       } catch (Exception e) {
       throw new DAOException("[SAPDBDAO.create] Exception:"+ e);
       } finally {
       try {
              this.release();
       } catch (Exception e) {}
       }
       return starPK;
}
public Collection findByZone(int zone) throws DAOException {
 return this.findKeysByWhereStmt(" tO_o.declinationZone = " + zone);
}
public StarPK findByPrimaryKey(StarPK starPK) throws DAOException {
 int declinationZone = -1;
 int numberWithinZone = -1;
 String catalogPrefix = null;
 String note = null;
 String zoneSign = null;
   try {
    StringBuffer sql = new StringBuffer();
    this.acquire();
    sql.append("SELECT tO_o.declinationZone, tO_o.numberWithinZone,
tO_o.catalogPrefix, tO_o.note, tO_o.zoneSign FROM ")
    .append(TABLE_NAME).append(" tO_o").append(" WHERE declinationZone = "+
starPK.getDeclinationZone()+ " AND numberWithinZone = "             +
starPK.getNumberWithinZone()+ " AND catalogPrefix = \'"
+ starPK.getCatalogPrefix()      + "\' AND note = \'
+ starPK.getNote()+ "\' AND zoneSign =  \'"
+ starPK.getZoneSign()+"\'");
 statement = getConnection().createStatement();
 ResultSet result = statement.executeQuery(sql.toString());
 while (result.next()) {
       declinationZone = result.getInt(1);
       numberWithinZone = result.getInt(2);
       catalogPrefix = result.getString(3);
       note = result.getString(4);
       zoneSign = result.getString(5);
       starPK = new StarPK( catalogPrefix,zoneSign,
       declinationZone,numberWithinZone, note);
}
```

```
      this.statement.close();
       } catch (Exception e) {
       throw new DAOException(
       "[SAPDBDAO.findByPrimaryKey] Exception occured: " + e);
    } finally {
       try {
       this.release();
       } catch (Exception e) {}
         }
       return starPK;
       }
 public Collection findByMagnitude(float magnitude) throws DAOException{
 return this.findKeysByWhereStmt(" tO_o.magnitude = " + magnitude);
}
public Collection findByMagnitudeLessThen(float magnitude) throws DAOException {
return this.findKeysByWhereStmt(" tO_o.magnitude < " + magnitude);
}
public Collection findByMagnitudeGreaterThen(float magnitude)
 throws DAOException {
 return this.findKeysByWhereStmt(" tO_o.magnitude > " + magnitude);
}
public Collection findByMagnitudeInRange(float magnitudeFrom,
      float magnitudeTO)throws DAOException {
return this.findKeysByWhereStmt(" tO_o.magnitude BETWEEN " + magnitudeFrom + " AND
" + magnitudeTO);
}
public Collection findByDeclination(int degrees, float minutes)throws DAOException
{
return this.findKeysByWhereStmt(" tO_o.declinationDegrees = "
+ degrees+ " AND tO_o.declinationMinutes = "+ minutes);
}
public Collection findByDeclinationInRange(int degreesFrom, int degreesTo)throws
DAOException {
return this.findKeysByWhereStmt(" tO_o.declinationDegrees BETWEEN "
      + degreesFrom+ " AND "      + degreesTo);
}
private Collection findKeysByWhereStmt(String whereStatement)
            throws DAOException {
 ArrayList keyList = new ArrayList();
 int declinationZone = -1;
 int numberWithinZone = -1;
 String catalogPrefix = null;
 String note = null;
 String zoneSign = null;
 int counter = 0;
 try {
  StringBuffer sql = new StringBuffer();
  this.acquire();
  sql.append("SELECT tO_o.declinationZone, tO_o.numberWithinZone,
tO_o.catalogPrefix, tO_o.note, tO_o.zoneSign FROM ").append(TABLE_NAME)
```

```
.append(" t0_o").append(" WHERE ").append(whereStatement);
statement = getConnection().createStatement();
 ResultSet result = statement.executeQuery(sql.toString());
      while (result.next()) {
            declinationZone = result.getInt(1);
            numberWithinZone = result.getInt(2);
            catalogPrefix = result.getString(3);
            note = result.getString(4);
            zoneSign = result.getString(5);
           keyList.add(new StarPK(catalogPrefix, zoneSign,
           declinationZone,numberWithinZone,note));
           counter++;
             if (counter >= this.getMaxNumberOfResults()) {
                         result.close();
                           this.statement.close();
                         return keyList;
            }
       }
       this.statement.close();
       } catch (Exception e) {
       throw new DAOException("[SAPDBDAO.findByWhereStmt] " + e);
       } finally {
             try {
                   this.release();
             } catch (Exception e) {}
             }
             return keyList;
       }
       public Collection findAll() throws DAOException {
             ArrayList keyList = new ArrayList();
             int declinationZone = -1;
             int numberWithinZone = -1;
             String catalogPrefix = null;
             String note = null;
             String zoneSign = null;
             int counter = 0;
   try {
       StringBuffer sql = new StringBuffer();
       this.acquire();
       sql.append("SELECT t0_o.declinationZone, t0_o.numberWithinZone,
t0_o.catalogPrefix, t0_o.note, t0_o.zoneSign FROM ").append(TABLE_NAME).append
(" t0_o");
      statement = getConnection().createStatement();
      ResultSet result = statement.executeQuery(sql.toString());
      while (result.next()) {
             declinationZone = result.getInt(1);
             numberWithinZone = result.getInt(2);
             catalogPrefix = result.getString(3);
             note = result.getString(4);
             zoneSign = result.getString(5);
             keyList.add(new StarPK(catalogPrefix,zoneSign,
```

```
            declinationZone,numberWithinZone,      note));
            counter++;
            if (counter >= this.getMaxNumberOfResults()) {
                  result.close();
                  this.statement.close();
                  return keyList;
            }
       }
            this.statement.close();
       } catch (Exception e) {
       throw new DAOException("[SAPDBDAO.findByWhereStmt] " + e);
            } finally {
                  try {
                        this.release();
                  } catch (Exception e) {   }
         }
             return keyList;
      }
      public StarVO loadData(StarPK starPK) throws DAOException {
            StarVO retVal = new StarVO();
            int count = 0;
            try {
            StringBuffer sql = new StringBuffer();
            this.acquire();
            sql.append("SELECT declinationZone, numberWithinZone, catalogPrefix,
note, zoneSign,signOfDeclination, rightAscentionSeconds, rightAscensionHours,
magnitude, rightAscensionMinutes, declinationDegrees, declinationMinutes FROM
STARBEAN WHERE ");
sql.append(" declinationZone = "+
starPK.getDeclinationZone()     + " AND numberWithinZone = "
             + starPK.getNumberWithinZone()+ " AND catalogPrefix = \'"
             + starPK.getCatalogPrefix()  + "\' AND note = \'"
             + starPK.getNote()+ "\' AND zoneSign =  \'"
             + starPK.getZoneSign()+ "\'");
            statement = getConnection().createStatement();
      ResultSet result = statement.executeQuery(sql.toString());
      while (result.next()) {

      retVal.setDeclinationZone(result.getInt(1));
      retVal.setNumberWithinZone(result.getInt(2));
      retVal.setCatalogPrefix(result.getString(3));
      retVal.setNote(result.getString(4));
      retVal.setZoneSign(result.getString(5));
      retVal.setSignOfDeclination(result.getString(6));
      retVal.setRightAscentionSeconds(result.getInt(7));
      retVal.setRightAscensionHours(result.getInt(8));
      retVal.setMagnitude(result.getFloat(9));
      retVal.setRightAscensionHours(result.getInt(10));
      retVal.setDeclinationDegrees(result.getInt(11));
      retVal.setDeclinationMinutes(result.getInt(12));
       }
```

```
            this.statement.close();
        } catch (Exception e) {
          throw new DAOException("[SAPDBDAO.loadData] " + e);
        } finally {
            try {
            this.release();
            } catch (Exception e) {}
           return retVal;
            }
        }
        public void delete(StarPK key) throws DAOException {
        try {
            StringBuffer sql = new StringBuffer();
            this.acquire();
            sql.append("DELETE FROM ").append(TABLE_NAME).append(
            " WHERE declinationZone = "+
key.getDeclinationZone()
            + " AND numberWithinZone = "+ key.getNumberWithinZone()
            + " AND catalogPrefix = \'"+
key.getCatalogPrefix()
            + "\' AND note = \'"+
key.getNote()+ "\' AND zoneSign =  \'"+ key.getZoneSign()+ "\'");
            statement = getConnection().createStatement();
            statement.executeUpdate(sql.toString());
           this.statement.close();
        } catch (Exception e) {
            throw new DAOException("[SAPDBDAO.delete] " + e);
            } finally {
                    try {
                        this.release();
                    } catch (Exception e) {}
           }
        }
        public void store(StarVO starVO) throws DAOException {
            try {
           StringBuffer sql = new StringBuffer();
           this.acquire();
sql.append("UPDATE ").append(TABLE_NAME).append(" SET ");
           sql.append(" catalogPrefix =  \'").append(
           starVO.getCatalogPrefix()).append(      "\',");
      sql.append("  zoneSign =  \'")
       .append(starVO.getZoneSign()).append("\',");
           sql.append(" declinationZone =  ").append(
           starVO.getDeclinationZone()).append(" ,");
           sql.append(" numberWithinZone =  \'").append(
           starVO.getNumberWithinZone());
sql.append("  note =  \'").append(starVO.getNote()).append("\',");
sql.append(" magnitude =  ").append(starVO.getMagnitude()).append(" ,");
           sql.append(" rightAscensionHours =  ")
           .append(starVO.getRightAscensionHours()).append(" ,");
           sql.append(" rightAscensionMinutes =  ")
```

```
            .append(starVO.getRightAscensionMinutes())
            .append(" ,");
            sql.append(" rightAscentionSeconds =  ")
            .append(starVO.getRightAscentionSeconds()).append(" ,");
            sql.append("  signOfDeclination =  \'")
            .append(starVO.getSignOfDeclination()).append("\',");
            sql.append(" declinationDegrees =  ")
            .append(starVO.getDeclinationDegrees()).append(" ,");
            sql.append(" declinationMinutes =  ")
            .append(starVO.getDeclinationMinutes()).append(" ,");
            sql.append(" WHERE declinationZone = "
            + starVO.getDeclinationZone()+ " AND numberWithinZone = "
            + starVO.getNumberWithinZone()+ " AND catalogPrefix = \'"
            + starVO.getCatalogPrefix()      + "\' AND note = \'"
            + starVO.getNote()+ "\' AND zoneSign =  \'"
            + starVO.getZoneSign()+ "\'");
            statement = getConnection().createStatement();
            statement.executeUpdate(sql.toString());
            this.statement.close();
            } catch (Exception e) {
                throw new DAOException("[SAPDBDAO.delete] " + e);
            } finally {
                  try {
                        this.release();
                  } catch (Exception e) {}
             }
    }
    public void activate() throws DAOException {}
    public void passivate() throws DAOException {      }
```

3.7.2 Praxis

Obwohl technisch möglich, sollte eine EntityBean niemals direkt von Clients angesprochen werden. Um unsere Messungen praxisnah zu gestalten, wurde auch hier indirekt über eine Session Facade auf die Entity zugegriffen. Bei der Suche können mehr Datensätze zurückgegeben werden, als der Client »verkraften« kann. In unserem Fall könnten es alle 325.000 Sterne sein. Bei dieser Datenmenge ist es sinnvoll, die Datensätze in dem EJB-Container zu cachen. Innerhalb dessen könnte der Client bequem navigieren, um nur die für ihn interessanten Datensätze zu holen.

Dieser Mechanismus wird von dem Value List Handler implementiert. Dabei handelt es sich um eine Stateful SessionBean, die für das Caching der Ergebnismenge zuständig ist. Diese kommuniziert mit der Session Facade (eine Stateless SessionBean) die lediglich den Zugriff auf die `StarBean` bzw. `StarBeanBMP` kapselt. Es wird jeweils das gleiche EAR (Enterprise Archive) deployed, allerdings mit unterschiedlicher Implementierung der Entity.

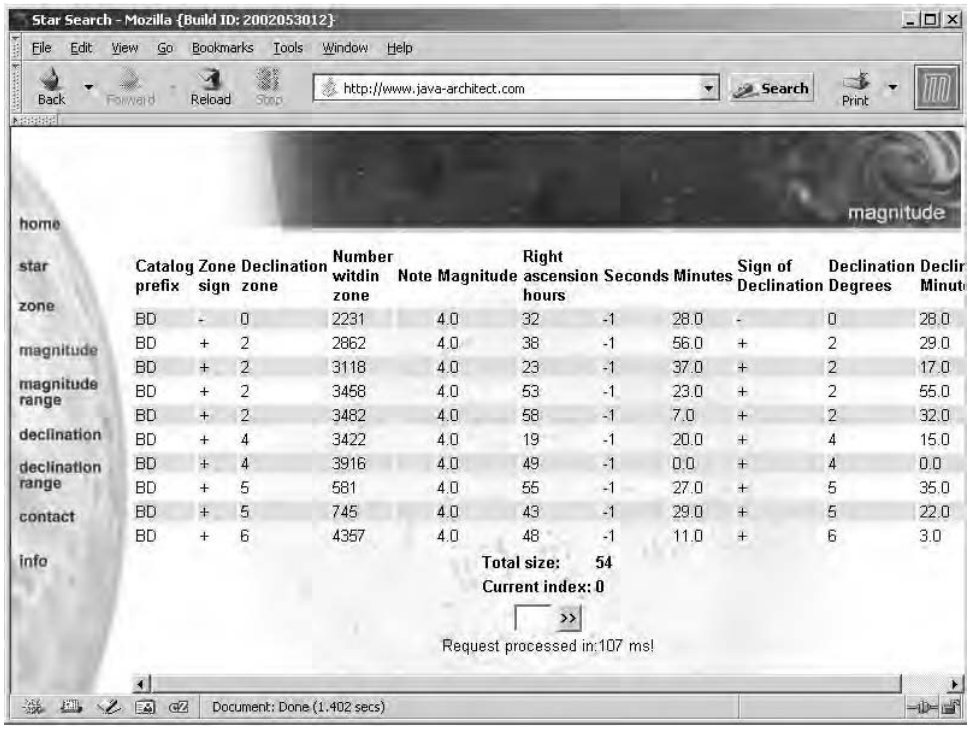


Abbildung 3.30: Die Navigation innerhalb einer Ergebnismenge des Webclients

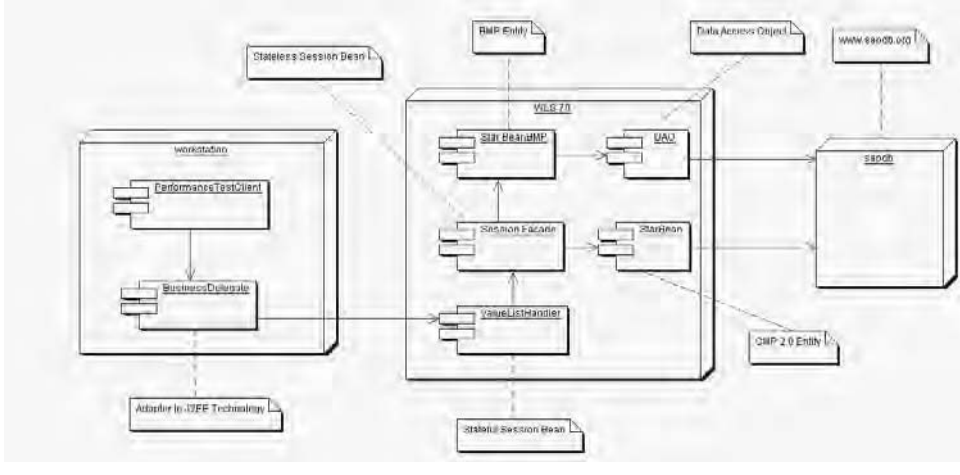


Abbildung 3.31: Der Zugriff des Clients auf die EntityBean

3.7.3 Performance

Für die Messung der Performance wurde ein Client geschrieben, der ausgewählte »finder«-Methoden der `StarBean` bzw. `StarBeanBMP` aufruft. Dieser kennt lediglich ein Business Delegate, der hier die EJB-Technologie abstrahiert. Der `PerformanceTestClient` wird innerhalb von JProbe 4.0 (*www.sitraka.com*) ausgeführt. Der Client muss nicht für die Messungen der CMP 2.0 und BMP-Varianten modifiziert werden.

```
public class PerformanceTestClient {
 private StarDelegateIF starDelegate = null;
 private StarDelegateFactory factory = null;
 public void init() throws Exception {
  this.factory = new
StarDelegateFactory("com.abien.j2ee.starsearch.delegate.StarDelegate");
  this.starDelegate = this.factory.getStarDelegateIF(false);
 }
 public void magnitudeSearch(int loops) throws SearchException {
      for (float i = 0; i < loops; i++)
            this.starDelegate.searchByMagnitude(i);
}
public void magnitudeInRange(int from,int to) throws SearchException {
 this.starDelegate.searchByMagnitudeInRange((float)from,(float)to);
}
public void declinationInRangeSearch(int from, int to) throws SearchException {
      this.starDelegate.searchByDeclinationInRange(from, to);
}
public static void main(String[] args) throws Exception {
 PerformanceTestClient testClient = new PerformanceTestClient();
      testClient.init();
      for(int i=0;i<10;i++){
            testClient.magnitudeSearch(i);
      }
      for(int i=0;i<5;i++){
            testClient.declinationInRangeSearch(1,i);
      }
      for(int i=0;i<5;i++){
            testClient.magnitudeInRange(1,i);
      }
}
}
```

Messung der CMP 2.0 Performance

Zuerst wurde die Anwendung mit der CMP 2.0 EntityBean deployed. Es wurden keine Optimierungen der CMP-Persistenz vorgenommen. Für die Suche wurden 92,353 sec benötigt.

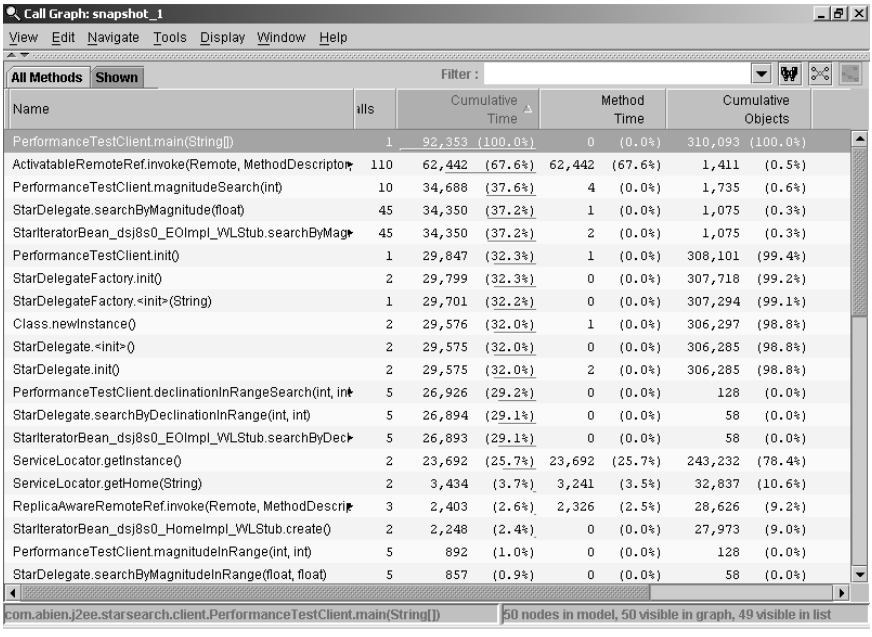


Abbildung 3.32: Die Performance der CMP 2.0 StarBean

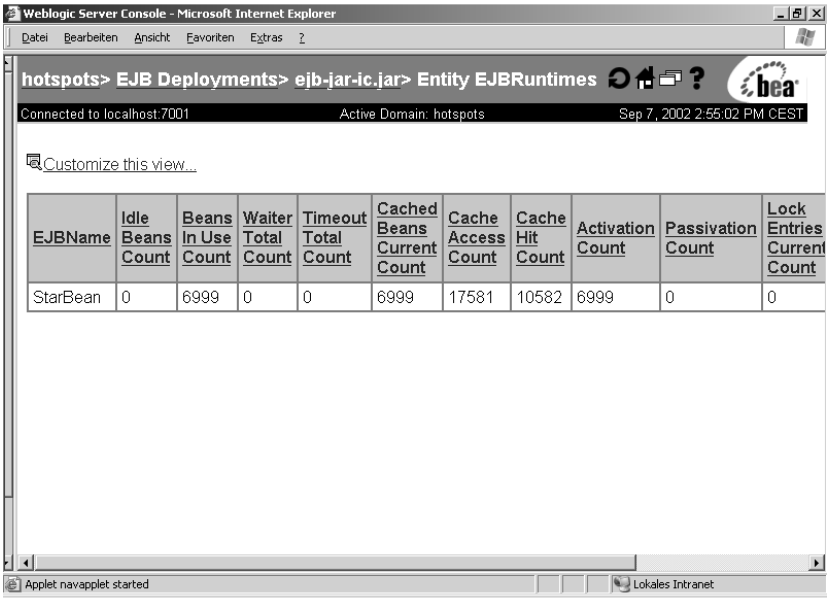


Abbildung 3.33: Monitoring-Ausgabe des Weblogic 7.0 Servers

Nach dem Testlauf waren 6.999 Instanzen der StarBean aktiv. Von 17.581 Cache-Zugriffen waren 10.582 erfolgreich. Es konnten 60% der Anfragen bereits durch den Cache beantwortet werden. In diesem Fall war der Zugriff auf die Datenbank nicht mehr notwendig. Mit einem Cache kann sowohl die Datenbank als auch der Applikationsserver entlastet werden. Die Konvertierung der elementaren Datentypen in Objektinstanzen ist nicht mehr notwendig – es kann sofort mit bereits gültigen Datensätzen gearbeitet werden. Die Cacheeinstellungen wurden nicht in der J2EE-Spezifikation festgeschrieben, sodass es sich hier um eine BEA-spezifische Erweiterung handelt. Die Anwendung bleibt jedoch weiterhin »J2EE-konform«, da es sich hier lediglich um zusätzliche Einstellungsmöglichkeiten handelt.

Messung der CMP 2.0 Performance (Finders Load Bean Einstellung)

Der Weblogic Application Server 7.0 erlaubt einige Einstellmöglichkeiten der CMP 2.0 Persistenz. Mit der Einstellung »Finders Load Bean« kann eine Menge von EntityBean-Instanzen mit einer SQL-Abfrage gesucht und geladen werden. Der zweite Testlauf wurde mit dieser Einstellung vorgenommen.

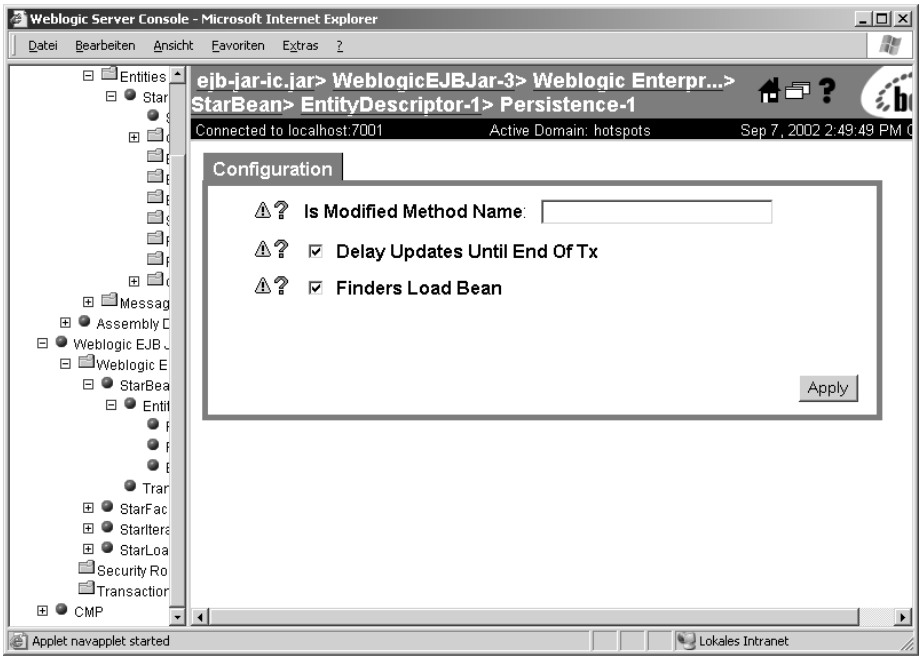


Abbildung 3.34: Einstellung »Finders Load Bean« in der Weblogic 7.0 Console

Die Performance konnte hier auf 79,343 sec verbessert werden. Mit dieser Einstellung konnten 10 sec eingespart werden.

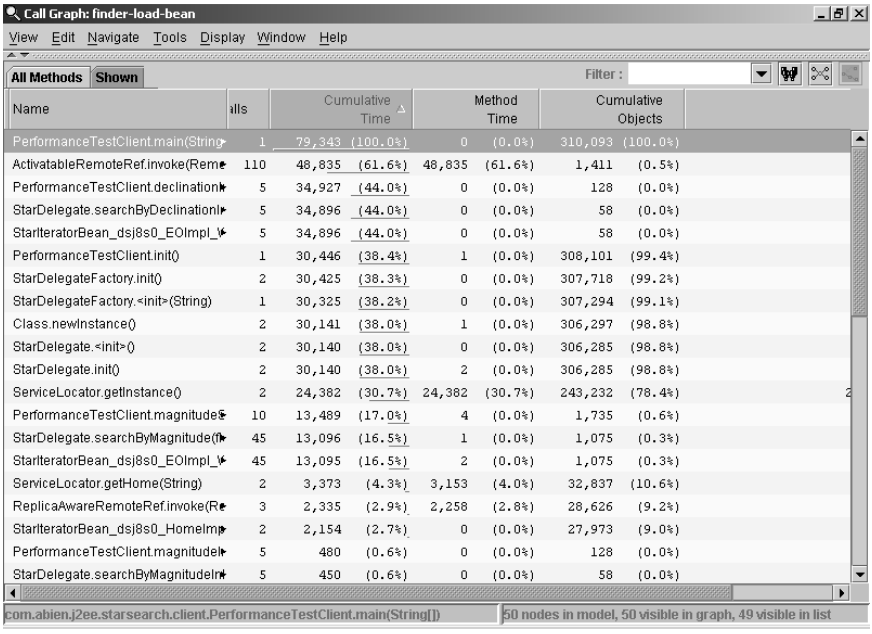


Abbildung 3.35: Die Performance der CMP 2.0 StarBean mit der »Finder Load Bean« Einstellung

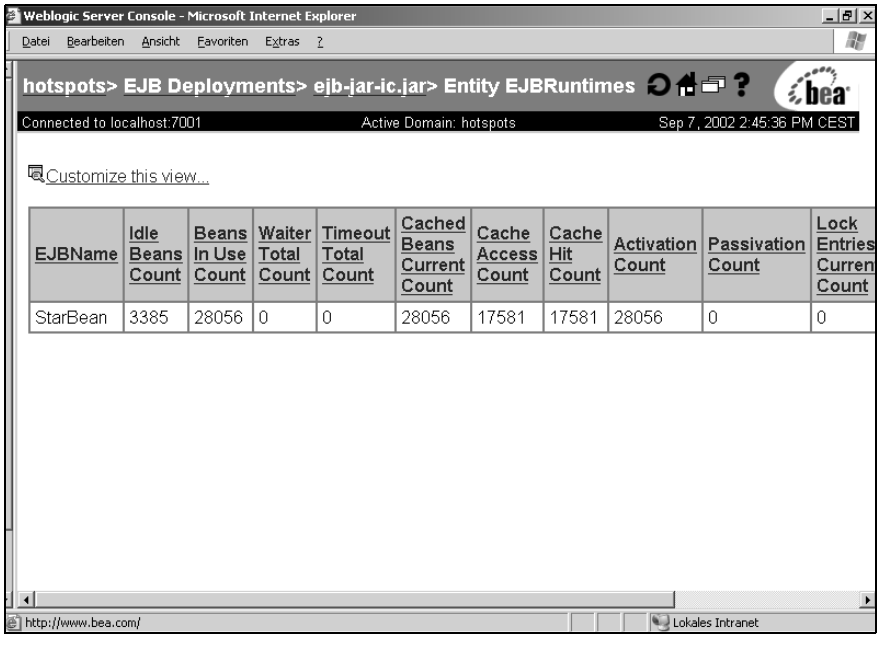
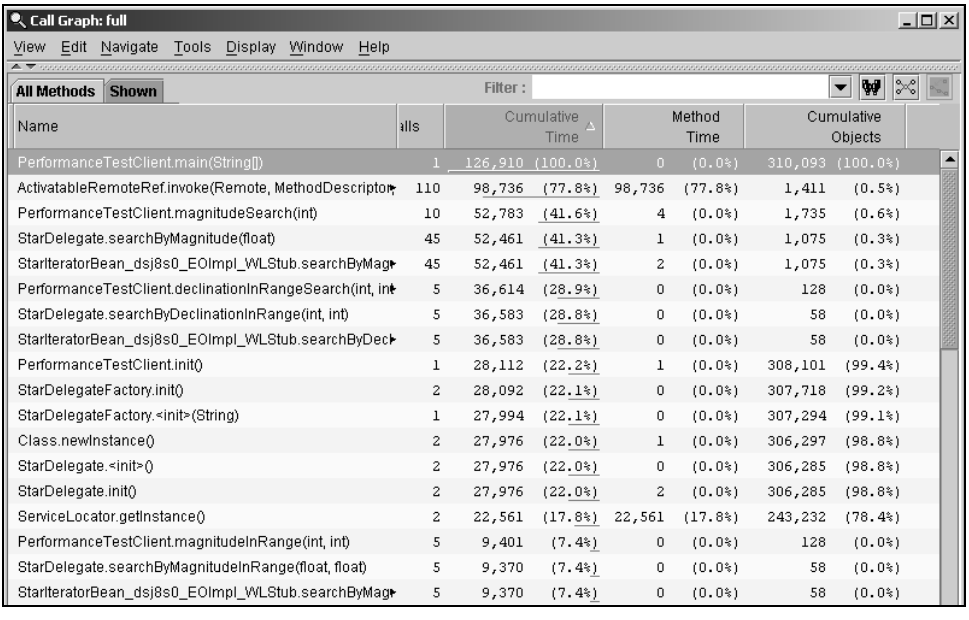


Abbildung 3.36: Monitoring-Ausgabe des Weblogic 7.0 Servers

Interessanterweise waren alle 17.581 Cacheanfragen erfolgreich. Die Erfolgsrate betrug hier ganze 100%. Die Anzahl der aktiven Instanzen ist auf 28.056 gestiegen. Es wurden also Beans geladen, die gar nicht benutzt wurden. Mit der höheren Anzahl der Instanzen wächst natürlich auch der Speicherbedarf des Applikationsservers.

Messung der BMP-Performance

Für die Messung der BMP-Performance wurde die `StarBeanBMP` deployed. Die Suche des `PerformanceTestClient` hat ganze 126,910 sec gedauert. Es wurden also 30 sec mehr benötigt als mit der nicht optimierten CMP-Persistenz.



Name	alls	Cumulative Time		Method Time		Cumulative Objects	
PerformanceTestClient.main(String[])	1	126,910	(100.0%)	0	(0.0%)	310,093	(100.0%)
ActivatableRemoteRef.invoke(Remote, MethodDescriptor	110	98,736	(77.8%)	98,736	(77.8%)	1,411	(0.5%)
PerformanceTestClient.magnitudeSearch(int)	10	52,783	(41.6%)	4	(0.0%)	1,735	(0.6%)
StarDelegate.searchByMagnitude(float)	45	52,461	(41.3%)	1	(0.0%)	1,075	(0.3%)
StarIteratorBean_dsj8s0_EOImpl_WLStub.searchByMag	45	52,461	(41.3%)	2	(0.0%)	1,075	(0.3%)
PerformanceTestClient.declinationInRangeSearch(int, int	5	36,614	(28.9%)	0	(0.0%)	128	(0.0%)
StarDelegate.searchByDeclinationInRange(int, int)	5	36,583	(28.8%)	0	(0.0%)	58	(0.0%)
StarIteratorBean_dsj8s0_EOImpl_WLStub.searchByDec	5	36,583	(28.8%)	0	(0.0%)	58	(0.0%)
PerformanceTestClient.init()	1	28,112	(22.2%)	1	(0.0%)	308,101	(99.4%)
StarDelegateFactory.init()	2	28,092	(22.1%)	0	(0.0%)	307,718	(99.2%)
StarDelegateFactory.<init>(String)	1	27,994	(22.1%)	0	(0.0%)	307,294	(99.1%)
Class.newInstance()	2	27,976	(22.0%)	1	(0.0%)	306,297	(98.8%)
StarDelegate.<init>()	2	27,976	(22.0%)	0	(0.0%)	306,285	(98.8%)
StarDelegate.init()	2	27,976	(22.0%)	2	(0.0%)	306,285	(98.8%)
ServiceLocator.getInstance()	2	22,561	(17.8%)	22,561	(17.8%)	243,232	(78.4%)
PerformanceTestClient.magnitudeInRange(int, int)	5	9,401	(7.4%)	0	(0.0%)	128	(0.0%)
StarDelegate.searchByMagnitudeInRange(float, float)	5	9,370	(7.4%)	0	(0.0%)	58	(0.0%)
StarIteratorBean_dsj8s0_EOImpl_WLStub.searchByMag	5	9,370	(7.4%)	0	(0.0%)	58	(0.0%)

Abbildung 3.37: Die Performance der BM- StarBeanBMP

Die Anzahl der aktiven Instanzen und auch der Cache-Zugriffe ist mit der CMP-Persistenz identisch. Der EJB-Container hat auf die BMP-EntityBean genauso oft zugegriffen wie auf die CMP-Instanzen. Es wird in dem Fall die gleiche Caching-Strategie für die CMP und BMP Persistenz verwendet.

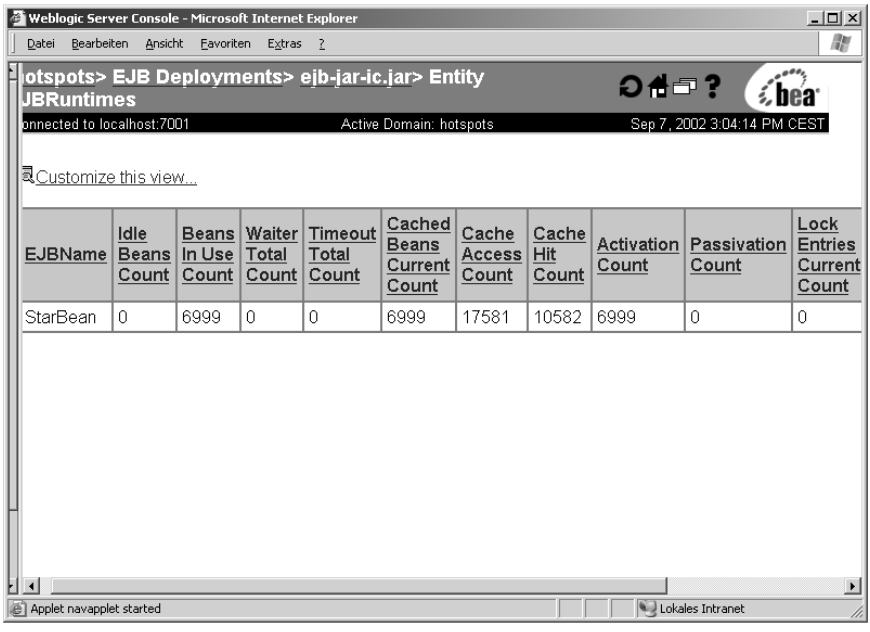


Abbildung 3.38: Monitoring-Ausgabe des Weblogic 7.0 Servers

Messung der BMP-Performance (Ergebnismenge auf 1.000 eingeschränkt)

Bei dieser Messung wurde die BMP-Bean optimiert. Es wurden nur die ersten 1.000 gefundenen Primärschlüssel zurückgegeben und somit auch höchstens 1.000 Instanzen pro Anfrage erzeugt.

```
private Collection findKeysByWhereStmt(String whereStatement)
            throws DAOException {
//…
statement = getConnection().createStatement();
 ResultSet result = statement.executeQuery(sql.toString());
      while (result.next()) {
            declinationZone = result.getInt(1);
            numberWithinZone = result.getInt(2);
            catalogPrefix = result.getString(3);
            note = result.getString(4);
            zoneSign = result.getString(5);
            keyList.add(new StarPK(catalogPrefix, zoneSign,
                    declinationZone,numberWithinZone,note));
            counter++;
            if (counter >= this.getMaxNumberOfResults()) {
                        result.close();
                        this.statement.close();
                        return keyList;
            }
      }
```

Diese Vorgehensweise kann beispielsweise bei Anwendungen benutzt werden, die nur eine beschränkte Datenmenge verarbeiten können. Bei einer Suchmaschine ist es beispielsweise unwahrscheinlich, dass ein Benutzer alle Suchergebnisse (es können einige Millionen sein) darstellen möchte.

Für die optimierte BMP-Suche hat der Client 79,019 sec benötigt. Die Performance der Suche konnte um über 47 sec verbessert werden.

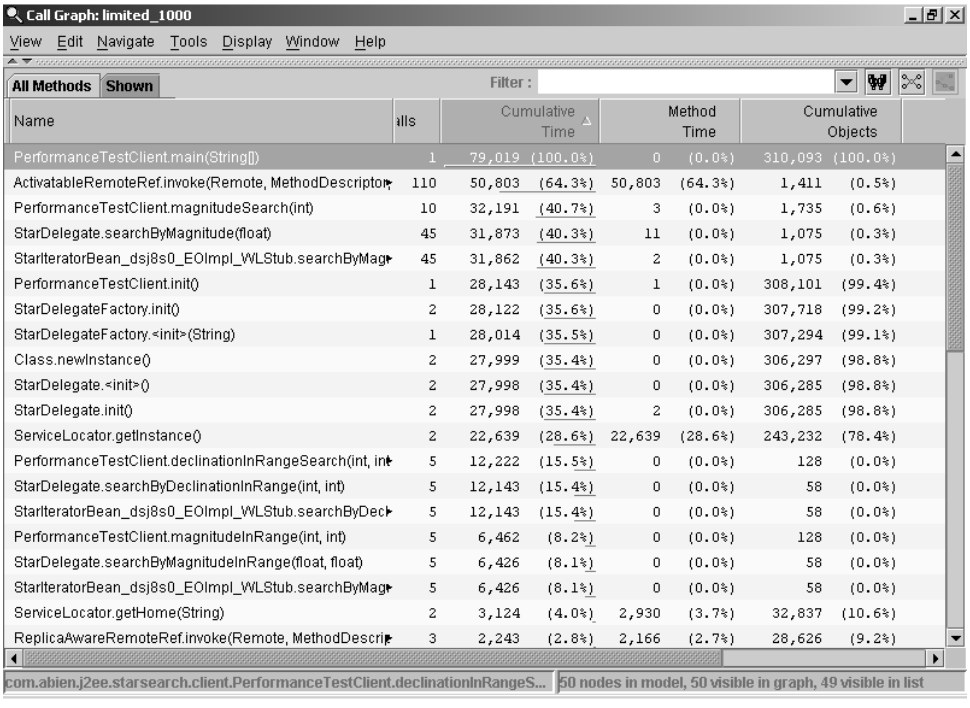


Abbildung 3.39: Die Performance der BM- StarBeanBMP. Die Suche wurde auf 1000 Datensätze eingeschränkt.

Da die Anzahl der Suchergebnisse bereits in der EntityBean eingeschränkt wurde, wurden wesentlich weniger Instanzen erzeugt als in der CMP-Persistenz. Die Anzahl der gleichzeitig aktiven Instanzen sank hier auf 4.024. Von 10.581 Cache-Anfragen waren leider nur 6.557 erfolgreich. Das Verhältnis blieb jedoch ungefähr gleich. Der Cache konnte jeweils ca. 60% der Anfragen erfolgreich beantworten.

Das beste Ergebnis wurde mit der BMP-Persistenz und der Einschränkung der Ergebnismenge auf die ersten 100 Datensätze erzielt. Dieses Feature kann leider nur für Anwendungsfälle verwendet werden, die nicht die ganze Ergebnismenge benötigen.

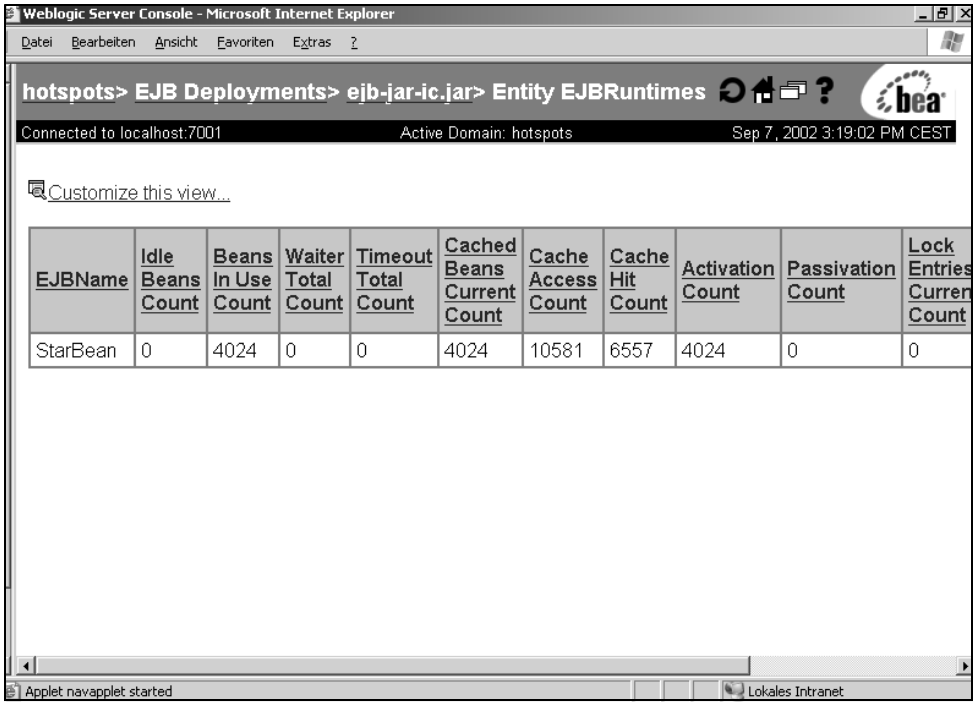


Abbildung 3.40: Monitoring-Ausgabe des Weblogic 7.0 Servers

Ansonsten war die CMP 2.0 Persistenz mit WLS 7.0 immer besser, als die BMP. Die CMP-Performance konnte weiter mit der »Finders Load Bean« Einstellung verbessert werden, da hier mit einer SQL-Abfage mehrere Bean-Instanzen geladen werden.

Zusammenfassung der Ergebnisse

	CMP 2.0	**CMP 2.0 (Finders Load Bean)**	**BMP**	**BMP (Einschränkung der Suche auf 1000 Datensätze)**	**BMP (Einschränkung der Suche auf 100 Datensätze)**
Zeit in ms (Weblogic Server 7.0.1)	92.353	79.343	126.910	79.019	56.562
Zeit in ms (JBoss 3.02)	Nicht möglich (Abbruch mit einem Fehler)	Nicht möglich (Abbruch mit einem Fehler)	133.081	100.378	63.429

3.7.4 Bewertung der Ergebnisse

Die Performance-Unterschiede zwischen der BMP und CMP 2.0 Persistenz kamen nicht überraschend. Die Standardeinstellung der CMP-Persistenz und auch die BMP-Persistenz benötigen für das Laden von n Instanzen n+1 SQL Abfragen. Die EJB sucht zunächst nach den Primärschlüsseln und gibt die in einer Collection an den EJB-Container zurück. Für die Suche wird bereits eine SQL-Abfrage benötigt. In unserem Fall handelte es sich um die folgende SQL-Abfrage: `SELECT declinationZone, numberWithinZone, catalogPrefix, note, t0_o.zoneSign FROM STARBEAN WHERE magnitude = 1.0`

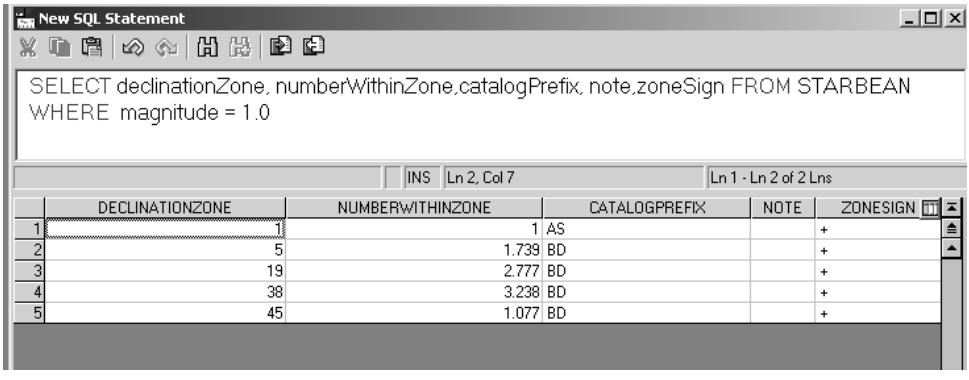


Abbildung 3.41: Die Suche nach den Primärschlüsseln einer EntityBean

Die `StarBean` wird durch einen »Composite-Key« eindeutig identifiziert, sodass auch alle beteiligten Spalten aus der Datenbank gelesen werden müssen. Der Primärschlüssel der Bean benötigt für die Initialisierung die folgenden Attribute `declinationZone`, `numberWithinZone`, `catalogPrefix`, `note`.

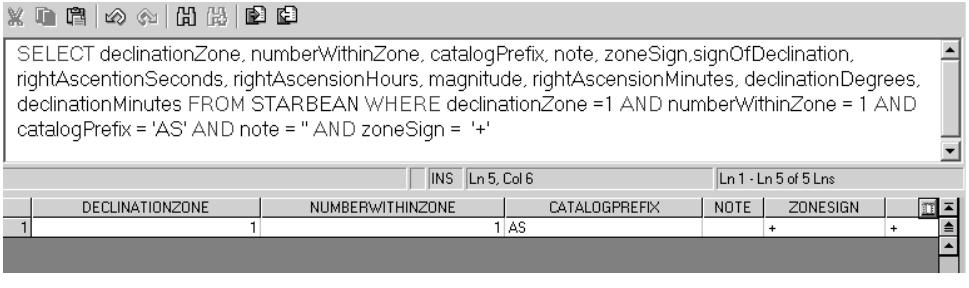


Abbildung 3.42: Laden der Entity mit einem bereits gefundenen Primärschlüssel

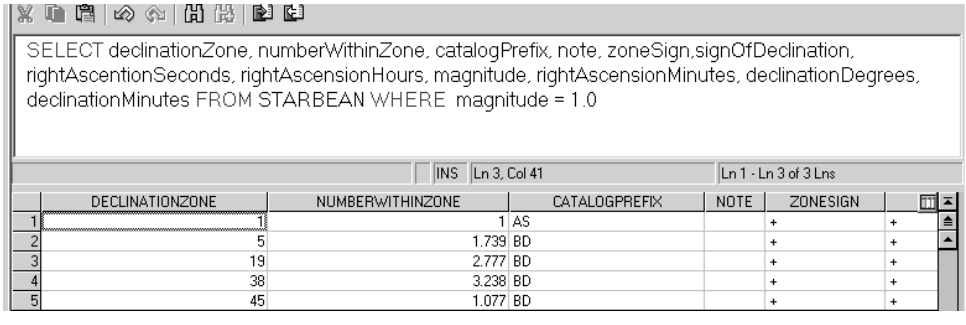
Erst beim Aufruf der ersten Methode der EntityBean wird diese geladen. Man spricht hier von sog. »Lazy Loading«. Dabei wird für jeden gefundenen Primärschlüssel die Methode `ejbLoad` und somit die folgende Abfrage gestartet: `SELECT declinationZone, numberWithinZone, catalogPrefix, note, zoneSign,signOfDeclination, rightAscentionSeconds, rightAscensionHours, magnitude, rightAscensionMinutes, declinationDegrees, declinationMinutes FROM STARBEAN WHERE declinationZone =1 AND numberWithinZone = 1 AND catalogPrefix = 'AS' AND note = '' AND zoneSign = '+'`

Wie Abbildung 1.42 zeigt, wird hier ein einzelner Datensatz zurückgegeben. Bei mehreren Datensätzen wäre die Eindeutigkeit des Primärschlüssels verletzt worden.

Falls nur wenige Daten aus einer größeren Ergebnismenge benötigt werden, ist diese Strategie sinvoll, da somit nur für jeden tatsächlichen Zugriff des Clients eine EntityBean Instanz erzeugt wird. Allerdings befinden sich auch wesentlich weniger EntityBeans im Cache.

CMP 2.0 Persistenz

Da für diese Persistenz der EJB-Container bzw der Persistenz-Manager zuständig ist, kann sie auch wesentlich flexibler gehandhabt werden. Bei der Einstellung »Finders Load Bean« sucht und lädt der EJB-Container alle Instanzen mit einer SQL-Abfrage. Es könnte sich dabei um den folgenden SQL-Code handeln: `SELECT declinationZone, numberWithinZone, catalogPrefix, note, zoneSign,signOfDeclination, rightAscentionSeconds, rightAscensionHours, magnitude, rightAscensionMinutes, declinationDegrees, declinationMinutes FROM STARBEAN WHERE magnitude = 1.0`. Dem EJB-Container steht es natürlich frei, die SQL-Abfrage weiter zu optimieren.



	DECLINATIONZONE	NUMBERWITHINZONE	CATALOGPREFIX	NOTE	ZONESIGN	
1	1	1	AS		+	+
2	5	1.739	BD		+	+
3	19	2.777	BD		+	+
4	38	3.238	BD		+	+
5	45	1.077	BD		+	+

Abbildung 3.43: Das optimierte Laden der EJB-Instanzen

Mit der Reduzierung der SQL-Abfragen (»Finders Loads Beans«) konnte die Performance weiter verbessert werden. Allerdings wurde hier für jeden gefundenen Datensatz sofort eine EJB-Instanz erzeugt. Bei sehr großen Ergebnismengen, könnte es zu Problemen mit dem Speicher der JVM kommen (`java.lang.OutOfMemoryError`).

Der EJB-Container verwendet bei der CMP für die Datenbankzugriffe bereits vorkompilierte `java.sql.PreparedStatements`. Dies trägt weiter zu der Performance-Verbesserung bei, da einmal erzeugte PreparedStatements wieder verwendet werden können.

Bei der Implementierung mancher Anwendungsfälle möchte man die Ergebnismenge jedoch auf die ersten Datensätze einschränken. Dies ist besonders für Geräte mit wenig Speicher sinnvoll. Da die Clients nicht direkt mit einer EntityBean kommunizieren sollten, wird hier auf das Session-Facade-Pattern zurückgegriffen. Die Session-Facade könnte die Ergebnismenge zwar einschränken, diese muss jedoch warten, bis die Suche nach den Datensätzen beendet ist. Obwohl man so die Größe des Ergebnisses einschränken könnte, muss man mit der schlechteren Performance der Anwendung leben. Eine Optimallösung wäre die Einschränkung der zurückgegebenen Menge von Primärschlüsseln (wie in der Beispielanwendung) oder noch besser – die Einschränkung der Ergebnismenge bereits in der Datenbank. Dies ist jedoch mit dem aktuellen CMP 2.0 bzw. CMP 2.1 Standards leider nicht möglich. Man ist hier auf die Erweiterungen des Applikationsserver-Herstellers angewiesen. Dann ist jedoch die CMP-Entity nicht auf anderen Applikationsservern lauffähig.

BMP-Persistenz

Der Entwickler hat bei der Wahl dieser Persistenz schlechtere Karten. Die Verwendung von `java.sql.PreparedStatements` ist in einer J2EE-Umgebung (noch) problematisch. In einer J2EE-Umgebung arbeitet man typischerweise mit Connection-Pools, da man den Mehraufwand für die Initialisierung einer Connection vermeiden möchte. Ein `PreparedStatement` wird bei seiner »Vorbereitung« oder Vorkompilierung mit einer `java.sql.Connection` assoziert. Es müssten also alle `Connection`-Instanzen aus dem Connection-Pool entfernt werden, um die `PreparedStatements` vorbereiten zu können. Nach dieser Initialisierungsphase, könnte man diese wie gewohnt verwenden. Viele Applikationsserver bieten bereits `PreparedStatement`s-Pools an, diese sollten jedoch aus Kompatibilitätsgründen nicht verwendet werden. Für Abhilfe sorgt hier die JDBC 3.0-API, da `PreparedStatements`-Pools in den JDBC- und somit auch J2EE-Standard aufgenommen werden.

Auch auf die Ladestrategien des EJB-Containers hat der Entwickler nur wenig Einfluss. Die SQL-Statements wurden in der EntityBean oder der DAO-Komponente »hardcodiert«, sodass es keine Möglichkeiten für »Finders Load Bean«-Abfragen mehr gibt. Das n+1-Problem bleibt also bestehen.

Da der Entwickler jedoch die volle Kontrolle über die Persistenz der Bean hat, ist es hier natürlich möglich, die Ergebnismenge mit einer Erweiterung der SQL-Statements einzuschränken. Falls die gewählte Datenbank dieses Feature nicht unterstützen sollte, können auch nur die ersten n-Datensätze aus dem `ResultSet` gelesen und in Primärschlüssel umgewandelt werden.

```
while (result.next()) {
      declinationZone = result.getInt(1);
      numberWithinZone = result.getInt(2);
      catalogPrefix = result.getString(3);
      note = result.getString(4);
      zoneSign = result.getString(5);
      keyList.add(new StarPK(catalogPrefix, zoneSign,
              declinationZone,numberWithinZone,note));
      counter++;
      if (counter >= this.getMaxNumberOfResults()) {
                  result.close();
                  this.statement.close();
                  return keyList;
      }
}
```

Die `ejbFind`-Methoden könnten so die Ergebnismenge »künstlich« einschränken, um unnötige Assoziationen der Primärschlüssel mit den Instanzen aus dem freien Pool zu verhindern. Für den Client und somit auch für die Session-Facade, bleibt dieser Vorgang vollkommen transparent. Die Home- und LocalHome-Interfaces können natürlich auch nur eine eingeschränkte Menge an Remote- bzw. Local-Instanzen an den Client zurückgeben. Die Zeit für die Assoziation der PK mit den freien Instanzen aus dem Pool oder vielleicht sogar die Erzeugung neuer Instanzen kann hier gespart werden.

3.7.5 Fazit

Die Flexibilität und »Intelligenz« des EJB-Containers ermöglicht eine effiziente Verwaltung der EJB-Instanzen und somit eine bessere Performance der Anwendung. Auch die Konfigurierbarkeit der CMP 2.0 Persistenz ermöglicht eine nachträgliche Anpassung der Lade- und Update-Strategien des EJB-Containers. In unserem Fall war die CMP-Persistenz immer besser als die BMP. Lediglich bei der Einschränkung der Ergebnismenge konnte die BMP-Performance bessere Werte erzielen, als die CMP 2.0 Persistenz.

Es wird oft versucht, mit Rahmenwerken und vordefinierten Vorgehensweisen den Entwickler bei der Implementierung und Entwurf von J2EE-Anwendungen zu unterstützen. Es ist jedoch gefährlich, sich endgültig für die BMP oder CMP 2.0 Persistenz zu entscheiden. Es gibt immer wieder Ausnahmefälle, für die sich die Verwendung von BMP-Persistenzen lohnt. Meiner Meinung nach sollte man auf jeden Fall mit der CMP 2.0 Persistenz beginnen. Falls man auf eigenen SQL-Code angewiesen ist, kann man jederzeit ohne größeren Aufwand auf die BMP-Strategie migrieren. Oft lassen sich die bereits generierten SQL-Statements als Basis für die »Eigenetwicklungen« wieder verwenden.

3.8 Wann wird eine EntityBean mit der Datenbank synchronisiert – oder was sind »commit-options«?

Während meiner EJB- und J2EE-Kurse sind sich die Teilnehmer oft unsicher, wann und wie der EJB-Container den Zustand der EJB-Instanzen mit der Persistenzschicht synchronisiert. Diese Frage gehört zu den meistgestellten, deswegen versuche ich diese hier zu beantworten. Eigentlich könnte meine Antwort so klingen: »Die Synchronisation der Bean-Instanzen mit der Datenbank hängt von den eingestellten commit-options der EntityBean ab«. Leider geben sich die Kursteilnehmer mit dieser Antwort nicht zufrieden – es wird sofort die nächste Frage gestellt: Was sind die »commit-options«? Somit gehört diese Frage auch zu der Kategorie der meistgestellten Fragen und wird nun auch beantwortet.

3.8.1 Problemstellung

Die Klasse einer EntityBean könnte mit einer Tabelle einer Datenbank verglichen werden. Eine Instanz dieser Klasse entspricht einem Datensatz aus dieser Tabelle. Dadurch werden die Daten der Tabelle »gecached« und somit redundant im EJB-Container gespeichert. Bei der Änderung des Zustandes dieser Instanz (oder dem Aufruf eines »Setters«) müsste der EJB-Container sofort den Zustand der Bean in die Datenbank zurückschreiben. Der EJB-Container ruft natürlich lediglich die Methode `ejbStore` auf – um alles andere kümmert sich die BMP- oder CMP-Beaninstanz. Eine relationale Datenbank würde an dieser Stelle das folgende SQL-Statement ausführen: `update <table> set column1 = ?1, column2 = ?2 where pk = ?3.`

Ähnlich sieht der umgekehrte Weg aus. Um den »Cache« mit der Datenbank zu synchronisieren wird die Methode `ejbLoad` aufgerufen. Hier wir mit dem Primärschlüssel nach den zugehörigen Daten gesucht. Die SQL-Abfrage könnte so aussehen: `select column1, column2, column3 from <table> where pk = ?1.`

An dieser Stelle stellt sich die Frage, wann die Synchronisierungsmethoden von dem Container aufgerufen werden. Ein wesentlicher Faktor ist die Transaktionssteuerung der Bean. Der Transaktionslevel der aufgerufenen Business-Methoden hat natürlich Einfluss auf die `ejbStore`/`ejbLoad`-Aufrufe des Containers. Der Transaktionslevel bestimmt, wann und ob überhaupt Transaktionen gestartet und committed werden. Der Entwickler erwartet ja, dass nach einem commit-Aufruf die Daten dauerhaft in der Datenbank abgelegt werden. Allerdings lässt sich die Synchronisierungstrategie des EJB-Containers mit Hilfe der commit-options wesentlich feiner einstellen.

3.8.2 Technischer Hintergrund

Die commit-options lassen sich pro EntityBean in dem herstellerabhängigen Deployment Descriptor festlegen. Die Einstellmöglichkeit gehört leider nicht in den Umfang des Standard-Deployment-Descriptors, was den Migrationsaufwand beim Wechsel des Applikationsservers erhöht. In der EJB 2.1 Spezifikation wurden die folgenden commit-options festgelegt:

1. **Option A**: Der Container »cached« die Bean-Instanzen zwischen den Transaktionen und stellt diese nicht in den Pool der »freien« Instanzen zurück. Der Container stellt sicher, dass die Instanz einen Exklusiv-Zugriff auf die Persistenzschicht erhält (Beispielsweise mit »`select for update`« SQL-Statement). Da der Zugriff der Entity »exklusiv« ist, muss die Methode `ejbLoad` bei der nächsten Transaktion nicht mehr aufgerufen werden. Diese Option ist mit den »pessimistischen Locks« vergleichbar.
2. **Option B**: Der Container »cached« die Bean-Instanzen zwischen den Transaktionen und stellt diese nicht in den Pool der »freien« Instanzen zurück. Der Container stellt jedoch nicht sicher, dass der Zugriff der Bean auf die Datenbank exklusiv ist. Aus dem Grund muss die Methode `ejbLoad` der Instanz bei Beginn der nächsten Transaktion aufgerufen werden.
3. **Option C**: Nach jeder Transaktion wird die Bean in den »freien« Pool gestellt. Somit ist der Container natürlich auch gezwungen die Methode `ejbLoad` sofort nach dem Beginn der nächster Transaktion der Bean aufzurufen.

Die commit-options sind sowohl für die Performance der Anwendung, als auch die Konsistenz der Daten sehr wichtig. Es könnte also passieren, dass eine Anwendung aufgrund falscher commit-options bereits von einer anderen Anwendung modifizierte und somit auch ungültige Daten liefert. Je nach der Komplizität der Abfragen kann eine zu restriktive Einstellung der commit-options Faktoren an Performance ausmachen.

	Schreibt den Zustand der Instanz in die Datenbank zurück. (`ejbStore` wird aufgerufen)	**Instanzen werden gecached – sie werden nicht in den Pool zurückgesteckt.**	**Die Daten der Instanzen bleiben gültig.**
Option A	Ja	Ja	Ja
Option B	Ja	Ja	Nein
Option C	Ja	Nein	Nein

Bei der Entwicklung ist es wichtig zu wissen, ob der Applikationsserver der einzige »Client« der Datenbank ist oder nicht. Wenn nicht, muss der Zustand jeder EntityBean-Instanz vor dem erfolgreichen Abschluss der Transaktion in die Datenbank zurückgeschrieben werden. Auch sofort nach dem »begin« einer Transaktion müssen die Daten

aus der Datenbank gelesen werden, um den Zustand der Bean-Instanz zu synchronisieren. In diesem Fall kommt die Option A in Frage. Der Container sperrt den Datensatz jeder einzelnen EntityBean, somit kann die Datenbank nicht modifiziert werden. Dies ermöglicht Caching von gültigen Instanzen. Allerdings werden andere Clients, die gerade mit dem Datensatz arbeiten, gesperrt, sodass der Durchsatz anderer Anwendung darunter leidet.

Falls die Datensätze nicht gesperrt werden können, empfiehlt es sich, mit der Option B zu arbeiten. Obwohl das Caching von gültigen Instanzen nicht möglich ist, lassen sich diese poolen. Nach dem »begin« einer neuen Transaktion muss jedoch die Methode `ejbLoad` aufgerufen werden. Man arbeitet immer mit den aktuellen Daten.

Die Option C ist die restriktivste, da hier weder die Daten, noch die Instanzen gepooled werden. In der Praxis findet diese Option kaum Anwendung.

3.8.3 Performance

Leider sind in der Praxis diese Einstellungen vielen Entwicklern gar nicht bekannt. Der Verlust der Datenintegrität, schlechte Performance der Anwendung und sogar Deadlocks können durch falsche comit-options-Einstellungen verursacht werden.

Für die Darstellung der Performance-Unterschiede wurde der `PerformanceTestClient` aus der Frage »Ist BMP schneller als die CMP 2.0 Persistenz?« wieder verwendet. Allerdings wurde hier die CMP-Variante der `StarBean` auf den WLS 7.0.1 deployed.

Da es sich hier um eine Sternesuchmaschine handelt, wäre natürlich die Read Only-Einstellung völlig ausreichend. Trotzdem wurde die Anwendung mehrmals mit unterschiedlichen commit-options deployed, um die Performance-Unterschiede zu demonstrieren.

Die commit-options lassen sich in dem BEA-spezifischen Deployment Deskriptor einstellen. Diese lassen sich mit der Einstellung »cache-between-transactions« abbilden. Falls »cache-between-transactions« eingeschaltet ist, entspricht dies der Option A, wenn nicht, der Option B. Die Option C ist nur schwer einstellbar. Es müsste sowohl der Cache als auch der Pool abgeschaltet werden. Somit könnte man den EJB-Container für die Erzeugung neuer Instanzen nach dem Beginn einer neuen Transaktion erzwingen. Leider findet diese Einstellung in der Praxis kaum Anwendung.

Die »Concurrency Strategies« bestimmen, auf welche Art und Weise die Integrität der Daten gesichert wird. Auch diese Einstellung ist sowohl für die Datenintegrität, als auch für die Performance der Anwendung sehr wichtig.

```
<entity-descriptor>
      <entity-cache>
      ...
      <concurrency-strategy>Database</concurrency-strategy>
```

```
        </entity-cache>
        ...
    </entity-descriptor>
```

Grundsätzlich lassen sich hier die folgenden concurrency-strategies einstellen:

1. **Exlusive**: Sperrt die aktuelle Entity-Instanz. Sorgt für den exklusiven Zugriff der Clients auf diese Instanz. Für die Sperre sorgt hier der EJB-Container. Diese Einstellung bietet beim eingeschalteten Cache eine gute Performance und stellt die Datenintegrität sicher, wenn es sich bei dem Applikationsserver um den einzigen Client der Datenbank handelt.
   Beim ausgeschalteten Cache wird zwar die Bean-Instanz während der Transaktion gesperrt und die Bean sofort nach dem Start der Transaktion neu geladen, die Daten in der Datenbank können aber jederzeit geändert werden. Mit dieser Einstellung kann es passieren, dass die Bean nach dem erfolgreichen Abschluss der Transaktion ungültig wird, da sich die Daten in der Persistenzschicht geändert haben. Diese Einstellung sollte lediglich verwendet werden, wenn die Änderung der Daten innerhalb einer Transaktion ausgeschlossen werden kann. Diese Einstellung könnte auch im Clusterbetrieb des Applikationsservers zu Problemen führen.

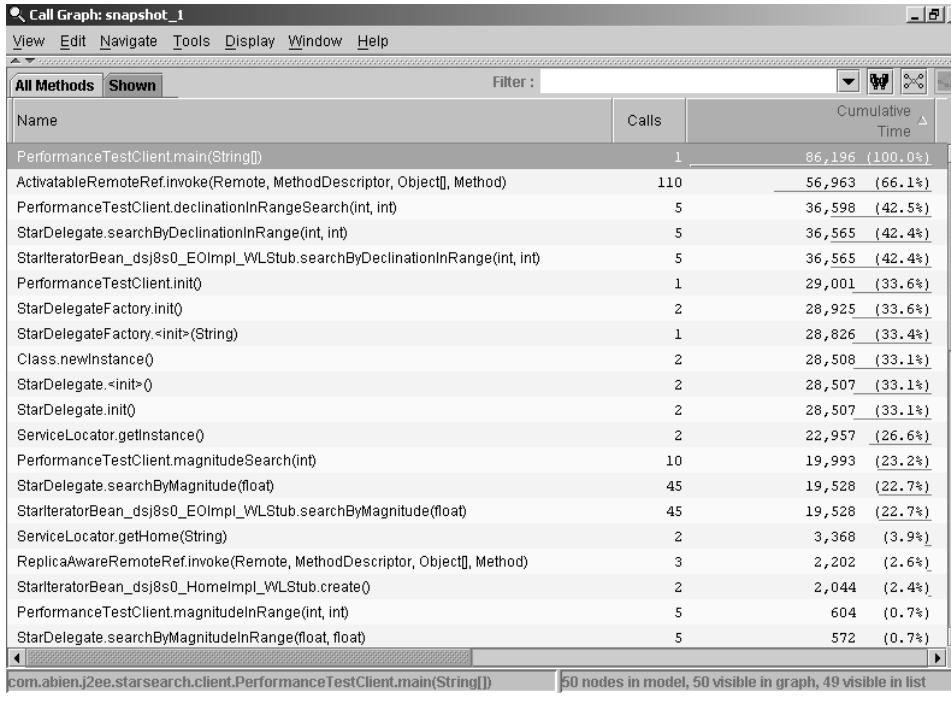


Abbildung 3.44: Die »Exclusive«-Einstellung mit Cache des Weblogic Applikationsservers 7.0.1 (Option A)

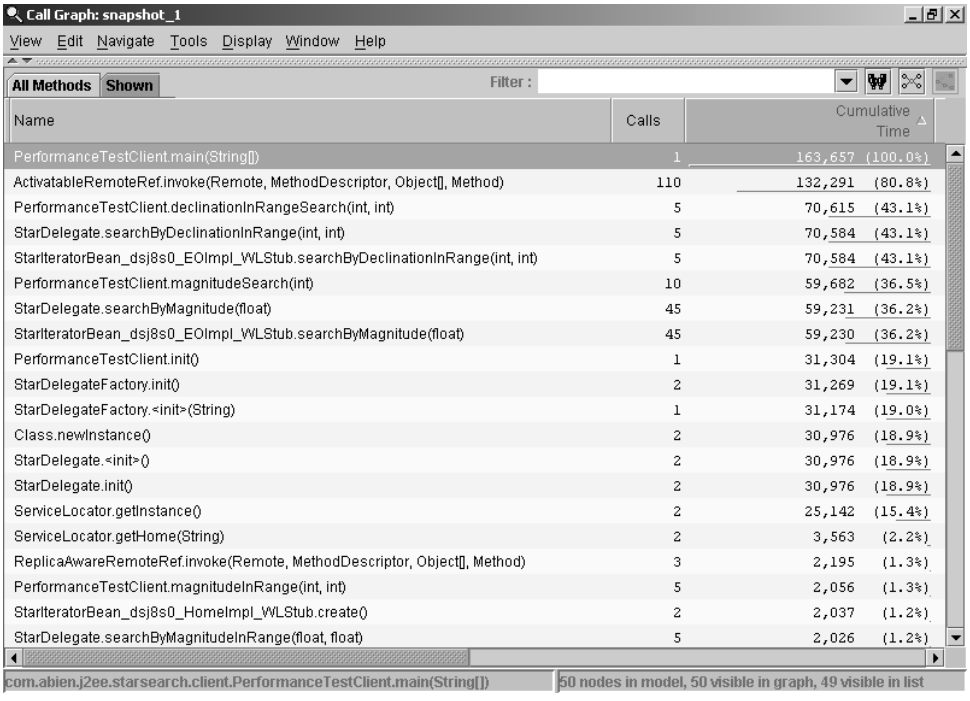


Abbildung 3.45: Die »Exclusive«-Einstellung ohne Cache des Weblogic Applikationsservers 7.0.1 (Option B)

2. **Database**: Verlässt sich auf die Locking-Mechanismen der Datenbank. Der EJB-Container delegiert die »Verantwortung« für das Locking der Datensätze zu der Datenbank. Diese Einstellung ist besonders im Clusterbetrieb des Applikationsservers zu empfehlen. Die Datenbank verwaltet hier das Locking einzelner Datensätze oder sogar ganzer Tabellen. Leider ist mit dieser Strategie lediglich die Option B möglich – die Daten müssen beim Start jeder Transaktion neu geladen und können zwischen den Transaktionen nicht gecached werden.

3. **Read Only**: Bei dieser Einstellung wird der Zustand der EntityBean nie in die Persistenzschicht zurückgeschrieben. Die Daten werden nicht mit der Persistenzschicht synchronisiert d.h. die Methode `ejbStore` muss nicht aufgerufen werden. Die Methode `ejbLoad` wird nur für das erstmalige Befüllen der EntityBean aufgerufen (Cache eingeschaltet), oder nach dem Beginn einer neuen Transaktion (Cache aus). Für die star-finder.com Anwendung eignet sich diese Einstellung hervorragend, da die Datenbank nur sehr selten geändert wird (immer dann, wenn ein neuer Stern entdeckt wird ☺).

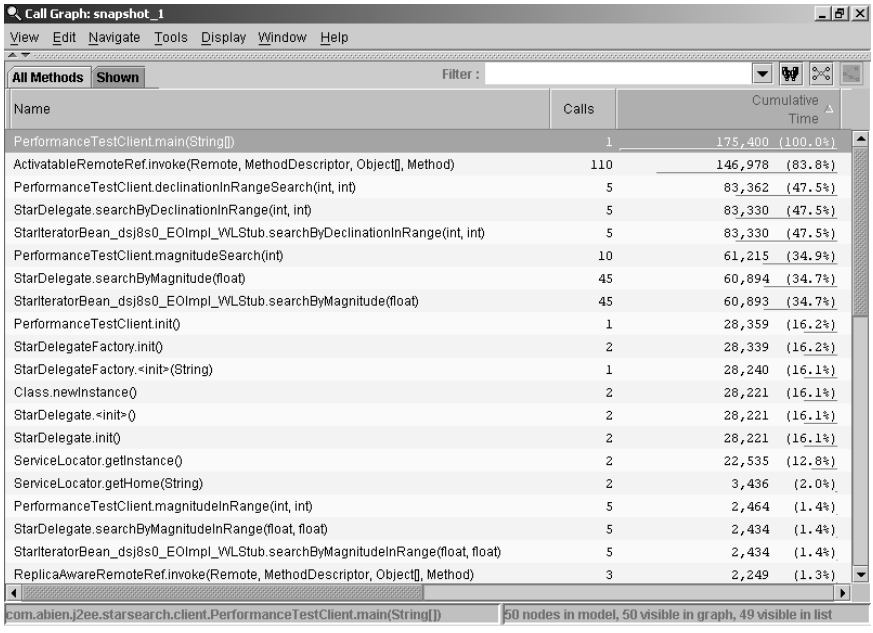


Abbildung 3.46: Die »Database«-Einstellung ohne Cache des Weblogic-Applikationsservers 7.0.1 (OptionB)

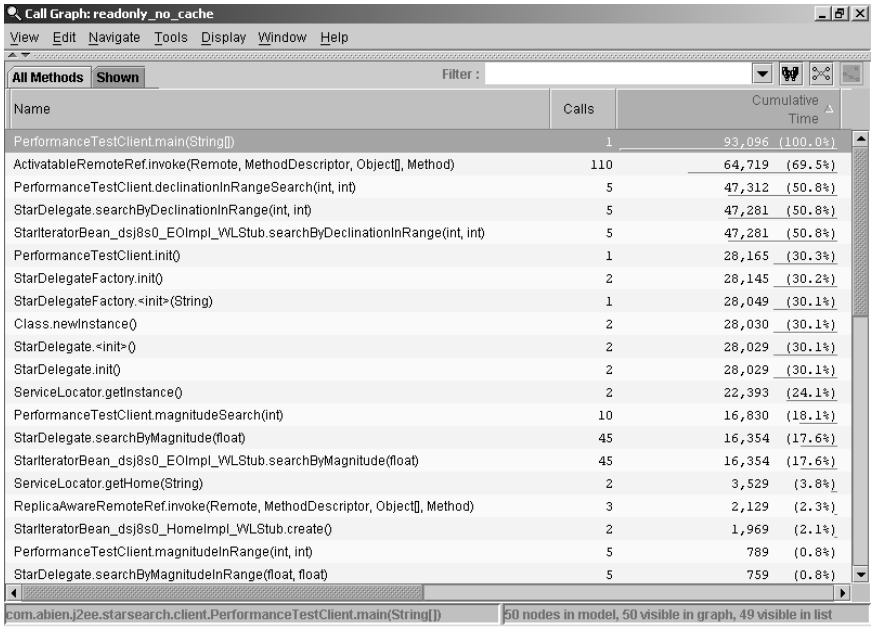


Abbildung 3.47: Die »ReadOnly«-Einstellung ohne Cache des Weblogic-Applikationsservers 7.0.1 (Option B)

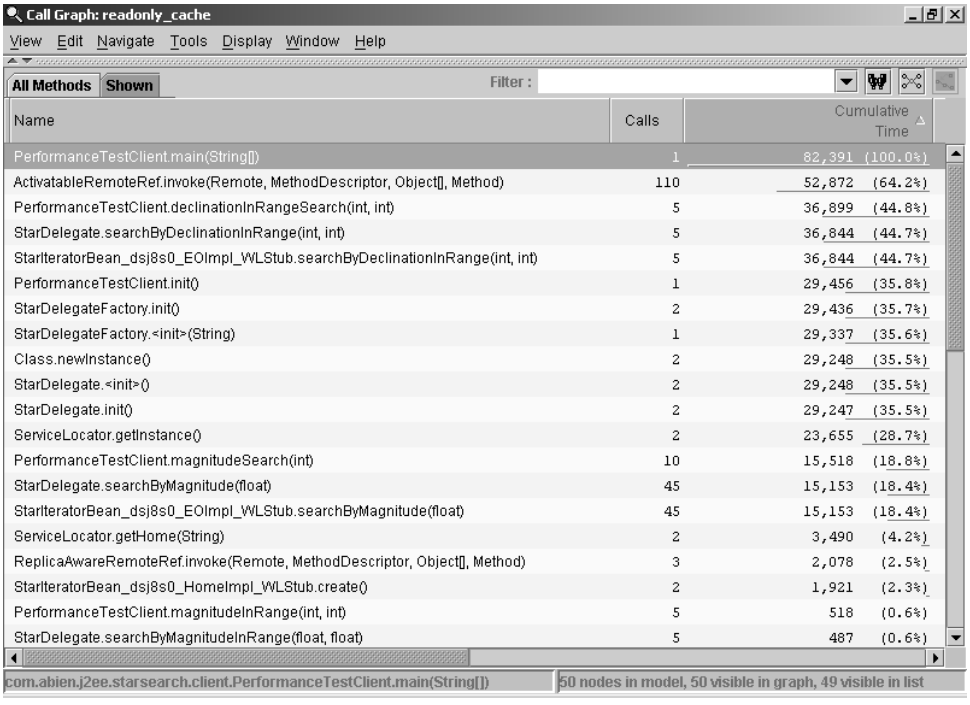


Abbildung 3.48: Die »ReadOnly«-Einstellung mit Cache des Weblogic-Applikationsservers 7.0.1 (Option A)

4. **Optimistic**: Die Bean-Instanz wird nicht gelockt. Stattdessen wird eine zusätzliche Spalte in der Tabelle angelegt, in der die aktuelle Version des Zustandes der Bean abgelegt wird. Diese Strategie entspricht den »Optimistischen Locks« (siehe: Wie implementiere ich optimistische Locks mit EJB 2.0 und der Container Managed Persistence (CMP) Strategie?) und erlaubt sowohl die maximale Performance der Anwendung als auch stellt die Integrität der Daten sicher.

Jede Option, bis auf »Database«, ist cache-fähig, d.h. der Zustand der Bean kann mehrere Transaktionen überleben und muss nicht jedes Mal aus der Datenbank mit dem Aufruf `ejbLoad` geholt werden.

Wie bereits vermutet, bietet die Read-Only-Strategie mit der Option A die beste Performance. Der EJB-Container ruft in dieser Einstellung die Methode `ejbStore` nicht auf, da man nur lesend auf die Entities zugreift. Somit wird auch die Datenbank nicht mehr geändert, was natürlich Zeit »spart«. Die Option A (Cache an) erlaubt dem EJB-Container auch die Aufrufe der Methode `ejbLoad` zu minimieren.

Die Option B der Read-Only-Strategie zwingt den Applikationsserver, die Daten nach jedem Transaktionsbeginn aus der Datenbank zu laden. Die Methode `ejbLoad` wird also sofort nach dem Beginn einer Transaktion aufgerufen. Dieser zusätzliche Aufruf sorgt hier für etwas schlechtere Performance. Falls der Applikationsserver nicht der einzige

Client der Datenbank ist, darf der Cache nicht eingeschaltet werden, da man ansonsten bereits veraltete oder nicht mehr existierende Datensätze lesen könnte.

	Exclusive	**Database**	**ReadOnly**
Mit Cache in ms (Option A)	86.196	Nicht möglich	82.391
Ohne Cache in ms (Option B)	163.657	175.400	93.096

Die zweitbeste Performance erzielte die Exclusive-Einstellung. Der EJB-Container ist hier in der Lage die Instanz der Bean zu sperren, und so für die Konsistenz der Daten zu sorgen. Wie bereits erwähnt eignet, sich diese Einstellung nur für einzelne Applikationsserver die nicht im Clusterbetrieb laufen. Da die Instanzen auch mit der Datenbank synchronisiert werden müssen, ist die Performance wesentlich schlechter, als die Performance der Read-Only-Einstellung.

Die Option A ermöglicht auch beim schreibenden Zugriff eine relativ gute Performance und stellt die Konsistenz der Daten sicher. Diese Option kann leider nur bei Applikationsservern benutzt werden, denen exklusiv eine Datenbank zur Verfügung steht. Es dürfen also keine anderen Clients parallel mit der gleichen Datenbank wie der Applikationsserver arbeiten.

Die Option B ist hier für eine noch schlechtere Performance verantwortlich. Bei dem abgeschalteten Cache bricht die Performance um fast 100% ein. Diese Einstellung kann bei einem nicht-geclusterten Applikationsserver benutzt werden, der nicht der einzige Client der Datenbank ist. Wenn man auch schreibend auf die EntityBeans zugreifen möchte, sollte die die Read-Only-Einstellung nicht verwendet werden.

3.8.4 Fazit

Bei dem Entwurf von J2EE-Anwendungen müssen die gleichen »low-Level« Technologien, wie noch zu »Mainframe-« oder »Host-Zeiten« berücksichtigt werden. Nur die Wahl der richtigen Strategie sorgt für gute Performance und die Konsistenz der Daten einer Anwendung.

3.9 Was passiert, wenn einer EntityBean die Daten aus der Datenbank zur Laufzeit geändert werden?

Oft werde ich gefragt, ob der Applikationsserver es merkt, wenn sich »seine« Daten in der Datenbank verändern. Es würde bedeuten, dass die Datenbank den Applikationsserver automatisch benachrichtigen würde, wenn sich die Daten aus dem Cache des Applikationsservers (Entity-Instanzen) geändert hätten.

3.9.1 Problemstellung

Jede EntityBean ist mit einem Primärschlüssel ausgestattet, der für die Identifizierung der Bean verantwortlich ist. Die befindet sich im »ready«-Zustand wenn sie mit dem Primärschlüssel assoziiert ist. Jederzeit können ihre Daten mit den Daten aus der Datenbank überschrieben werden. Dies geschieht mit dem Aufruf der Methode `ejbLoad`. Der gleiche Datensatz wird einmal als eine Zeile in einer Tabelle und einmal als die Entity-Instanz gehalten. Die EntityBean cached also die Daten der Datenbank. In der Datenbank werden Daten dauerhaft gespeichert, die aus Geschäftslogikaufrufen stammen und mit einem »commit«-Aufruf erfolgreich abgeschlossen wurden. Bei diesen Daten handelt es sich um den Ausgangspunkt für neue Transaktionen. Falls aber gleichzeitig mit der Datenbank und mit der EntityBean gearbeitet wird, kann es zum Integritätsverlust der Daten kommen.

3.9.2 Technischer Hintergrund

Leider »weiß« die Datenbank nicht einmal, ob ein Applikationsserver oder ein ganz »gewöhnlicher« Client mit ihr kommuniziert. Beide benutzen sogar den gleichen JDBC-Treiber! Die Datenbank hat überhaupt keine Möglichkeit den Applikationsserver zu benachrichtigen.

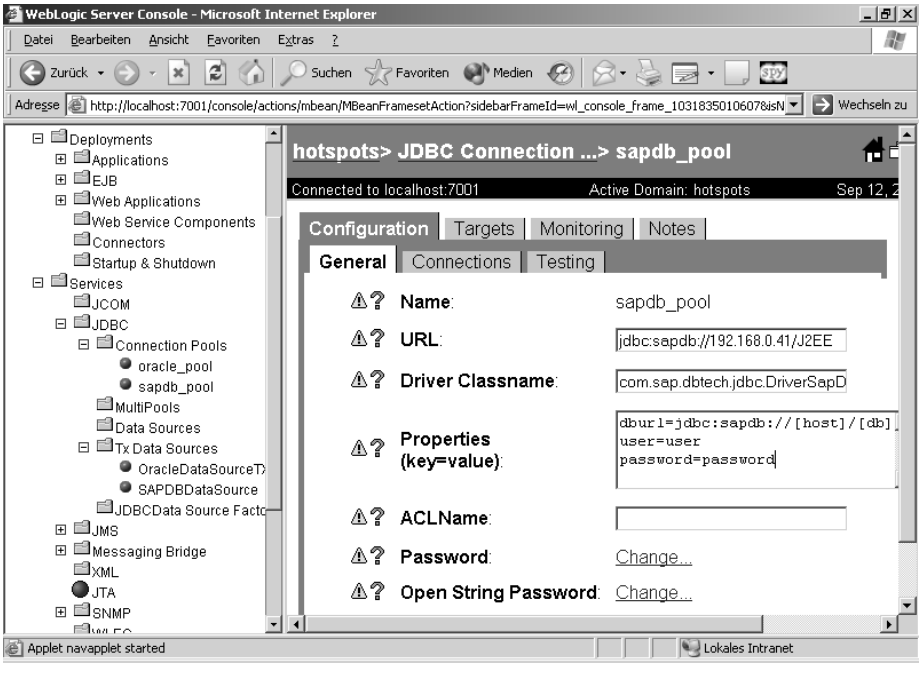


Abbildung 3.49: Die Datenbank-Einstellungen des WebLogic-Applikationsservers 7.0.1

Der Entwickler muss sich mit der Synchronisierung der Daten auseinandersetzen. Die Tatsache, ob der Applikationsserver der einzige Benutzer der Datenbank ist oder nicht, ist an dieser Stelle sehr wichtig. Falls außer dem Applikationsserver auch andere Anwendungen auf die Datenbank schreibend zugreifen, dürfen die EntityBean-Instanzen nicht gecached werden.

3.9.3 Lösung

Die Frage könnte bereits an dieser Stelle beantwortet werden: »Es passiert gar nichts«. Der Entwickler muss also dafür sorgen, dass entweder die Daten nicht geändert werden können – also gesperrt werden – oder dass die Daten nach jedem Start einer neuen Transaktion aus der Persistenzschicht geladen werden.

Die zweite Möglichkeit ist wesentlich langsamer, da hier eine SQL-Abfrage zuerst ausgeführt werden muss, um die einzelnen Attribute der Bean-Instanz mit gültigen Werten zu überschreiben.

Dem Entwickler stehen hier also die Mittel, die bereits in der Frage »Wann wird eine EntityBean mit der Datenbank synchronisiert – oder was sind »commit-options?« besprochen wurden, zur Verfügung.

3.9.4 Praxis

Auf jeden Fall sollte der Applikationsserver zunächst als einziger Benutzer der Datenbank auftreten. Somit könnten die gültigen EntityBean-Instanzen auf dem Applikationsserver gecached werden. Der EJB-Container übernimmt die Synchronisierung der Daten. Bei schreibenden Zugriffen auf die EntityBeans, wird bei dieser Einstellung zuerst der Cache (die EntityBean Instanz) und erst später die Datenbank geändert (nach dem Aufruf der Methode `ejbStore`).

Leider lässt sich diese Forderung bei den Kunden nur schwer durchsetzen. Nur selten können die Applikationsserver mit der Datenbank exklusiv arbeiten. Falls es noch andere Anwendungen gibt, muss der Cache abgeschaltet werden. Dies äußert sich durch eine schlechtere Performance, die Konsistenz der Daten bleibt weiterhin gesichert. Eine andere Möglichkeit ist die Sperrung des gerade zu bearbeitenden Datensatzes für die Dauer der Transaktion. Die Daten könnten aus einer relationalen Datenbank mit der SQL-Abfrage `"select … for update"` geholt werden. Die Datenbank sperrt dann die einzelnen Datensätze automatisch. Anderen Anwendungen ist es nicht möglich, den geraden von dem Applikationsserver bearbeiteten Datensatz zu lesen. Alle Interessenten müssen so lange warten, bis die Transaktion mit einem »commit« oder »rollback« abgeschlossen wird. Falls die Datensatzsperre nicht erwünscht ist (pessimistische Locks), können jederzeit auch optimistische Locks eingeführt werden (siehe: »Wie implementiere ich optimistische Locks mit EJB 2.0 und der Container Managed Persistence (CMP) Strategie?«).

Bei dieser Strategie geht man zunächst davon aus, dass kein anderer an dem gerade benötigten Datensatz interessiert ist. Jeder gelesene Datensatz erhält eine Versionsnummer. Bei jeder Änderung der Datenbank wird die Version in der Datenbank erhöht. Falls beim Zurückschreiben der modifizierten Daten die Version nicht mit der Datenbankversion übereinstimmt, wird eine Exception geworfen. Der Benutzer muss sich den aktuellen Datensatz noch einmal holen und hoffen, dass diesmal alles gut geht …

3.10 Was bedeutet *reentrant* bei den Deployment-Einstellungen der EntityBeans?

Beim Deployment von EntityBeans wird man mit der »Reentrant«-Einstellung konfrontiert. Obwohl diese Einstellung ein fester Bestandteil des Deployment Descriptors ist, ist sie in der Entwicklergemeinde noch ziemlich unbekannt.

```
<entity>
  <display-name>StarBean</display-name>
  <ejb-name>StarBean</ejb-name>
  <home>com.abien.j2ee.starsearch.ejb.StarHome</home>
  <remote>com.abien.j2ee.starsearch.ejb.Star</remote>
  <local-home>com.abien.j2ee.starsearch.ejb.StarLocalHome</local-home>
  <local>com.abien.j2ee.starsearch.ejb.StarLocal</local>
  <ejb-class>com.abien.j2ee.starsearch.ejb.StarBean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>com.abien.j2ee.starsearch.ejb.StarPK</prim-key-class>
      <reentrant>False/True</reentrant>
      <abstract-schema-name>StarBase</abstract-schema-name>
      <cmp-field>
        <description>no description</description>
        <field-name>signOfDeclination</field-name>
      </cmp-field>
      <cmp-field>
...
</entity>
```

Da der Tag **<reentrant>** für das Deployment einer Entity notwendig ist, wird man beim Erzeugen des Deployment Descriptors nach dieser Einstellung »gefragt«. Meistens wird diese Einstellung von dem Deployer einfach »übersehen«. Dies kann jedoch für die Anwendung fatale Folgen haben.

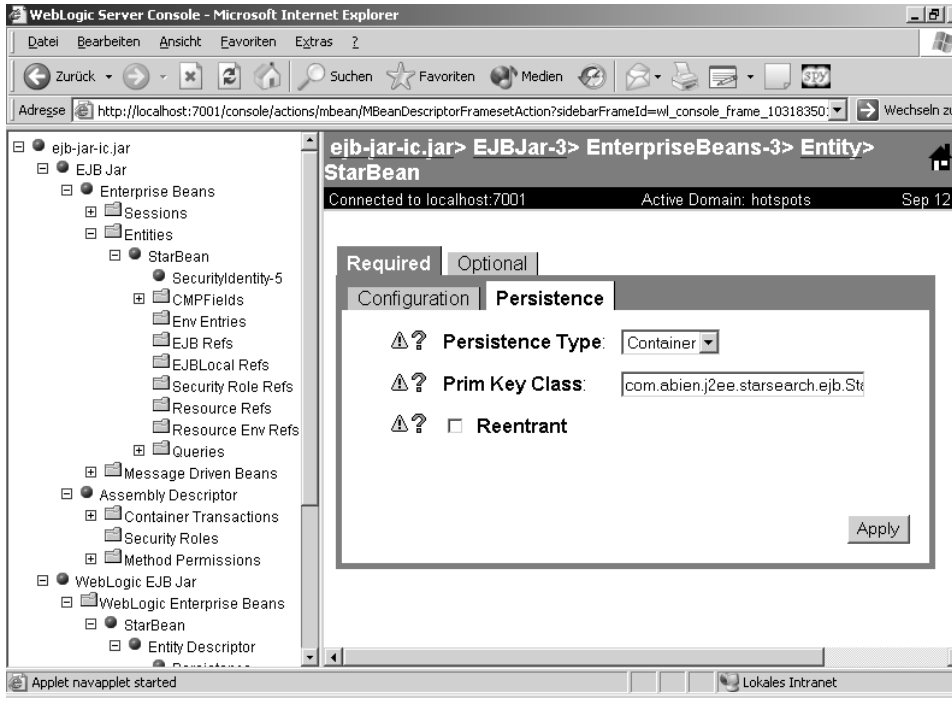


Abbildung 3.50: Die reentrant-Einstellung einer EntityBean

3.10.1 Problemstellung

Auf eine EntityBean können mehrere Clients in unterschiedlichen Transaktionskontexten gleichzeitig zugreifen. Das bedeutet auch, dass Clients nebenläufig oder multithreaded mit der gleichen EntityBean-Instanz arbeiten können.

Ferner kann es passieren, dass sog. Loopbacks für die Abbildung der Geschäftslogik erforderlich sind. Ein Client könnte die Methode A der ersten Entity aufrufen. Diese benötigt eine andere Entität für die Abbildung der Geschäftslogik und ruft die Methode B innerhalb der gleichen Transaktion auf. Falls die zweite Entity wieder die erste innerhalb der gleichen Transaktion aufrufen sollte, wird hier von einem Loopback gesprochen.

Die Einstellung »reentrant« bestimmt, ob diese Aktionen von dem EJB-Container erlaubt werden oder nicht. Falls man die EntityBean mit der Einstellung `reentrant=false` deployed und die beschriebenen Regeln verletzt, wirft der Container eine `javax.ejb.RemoteException`, falls über die Remote-Interfaces mit der EJB kommuniziert wurde und die `javax.ejb.EJBException`, falls über die Local-Interfaces auf die EJB zugegriffen wurde.

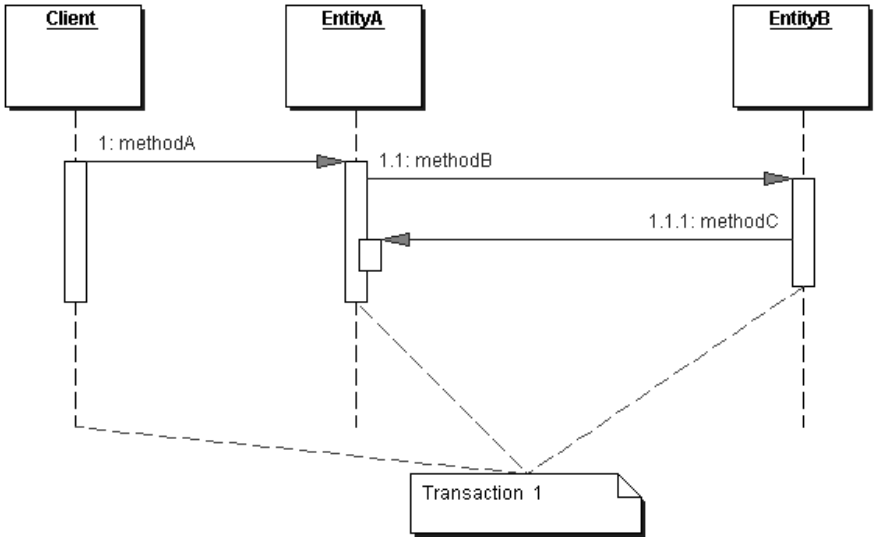


Abbildung 3.51: Ein Loopback zwischen zwei Entity-Instanzen

Bei der Einstellung `reentrant=true` überwacht der EJB-Container die Loopbacks nicht. Es können also ungewollt »Endlosschleifen« entstehen. Der Entwickler muss selbst zwischen gewollten und ungewollten Loopbacks entscheiden. »Reentrant« bedeutet an dieser Stelle, dass alle Aufrufe dieser Bean-Instanz innerhalb der gleichen Transaktion wieder »zurückkehren« können. Die gleiche Instanz kann also innerhalb der gleichen Transaktion »wieder betreten« werden.

3.10.2 Empfehlung

Eine EntityBean sollte mit der Einstellung `reentrant=true` deployed werden, wenn man die Loopbacks oder Callbacks benötigt. Mit dieser Einstellung wird die Schutzfunktion des EJB-Containers abgeschaltet, sodass ungewollte Callbacks entstehen können. Ein ungewolltes Callback kann auch Endlosschleifen verursachen, was wiederum zu einem `java.lang.StackOverflowError` führen kann. Ferner muss auch der Entwickler dafür sorgen, dass die EntityBean »threadsafe« ist.

Fall ein Error im EJB-Container geworfen wird, kann dies die Zerstörung der EJB bedeuten. Im Normalfall reicht die Einstellung `reentrant=false` vollkommen aus. Der Container überwacht hier das Wiederbetreten der Bean und wirft dementsprechend `java.rmi.RemoteException` oder `javax.ejb.EJBException` aus, falls diese Regel verletzt wird. Für die Serialisierung der ankommenden Aufrufe sorgt der EJB-Container.

3.11 Was bringt die Benutzung von Local-Interfaces für die Performance der Anwendung?

Eine der wichtigsten Neuerungen der EJB 2.0 Spezifikation war die Einführung der Local-Interfaces. Es handelt es sich hier um eine zusätzliche Möglichkeit für die Kommunikation mit den EJB-Komponenten. Eine EJB-Komponente kann mit vier, anstatt wie bisher mit zwei Interfaces (zwei RemoteHome, Remote und LocalHome, Local) deployed werden. Dies bedeutet natürlich auch einen Mehraufwand für den Entwickler, Assembler und Deployer. Aus diesem Grund wird oft vor größeren Projekten zunächst evaluiert, ob Local-Interfaces benutzt werden sollten oder nicht. Da diese Frage natürlich auch die J2EE-Entwickler beschäftigt, gehört sie zu den meistgestellten und ich möchte sie an dieser Stelle beantworten.

3.11.1 Problemstellung

Bei den EJBs handelt es sich um Komponenten, die auf eine natürliche Art und Weise Remote-Schnittstellen bereitstellen. Bei dem Deployment-Prozess wird die notwendige Infrastruktur automatisch generiert. Dabei handelt es sich um die Stubs und Skeletons, die für den transparenten Zugriff eines Clients auf die EJB-Komponente verantwortlich sind. Mit dem Tausch der Stubs und Skeletons kann auch das Netzwerkprotokoll ausgetauscht werden.

Bei der Kommunikation über die Remote-Schnittstelle der Beans ist mit einem gewissen Mehraufwand zu rechnen. Insbesondere ist die Serialisierung der Parameter und Rückgabewerte um Faktoren langsamer als bei einem lokalen Methodenaufruf.

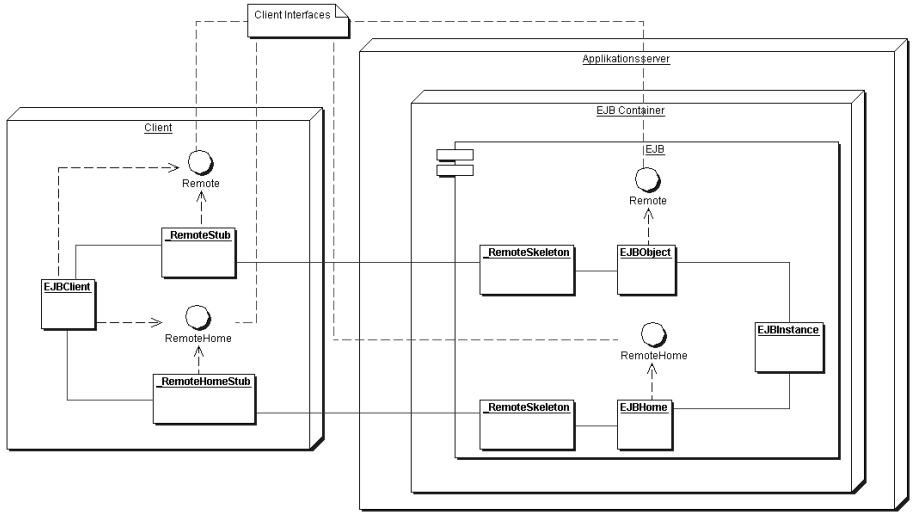


Abbildung 3.52: Der »konventionelle« Zugriff des Clients auf eine EJB-Komponente

Bei der EJB 1.1 Spezifikation war man gezwungen, über die Remote-Schnittstelle auch auf eine lokale Komponente zuzugreifen. Diese Art der Kommunikation war sehr uneffektiv, da sehr viel CPU-Zeit für den durch die Stubs und Skeletons verursachten Mehraufwand benötigt wurde.

Allerdings »erkannten« die guten Applikationsserver, ob sich zwei EJBs im gleichen EJB-Container auf der gleichen JVM befanden oder nicht. Falls die Kommunikationspartner sich auf der gleichen JVM befanden, war die Kommunikation über die Stubs und Skeletons unnötig und wurde einfach weggelassen. Mit diesem »Algorithmus« war es möglich, die Gesamtperformance der Anwendung deutlich zu erhöhen.

Diese Optimierungen waren leider kein J2EE-Standard, sodass man sich auf diese bei der Entwicklung von J2EE-Anwendungen nicht verlassen konnte. Manche Applikationsserver haben den Zugriff optimiert – andere nicht. Es konnte also passieren, dass eine J2EE-konforme Anwendung auf einem Applikationsserver sehr gute Performance erzielt, auf einem anderen unbrauchbar langsam ist.

Für Abhilfe sorgte die EJB 2.0 Spezifikation mit der Einführung der Local-Views oder der Local-Interfaces. Mit dieser Erweiterung ist es möglich, sich nur für den lokalen Zugriff zu entscheiden. Voraussetzung dafür ist das Deployment der beiden EJBs in den gleichen EJB-Container. Da bei dieser Kommunikation die Stubs und Skeletons fehlen, können sich die Kommunikationspartner nur gegenseitig aufrufen, wenn sie in der gleichen JVM laufen. Es ist also eine »klassische« Referenz einer EJB für den Zugriff über die Local-Interfaces notwendig.

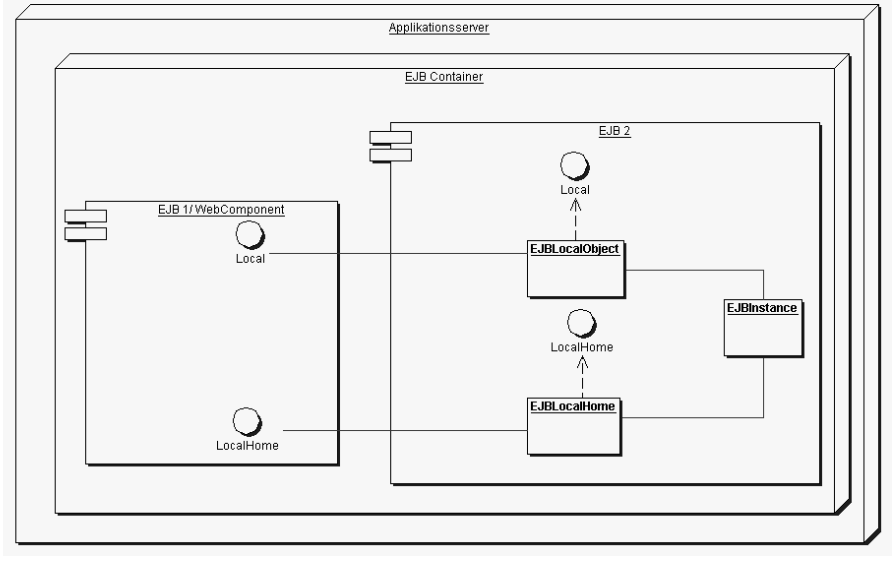


Abbildung 3.53: Der Zugriff auf eine EJB-Komponente über die Local-Interfaces

Bei dem Zugriff über die Local-Interfaces auf eine EJB-Instanz wird jedoch nicht direkt mit ihr gearbeitet, sondern es werden zuerst die EJBLocalObject bzw. die EJBLocalHome Instanzen angesprochen.

Diese entsprechen dem Proxy-Pattern, d.h. sie sind der Stellvertreter der Bean-Instanz. Egal, ob die Kommunikation über Remote- oder Local-Interfaces stattfindet, ein direkter Zugriff auf die EJB-Instanz ist weiterhin nicht möglich.

3.11.2 Performance

Für die Performance-Messung wurde ein »Standalone«-Client benutzt, der über die Remote-Interfaces auf die EJB-Facade zugreift. Der gleiche Client wurde für die Untersuchung der Local- und Remote-Performance verwendet. Für den Client ist die Benutzung der Local- oder Remote-Interfaces zwischen der Facade und der endgültigen EJB vollkommen transparent. Der folgende Client wurde innerhalb von JProbe 4.0-Profiler als eine Anwendung gestartet, um die Performance zu messen.

```
public class PerformanceTestClient {

      public static void main(String[] args) {
      int loops = Integer.parseInt(args[0]);
      try {
      Context context = new InitialContext();
      Object ref = context.lookup("timerfacade");
TimerFacadeHome home =(TimerFacadeHome) PortableRemoteObject.narrow(ref,
TimerFacadeHome.class);
            TimerFacade timer = home.create();
            for (int i = 0; i < loops; i++) {
                  timer.getTime();
            }
            timer.remove();
            } catch (RemoteException e) {
            } catch (NamingException e) {
            } catch (CreateException e) {
            } catch (RemoveException e) {
            }
      }
}
```

Ähnlich wie auch der Client, ist die Implementierung der `TimerBean` von den Local- oder Remote-Interfaces unabhängig. Der gleiche Quellcode kann also sowohl mit Local- als auch mit Remote-Interfaces deployed werden.

```
public class TimerBean implements SessionBean {
      public void setSessionContext(SessionContext arg0){}
      public void ejbRemove(){}
      public void ejbActivate() {      }
      public void ejbPassivate(){      }
```

```
        public Date getTime(){return new Date();      }
        public void ejbCreate(){}
}
```

Remote-Interfaces

Erst beim Deployment wird entschieden, mit welchen Interfaces die Implementierung ausgestattet wird. Zuerst wird die `TimerBean` lediglich mit den Remote-Interfaces deployed. Dementsprechend muss auch das `<ejb-ref>`-Element angepasst werden.

```
<session>
      <display-name>TimerBean</display-name>
      <ejb-name>TimerBean</ejb-name>
      <home>com.hotspots.local_vs_remote.timer.TimerHome</home>
      <remote>com.hotspots.local_vs_remote.timer.Timer</remote>
    <ejb-class>com.hotspots.local_vs_remote.timer.TimerBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
      ...
    </session>
    <session>
      <display-name>TimerFacadeBean</display-name>
      <ejb-name>TimerFacadeBean</ejb-name>
      <home>com.hotspots.local_vs_remote.facade.TimerFacadeHome</home>
      <remote>com.hotspots.local_vs_remote.facade.TimerFacade</remote>
      <ejb-class>com.hotspots…facade.remote.TimerFacadeBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
      <ejb-ref>
        <ejb-ref-name>ejb/Timer</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>com.hotspots.local_vs_remote.timer.TimerHome</home>
        <remote>com.hotspots.local_vs_remote.timer.Timer</remote>
        <ejb-link>ejb-jar-ic.jar#TimerBean</ejb-link>
      </ejb-ref>
      ...
     </session>
```

Die `TimerFacadeBean` benutzt den Namen `ejb/Timer` in ihrem Quellcode, um eine »Referenz« auf die `TimerBean` über die Remote-Interfaces zu erhalten. Es handelt sich hier um einen Schlüssel, der von dem EJB-Container bzw. Applikationsserver aufgelöst wird. Da hier zuerst mit den Remote-Interfaces gearbeitet wird, muss das `PortableRemoteObject` für das Narrowing benutzt werden. Es handelt sich dabei um die Konvertierung eines CORBA-Objektes in ein JAVA-Objekt. Da für die Remote-Kommunikation RMI-IIOP benutzt wird, ist diese Maßnahme erforderlich.

```
public class TimerFacadeBean implements SessionBean {
      private TimerHome      home      = null;
      private Timer          timer     = null;
```

```
        private Context        context   = null;
        public void setSessionContext(SessionContext arg0){
        try {
              this.context = new InitialContext();
        Object ref    = this.context.lookup("java:comp/env/ejb/Timer");
        this.home = (TimerHome)PortableRemoteObject.narrow(ref,TimerHome.class);
        this.timer   =    home.create();
          } catch (RemoteException e) {
        } catch (NamingException e) {
    } catch (CreateException e) {
        }
        }
        public void ejbRemove(){}
        public void ejbActivate() {      }
        public void ejbPassivate(){      }

        //Businesslogic
        public String getTime(){
        try {
              return this.timer.getTime().toString();
        } catch (RemoteException e) {
        throw new EJBException("Exception occured: " + e);
        }
      }
        public void ejbCreate(){}
    }
```

Bei der logischen Betrachtung stellt man fest, dass die Facade aus dem EJB-Container »rausgehen« muss, um auf die Remote-Interfaces zugreifen zu können. Obwohl beide sich in einem Container befinden und auf einer JVM laufen, greift die `TimerFacadeBean` auf die `TimerBean` über das RMI-IIOP Protokoll zu. Die `TimerBean` könnte also auch auf einem anderen, entfernten Applikationsserver laufen.

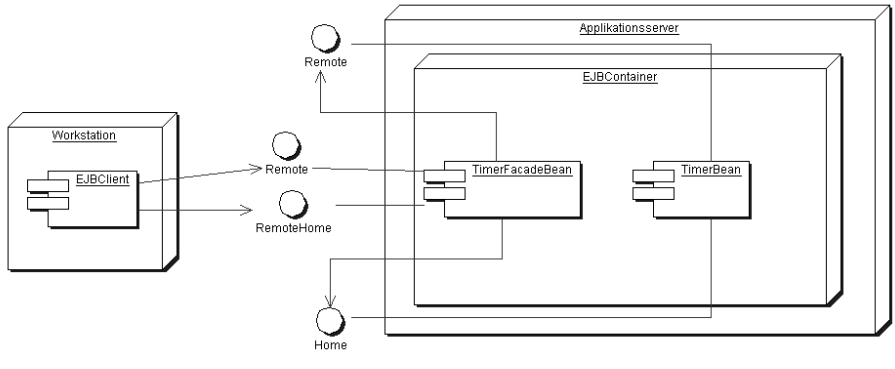


Abbildung 3.54: Zugriff über die Remote-Interfaces der EJB

Die 10.000 Aufrufe dauern hier auf dem WebLogic Applikationsserver 7.0 101,069 sec. Interessanterweise braucht das `InitialContext` für die Initialisierung ganze 26,754 sec. Diese Zeit könnte man getrost von der Gesamtzeit abziehen, da die `InitialContext`-Instanz sich gut cachen lässt. Man könnte hier mit dem Service-Locator-Pattern arbeiten, um sowohl das `InitialContext` als auch die Home-Interfaces, Queues oder Topics effizienter verwalten zu können.

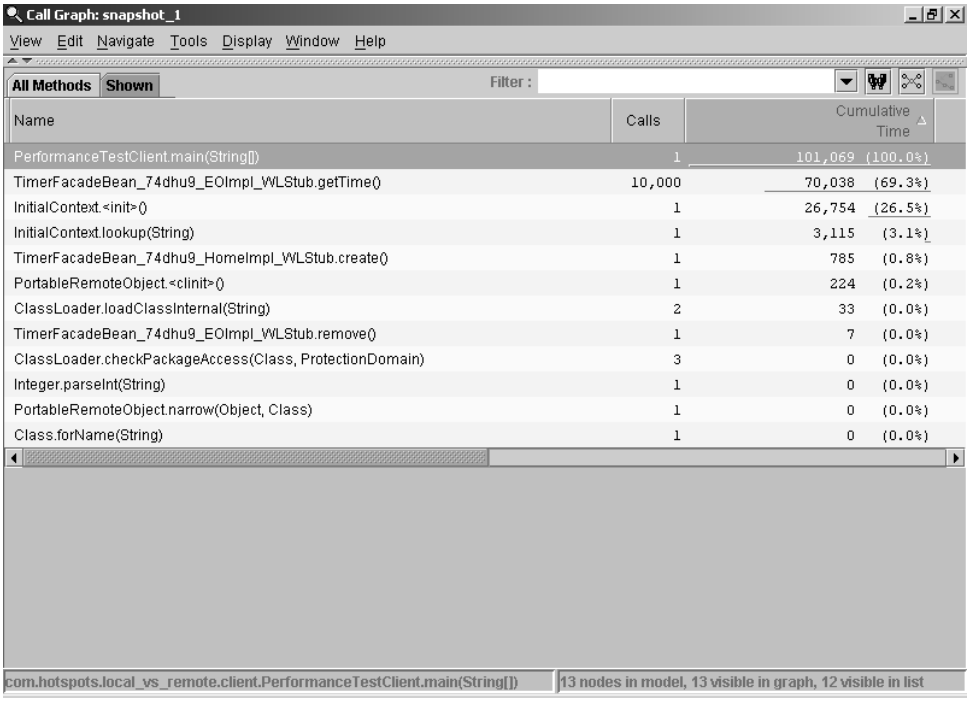


Abbildung 3.55: Performance des Zugriffs über die Remote-Interfaces

Local-Interfaces

Für die Umsetzung der »lokalen Sicht« einer EJB ist neben der Implementierung des `LocalHome`-Interfaces und des `Local`-Interfaces noch die Änderung des Deployment Descriptors notwendig. Bei dem `LocalHome`-Interface handelt es sich um eine »Local-Factory« – es darf also nur das Local- und nicht das Remote-Interface erzeugen. Ferner erbt dieses Interface nicht von dem Interface `javax.ejb.EJBHome`, sondern vom Interface `javax.ejb.EJBLocalHome`.

```
public interface TimerLocalHome extends EJBLocalHome {
      public TimerLocal create() throws CreateException;
}
```

Das Local-Interface erbt wiederum von dem Interface `javax.ejb.EJBLocalObject`. Weder die Methoden des Local-Home, noch des Local-Interfaces werfen `java.rmi.RemoteException`. Alle anderen Exceptions wie `javax.ejb.CreateException`, `javax.ejb.FinderException` usw. werden wie gewohnt geworfen.

```
public interface TimerLocal extends EJBLocalObject {
      public Date getTime();
}
```

Bei der Anpassung des Deployment Descriptors bleibt die `TimerFacadeBean` nicht völlig verschont. Der Schlüssel des `ejb/TimerLocal` muss hier das Local-Home-Interface liefern. Der Eintrag muss dementsprechend im Deploymentdescriptor geändert werden. Es werden hier an dieser Stelle die `TimerLocalHome` und `TimerLocal` Interfaces eingetragen. Natürlich müssen diese auch mitdeployed werden, damit die Implementierung der Interfaces (Proxy) generiert werden kann.

```
<session>
   <display-name>TimerFacadeBean</display-name>
   <ejb-name>TimerFacadeBean</ejb-name>
   <home>com.hotspots.local_vs_remote.facade.TimerFacadeHome</home>
   <remote>com.hotspots.local_vs_remote.facade.TimerFacade</remote>
   <ejb-class>com.hotspots….facade.local.TimerFacadeBean</ejb-class>
   <session-type>Stateless</session-type>
   <transaction-type>Bean</transaction-type>
   <ejb-local-ref>
     <ejb-ref-name>ejb/TimerLocal</ejb-ref-name>
     <ejb-ref-type>Session</ejb-ref-type>
     <local-home>com.hotspots…..timer.TimerLocalHome</local-home>
     <local>com.hotspots.local_vs_remote.timer.TimerLocal</local>
     <ejb-link>ejb-jar-ic.jar#TimerBean</ejb-link>
   </ejb-local-ref>

…

 </session>
 <session>
   <display-name>TimerBean</display-name>
   <ejb-name>TimerBean</ejb-name>
   <local-home>com.hotspots…..timer.TimerLocalHome</local-home>
   <local>com.hotspots.local_vs_remote.timer.TimerLocal</local>
   <ejb-class>com…..timer.TimerBean</ejb-class>
   <session-type>Stateless</session-type>
   <transaction-type>Bean</transaction-type>
 </session>
```

Die Implementierung der `TimerFacadeBean` musste allerdings angepasst werden. Diese kennt »nur« noch die Local-Interfaces. Zunächst wird wie erwartet nach dem Local-Home-Interface gesucht. Dieses erzeugt das Interface `TimerLocal`. Das »Narrowing« mit

dem javax.rmi.PortableRemoteObject kann an dieser Stelle weggelassen werden, da nicht über ein IIOP-Objektprotokoll auf die Proxies zugegriffen wird, sondern per einfacher Referenz. Man erhält also eine »normale« Referenz auf das EJBLocalHome-Proxy zurück. Da die TimerBean nur mit den Local-Interfaces deployed wird, werden auch keine Stubs und Skeletons generiert. Auf sie kann nur noch lokal zugegriffen werden. Eine Anwendung, die sich außerhalb der JVM und somit auch außerhalb des EJB-Containers befindet, ist nicht in der Lage, mit der TimerBean zu kommunizieren. Sie ist also gezwungen, die TimerFacadeBean für die Kommunikation mit der TimerBean zu benutzen.

```
public class TimerFacadeBean implements SessionBean {
      private TimerLocalHome localHome = null;
      private TimerLocal     local     = null;
      private Context        context   = null;

      public void setSessionContext(SessionContext arg0){
      try {
      this.context = new InitialContext();
      this.localHome = (TimerLocalHome)this.context.lookup("java:comp/env/ejb/
TimerLocal");
      this.local = localHome.create();
      } catch (NamingException e) {
      } catch (CreateException e) {}
      }
      public void ejbRemove(){}
      public void ejbActivate() {}
      public void ejbPassivate(){}
      public String getTime(){
            return this.local.getTime().toString();
}
      public void ejbCreate(){}

}
```

Da die TimerFacadeBean auf die TimerBean über ihre Local-Interfaces zugreift, ist in der Methode getTime das Fangen der RemoteException nicht mehr notwendig. Das Fehlen dieser Exception erinnert uns daran, dass bei dem Zugriff auf die TimerBean die Stubs und Skeletons nicht notwendig sind. Die RemoteException ist mit einer »Kommunikationsexception« vergleichbar. Bei der lokalen Kommunikation kann allerdings nichts schief gehen. Wenn man bereits die Referenz auf das benötigte Objekt bekommen hat, kann man es auch aufrufen.

Der Zugriff über die Local-Interfaces dauerte 92,339 sec. Es war also nur unwesentlich schneller als der Zugriff über die Remote-Interfaces.

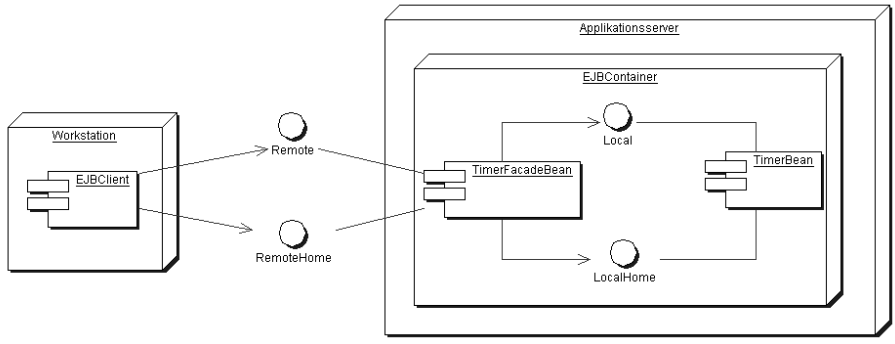


Abbildung 3.56: Zugriff über die Local-Interfaces der EJB

Call Graph: snapshot_1

View Edit Navigate Tools Display Window Help

All Methods Shown Filter :

Name	Calls	Cumulative Time
PerformanceTestClient.main(String[])	1	101,069 (100.0%)
TimerFacadeBean_74dhu9_EOImpl_WLStub.getTime()	10,000	70,038 (69.3%)
InitialContext.<init>()	1	26,754 (26.5%)
InitialContext.lookup(String)	1	3,115 (3.1%)
TimerFacadeBean_74dhu9_HomeImpl_WLStub.create()	1	785 (0.8%)
PortableRemoteObject.<clinit>()	1	224 (0.2%)
ClassLoader.loadClassInternal(String)	2	33 (0.0%)
TimerFacadeBean_74dhu9_EOImpl_WLStub.remove()	1	7 (0.0%)
ClassLoader.checkPackageAccess(Class, ProtectionDomain)	3	0 (0.0%)
Integer.parseInt(String)	1	0 (0.0%)
PortableRemoteObject.narrow(Object, Class)	1	0 (0.0%)
Class.forName(String)	1	0 (0.0%)

com.hotspots.local_vs_remote.client.PerformanceTestClient.main(String[]) 13 nodes in model, 13 visible in graph, 12 visible in list

Abbildung 3.57: Die Performance des Zugriffs über die Local-Interfaces

Es könnte also sein, dass der Applikationsserver tatsächlich erkennt, ob die beiden Kommunikaitonspartner sich in der gleichen JVM befinden oder nicht. Bei einer »lokalen« Beziehung könnte er dafür sorgen, dass die Kommunikation ohne die Stubs und Skeletons stattfindet. Wie bereits erwähnt, handelt es sich hier um eine herstellerabhängige Optimierung, die nicht in der J2EE-Spezifikation definiert wurde.

	Local	Remote
WLS 7.0.1	92,339	101,069
J2SDKEE 1.3.1 RI	251,640	250,422

Die Performance wurde noch einmal mit der Referenzimplementierung J2EE 1.3 RI gemessen. Auch hier kam ich jedoch zum gleichen Ergebnis. Die Performance ist bei beiden Varianten nahezu gleich.

3.11.3 Lösung

Bei der Studie des Resultats könnte man den Nutzen der Local-Interfaces in Frage stellen. Die Performance der beiden Varianten ist ziemlich ähnlich, es lohnt sich also nicht, nur wegen der besseren Performance die bestehende Anwendung anzupassen, oder?

Auf jeden Fall ist der Applikationsserver gezwungen beide Komponenten auf der gleichen JVM zu »halten«. Bei dem Zugriff über die Remote-Interfaces auf eine EJB ist dies nicht der Fall. In einer geclusterten Umgebung werden die EJB-Container auf unterschiedliche Maschinen verteilt. Das bedeutet auch, dass die gleiche Anwendung auf mehreren JVMs gleichzeitig laufen kann.

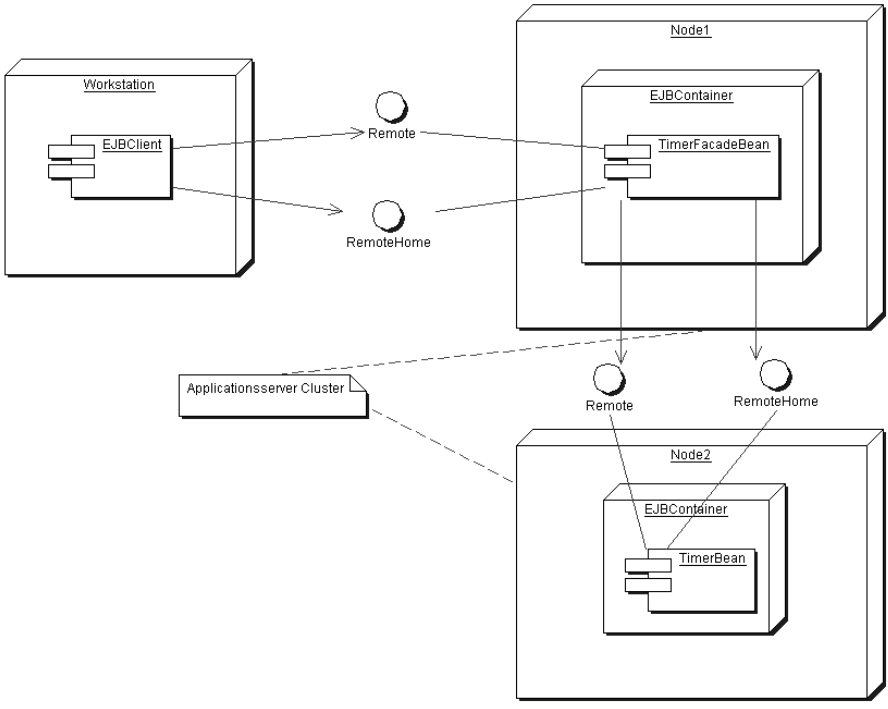


Abbildung 3.58: Das »Cluster-Problem«

In so einer Umgebung kann es passieren, dass die Performance der Remote-Lösung dramatisch einbricht.

Für den Client wäre der Vorfall allerdings vollkommen transparent. Der Applikationsserver könnte sich für Verlegung der `TimerBean`-Komponente auf den anderen Knoten entscheiden. In dem Fall kann der Zugriff nicht mehr optimiert werden – der EJB-Container ist gezwungen mit Hilfe der Stubs und Skeletons mit der entfernten Instanz zu kommunizieren. Je höher die gegenseitige Abhängigkeit innerhalb der Komponente (also je höher die Kohäsion), desto mehr könnte die Performance der Anwendung darunter leiden.

Bei der Benutzung der Local-Interfaces bleibt die Performance auf jeden Fall konstant, da der EJB-Container die ganze Komponente auf einem Knoten ausführen muss.

3.11.4 Fazit

Leider wurde mir viel zu oft die falsche Frage gestellt. Bei der Benutzung der Local-Interfaces geht es nicht so sehr um die Verbesserung der Performance, vielmehr ist es eine Entscheidung des Architekten, die Komponenten mit einer zusätzlichen Sichtbarkeit (Scope) auszustatten. Ähnlich wie bei der `package`-weiten, der `public` oder `private` Sichtbarkeit, erhält man hier eine zusätzliche Möglichkeit, das Innenleben einer Komponente zu kapseln.

Da es dem Client nicht möglich ist, von außerhalb des Containers auf die Local-Interfaces zuzugreifen, bleibt dies nur den ausgesuchten Komponenten gewährt. Mit dem Ansatz lassen sich unabhängige EJBs mit Hilfe einer Facade zu einer Komponente zusammenfassen. Nur diese Facade wird dann mit einem Remote-Interface ausgestattet. Der Client ist nicht in der Lage, auf die Subkomponenten oder Subsysteme zuzugreifen – somit ist er auch von ihnen total entkoppelt. Die einzige Koppelung des Clients zu der Komponente ist das Remote-Interface der Facade.

Obwohl ich mir viel Mühe gemacht habe die Frage zu beantworten, Sie viel Zeit verbrachten meine Ausführungen zu verstehen, spielt die Performance in dem Fall überhaupt keine Rolle. Schließlich fragt Sie auch keiner (mehr), ob der Aufruf einer `private`-Methode genauso schnell ist wie der Aufruf einer `public` oder »`package`« Methode.

3.12 Wo soll der Zustand des Benutzers abgelegt werden (Präsentations- oder Geschäftslogikschicht)?

Das HTTP-Protokoll ist zustandslos. Das bringt enorme Vorteile mit sich. Die Performance der Webserver lässt sich beliebig skalieren, die Ausfallsicherheit kann auch beliebig gesteigert werden. Leider sind nur wenige Web-Anwendungen zustandslos. Aus dem Grund muss der Zustand aufwändig »emuliert« werden. Wie geht man mit

der Tatsache um? Wo hält man am besten den Zustand einer J2EE-Anwendung, in der Präsentationsschicht oder in der Geschäftslogikschicht?

3.12.1 Problemstellung

Der J2EE-Entwickler ist gezwungen, die benutzerabhängigen Daten auf dem Applikationsserver zu speichern. Dies wird durch Anforderungen des Kunden und somit durch die Anwendungsfälle verursacht. Die meisten Applikationen »wissen«, mit welchem Anwender gerade kommuniziert wird. Dieses »Wissen« lässt sich nur mit dem Aufbau einer Session realisieren. Es müssen jedes Mal Informationen zwischen dem Client und dem Server übertragen werden und diese eine Zuordnung eines Requests zu einem Benutzer ermöglichen. So lange diese Informationen zwischen einem Client und dem Server übertragen werden, so lange bleibt auch die Session erhalten. In dieser Session können dann bequem zusätzliche Daten eines Benutzers abgelegt werden – oft wird hier von dem »Konversationsgedächtnis« gesprochen. Ein Beispiel für diese Vorgehensweise wäre ein Warenkorb eines Internetshops. Der Benutzer kann bequem seine ausgewählten Artikel in dem Warenkorb ablegen, nach dem Verlassen der Seite wird die Session und somit auch der Inhalt des Warenkorbs, zerstört.

An dieser Stelle stellt sich natürlich die Frage, ob diese »Daten« im Web-Container oder EJB-Container abgelegt werden sollen.

3.12.2 Technischer Hintergrund

Die einfachste Möglichkeit ist die Ablage der Daten in der `javax.servlet.http.HttpSession`. Die `HttpSession` ermöglicht die Ablage der Daten mit Hilfe der Schlüssel-Werte-Paaren. Die `HttpSession` lässt sich mit einer `java.util.HashTable` oder einer `java.util.HashMap` vergleichen – es handelt sich um eine tabellenartige Struktur. Pro Benutzersession existiert auch eine Instanz der `HttpSession` im Web-Container. Für die Zuordnung des Benutzers zu der Session sorgt der Web-Container. Mit Hilfe der SessionID wird die Zuordnung überhaupt ermöglicht. Bei dieser SessionID handelt es sich um einen Schlüssel (»jsessionid«), der zwischen dem Browser und dem Web-Container übertragen wird.

Der Web-Container versucht zunächst den Wert des Schlüssels mit Hilfe der Cookies zu übertragen. Bei einem Cookie handelt es sich um eine Datei oder einen Speicherbereich, der von dem Web-Browser des Clients verwaltet wird. Der Browser stellt den Inhalt der Cookies dem Applikationsserver zur Verfügung. Der EJB-Container ist in der Lage, auf den Inhalt zuzugreifen und assoziiert die »richtige« `HttpSession` mit dem `HttpServletRequest` des Benutzers.

Abbildung 3.59: Das Konversationsgedächtnis des Web-Containers

Es kann hier bequem auf die aktuelle `HttpSession`-Instanz und somit auch auf seine Daten des Benutzers zugegriffen werden. Mit dieser »Technologie« wäre die Ablage eines Warenkorbes des aktuellen Benutzers möglich.

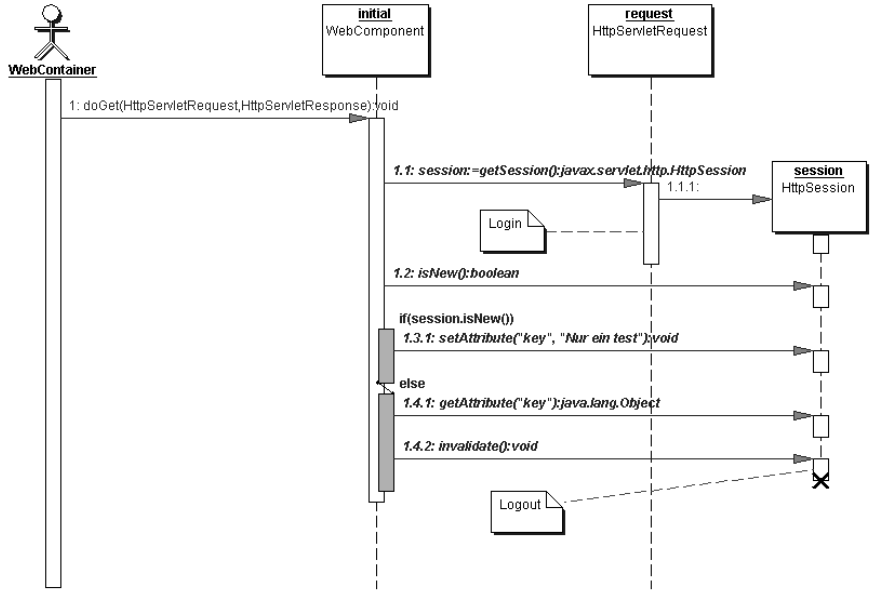


Abbildung 3.60: Die Arbeit mit der HttpSession

Die Ablage der Session-Daten ist nicht nur in der Präsentationsschicht, sondern auch in der Geschäftslogikschicht möglich. Zu diesem Zweck eignen sich hervorragend die `Stateful SessionBeans`. Jedem Benutzer bzw. jeder Session wird eine Instanz der Stateful SessionBean zugeordnet. Nach dem Beenden der Session wird die Instanz sofort zerstört. Der Benutzer steht in einer 1:1-Beziehung zu einer Stateful SessionBean Instanz.

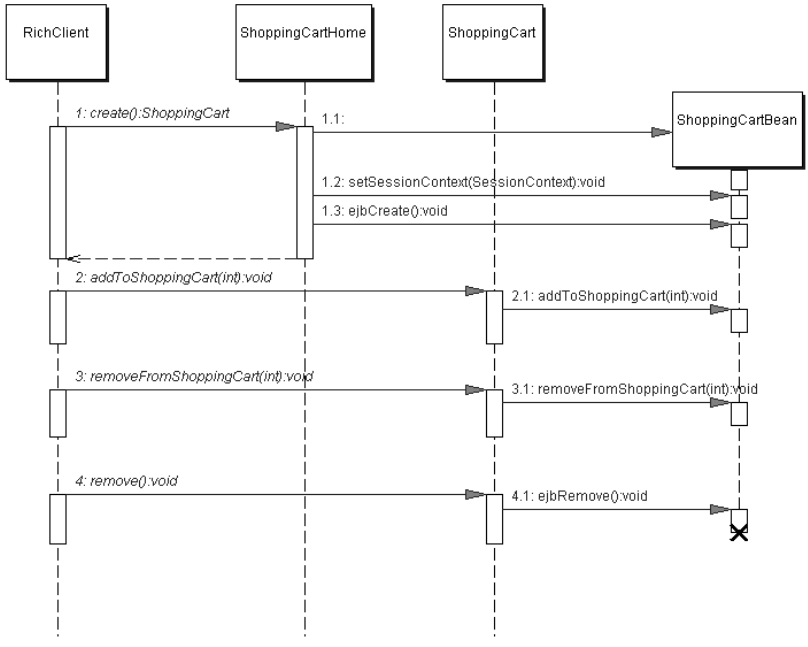


Abbildung 3.61: Arbeiten mit einer Stateful SessionBean (EJBObject- und HomeObject- Instanzen wurden nicht berücksichtigt)

Dabei könnte es sich hier bereits um einen »Warenkorb« handeln, sodass aus der objektorientierten Sicht dieser Ansatz eleganter ist. Der Entwickler kann hier einfach eine Komponente erzeugen, benutzen und dann zerstören – für die Aufrechterhaltung der Session sorgt bereits die Komponente. Die Sessionverwaltung ist in diesem Fall für den Entwickler vollkommen transparent. Neben der Transparenz stehen dem Entwickler auch andere, zusätzliche Services wie Transaktionen, Security oder der Zugriff von außerhalb des Containers automatisch zur Verfügung. Ferner bleibt die Geschäftslogik dort, wo sie auch hingehört – nämlich in der Geschäftslogikschicht.

3.12.3 Lösung

Die Antwort auf diese vieldiskutierte Frage ist relativ einfach. Wenn für die Abbildung der Geschäftslogik des Use Cases Transaktionen oder andere Dienste des EJB-Containers benötigt werden, sollte man sich für die Stateful SessionBeans entscheiden. Die

Entscheidung wird einem weiter erleichtert, wenn das Geschäftsobjekt selbst zustandsbehaftet ist. In dem Fall wäre die Benutzung der `HttpSession` schlichtweg »unnatürlich«, da hier noch eine zusätzliche Komponente (z.B. SessionManager) für die transparente Session-Verwaltung sorgen müsste. Ferner stünde diese Implementierung lediglich den Web-Anwendungen zur Verfügung - die Geschäftslogikschicht wäre weiterhin zustandslos. Ein »Rich-Client« müsste die Zustandsverwaltung ein weiteres Mal implementieren und sich auch um die Transaktionen kümmern. Transaktionen sollten allerdings für die Präsentationsschicht weiterhin absolut transparent bleiben, ein Client ist lediglich an der Geschäftslogik und nicht an der Technologie interessiert.

Andererseits sollte man die `HttpSession` bevorzugen, wenn »Hilfsdaten« der Präsentationsschicht abgelegt werden müssen. Wenn man dem Internetbenutzer die Historie der bereits besuchten Seiten anzeigen möchte, sollte diese in der `HttpSession` abgelegt werden. Diese Geschäftslogik ist nämlich »präsentationsschicht-gebunden« und deswegen nicht so gut wieder verwendbar. Jeder Rich-Client könnte diese Funktionalität auf einfachste Art und Weise nachbilden – was auch den Applikationsserver entlasten könnte. Ferner hat dieser Zustand nichts mit der Geschäftslogik zu tun, vielmehr mit der technischen Realisierung der Präsentationsschicht. Aus diesem Grund sollte die Verwaltung solcher Zustände in der Präsentationsschicht gehandhabt werden.

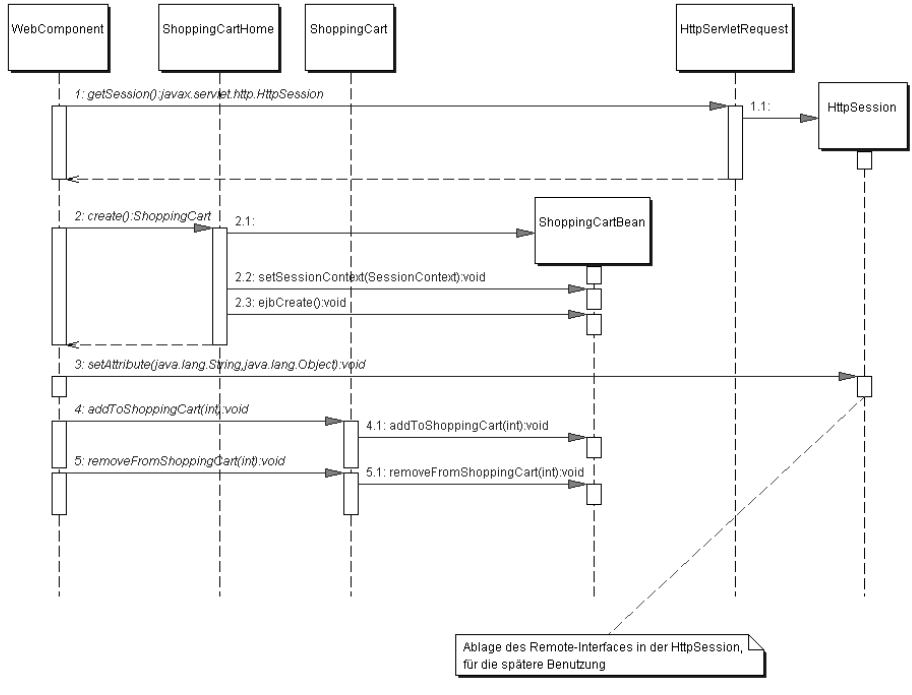


Abbildung 3.62: Das Zusammenspiel der HttpSession und einer Stateful SessionBean

Leider kann auf die `HttpSession` nicht ganz verzichtet werden. Der Zustand einer Stateful SessionBean ist mit der Instanz des Remote-Interfaces assoziert. Der Benutzer der Stateful SessionBean, muss sich die Referenz auf das Remote-Interface »merken«, um auf den aktuellen Zustand der Stateful SessionBean zugreifen zu können. Eine Webkomponente ist somit gezwungen, das Remote-Interface in der `HttpSession` für die spätere Benutzung abzulegen. Falls sich hier um eine Persistente-Session handelt, müssen anstatt von Remote- oder Local-Interfaces ihre Handles in der `HttpSession` abgelegt werden. Persistente Sessions werden oft in verschiedenen Cluster-Implementierungen verwendet. Dabei wird bei jeder Änderung der Session der Inhalt in einer zentralen Datenbank abgelegt.

Bei dem nächsten Request kann wieder aus der bestehenden `HttpSession`-Instanz das Remote-Interface herausgeholt werden. Mit diesem Verweis auf die Stateful SessionBean ist es nun möglich, auf die Session-Daten zuzugreifen.

3.12.4 Praxis

In der Produktionsumgebung werden Cluster verwendet, um die Ausfallsicherheit des Systems zu erhöhen. Dabei handelt es sich um identisch (redundant) eingerichteten Serverknoten. Auf jedem Knoten werden die gleichen Komponenten installiert. Der Client greift zunächst auf den Dispatcher zu. Der Dispatcher wählt, abhängig von der eingerichteten Strategie, den entsprechenden Knoten aus. Jeder weiterer Request kann auf einem anderen Knoten landen. Diese Tatsache spielt bei den zustandslosen Komponenten keine Rolle, in unserem Fall sind alle Komponenten zustandsbehaftet. Es muss sichergestellt werden, dass den Client auf beiden Knoten der gleiche Zustand »erwartet«. Diese Bedingung muss sowohl für den Web-Container, als auch für den EJB-Container erfüllt werden.

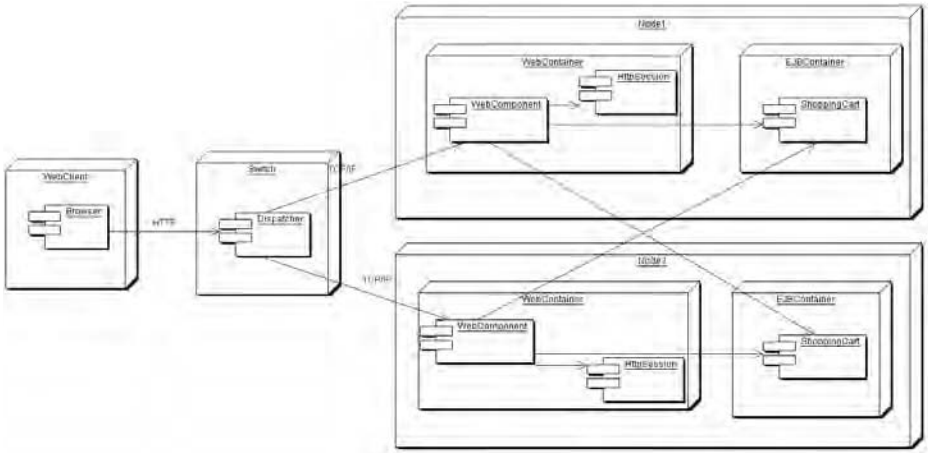


Abbildung 3.63: Die HttpSession und Stateful SessionBean im Clusterbetrieb

Die Art und Weise wie es zu dem Datenabgleich im Cluster kommt, wird nicht in der J2EE-Spezifikation definiert. Es handelt sich hier im Grunde um eine herstellerspezifische Erweiterung, die unterschiedlich gelöst werden kann. In der Praxis haben sich jedoch die folgenden Strategien in verschiedensten Variationen bewährt:

- Die Ablage der Daten im gemeinsamen Speicher: dabei werden die Objektinstanzen meistens in einer zentralen Datenbank abgelegt. Bei der Benutzung dieser Strategie kann es zu Performance-Problemen kommen.
- Die Replizierung der Daten, meistens mit Hilfe von UDP: bei der Benutzung dieser Strategie kommt es meistens zu keinen Performance-Verlusten. Die Zustände können hier auch asynchron repliziert werden. Es kann jedoch bei einem Absturz des Applikationsservers zu Datenverlusten kommen.

Die meisten Applikationsserver-Hersteller weisen darauf hin, dass die Menge der Daten, die in der `HttpSession` abgelegt werden, klein gehalten werden sollte. Dabei wird meist eine Größenordnung von 2-4 kB angegeben.

Die Replizierung der Stateful SessionBean kann auch auf unterschiedliche Art und Weise erfolgen. Ähnlich wie bei der `HttpSession`, können auch hier die Daten entweder in einer gemeinsamen Datenbank abgelegt oder der Zustand automatisch auf alle Clusterknoten verteilt werden. Dabei wird oft von »In-Memory-Replication« gesprochen.

Aus diesem Grund ist es sehr wichtig, sich mit der Cluster-Technologie des Applikationsservers auseinanderzusetzen, um zu entscheiden, wo denn endgültig der Zustand abgelegt wird. Noch wichtiger ist es jedoch, die Architektur so flexibel zu gestalten, dass man auf die Besonderheiten verschiedener Hersteller flexibel reagieren kann.

3.13 Was ist in einer J2EE-Umgebung nicht erlaubt?

3.13.1 Problemstellung

Bei der Entwicklung von J2EE-Anwendungen stößt man zwangsläufig auf die in der EJB-Spezifikation beschriebenen Restriktionen. Es ist vieles nicht erlaubt, was bei Eigenentwicklungen schnell und einfach implementiert werden kann. In einer J2EE-Umgebung lassen sich diese Features nur mit großem Aufwand oder einigen Gedanken über die Vorgehensweise implementieren. Was ist im EJB-Container nicht erlaubt?

3.13.2 Restriktionen bei der Entwicklung von EJBs (EJB 2.1 Spezifikation)

»An enterprise Bean must not use read/write static fields. Using read-only static fields is allowed. Therefore, it is recommended that all static fields in the enterprise bean class be declared as final.«

Eine EJB-Komponente sollte nur lesend auf die `static` Attribute zugreifen. Bei der Deklaration der Attribute innerhalb einer EJB sollte das Schlüsselwort `final` verwendet werden.

Diese Restriktion lässt sich mit der Tatsache rechtfertigen, dass ein EJB-Container auch in einer geclusterten Umgebung ablaufen könnte. Dies würde bedeuten, dass eine EJB-Komponente auf mehreren JVMs ablaufen kann. Static-Attribute sind jedoch JVM-bezogen – im Cluster würde der schreibende Zugriff auf die static-Attribute nicht funktionieren. Oft wird an dieser Stelle auch gefragt, ob das Singleton-Pattern im Cluster erlaubt ist. Die Verwendung eines Singletons ist nur dann erlaubt, wenn man auf diesen lediglich im »read only«-Modus zugreift. Die Benutzung des Service-Locator-Patterns ist absolut legitim, da hier Ressourcen aus dem JNDI-Context gelesen werden.

Grunsätzlich kann ich jedoch empfehlen, den Einsatz von static Feldern und Methoden in einer J2EE-Umgebung zu minimieren. Ein Objekt, das statisch referenziert wird, kann nicht »garbage collected« werden …

»An enterprise Bean must not use thread synchronization primitives to synchronize execution of multiple instances.«

Der Modifier `synchronized` sollte von einer EnterpriseBean nicht verwendet werden. Die Synchronisierung der Threads funktioniert nur innerhalb einer JVM. Eine Enterprise Bean könnte auf mehreren JVMs »gleichzeitig« ablaufen.

»An enterprise Bean must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard.«

Nur die wenigsten Applikationsserver haben eine Grafikkarte – die grafische Ausgabe der Benutzerschnittstelle ist an dieser Stelle unmöglich. Die Benutzung des AWT/Swing Frameworks könnte den EJB-Container und somit auch den Applikationsserver blockieren. Ferner lässt sich nur schwer bestimmen, auf welchem Clusterknoten die GUI aufgehen wird …

»An enterprise bean must not use the java.io package to attempt to access files and directories in the file system.«

Bei der Benutzung des `java.io`-Packages wird indirekt auf die Ressourcen des Betriebssystems zugegriffen. Ferner muss man sich hier um die Synchronisierung/Sperre der Dateien oder Streams bei nebenläufigem Zugriff kümmern. Außerdem müsste ein Abgleich zwischen den Dateien in den Clusterknoten stattfinden. Diese Anforderung lässt sich allerdings lediglich mit dem Einsatz zusätzlicher Mittel wie z.B. NFS realisieren. Sowohl die Synchronisierung als auch die Replizierung der Inhalte ist auf dem Applikationsserver problematisch und lässt sich nur schwer betriebssystem- bzw. applikationsserverunabhängig implementieren. Aus diesen Gründen ist die direkte Verwendung des `java.io` Packages von EJBs nicht erlaubt.

»An enterprise bean must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicast.«

Eine EJB-Komponente sollte nicht die Aufgaben eine »SocketServers« übernehmen. Ein `java.net.SocketServer` blockiert den aktuellen Thread, bis sich ein `java.net.Socket` mit dem Server verbindet. Die Threads sollten jedoch in einer J2EE-Umgebung von dem EJB-Container bzw. von dem Applikationsserver und nicht von der Komponente selbst verwaltet werden. Die Benutzung von `java.net.Socket` als einen Client ist jedoch erlaubt. Viele J2EE-konforme Technologien wie JDBC-Treiber, JavaMail, JMS, RMI, und RMI-IIOP basieren auf den Sockets.

»The enterprise bean must not attempt to query a class to obtain information about the declared members that are not otherwise accessible to the enterprise bean because of the security rules of the Java language. The enterprise bean must not attempt to use the Reflection API to access information that the security rules of the Java programming language make unavailable.«

»The enterprise bean must not attempt to gain access to packages and classes that the usual rules of the Java programming language make unavailable to the enterprise bean.«

Eine EJB sollte nicht den Reflection-Mechanismus für den Zugriff auf private, protected oder package-weiter Ressourcen wie z.B. Attribute oder Methoden verwenden. Der eingebaute Java-Sicherheitsmechanismus sollte nicht von den EJBs ausgeschaltet werden. Eine EJB-Komponente sollte die Fähigkeiten des `java.lang.reflect.AccessibleObject` nicht verwenden, um Zugriff auf z.B. private Attribute mit Hilfe der Reflection-API zu erhalten. Diese Funktionalität könnte die Sicherheit eines EJB-Containers bzw. Applikationsservers gefährden.

»The enterprise bean must not attempt to create a class loader; obtain the current class loader; set the context class loader; set security manager; create a new security manager; stop the JVM; or change the input, output, and error streams.«

Die Einstellungen des SecurityManagers, Context Class Loaders, und die Funktionalitäten der Klasse `java.lang.System` sind größtenteils für die Umsetzung der Einschränkungen und Verhalten eines J2EE-Applikationsservers verantwortlich. Die Änderung dieser Einstellungen könnte die Sicherheit und die Stabilität der Anwendung und sogar des ganzen Applikationsservers gefährden. Ferner könnte eine J2EE-Anwendung mit dieser Einstellung die komplette Ablaufumgebung des ganzen Applikationsservers verändern. Die »anderen« Anwendungen verlassen sich jedoch auf die in der J2EE-Spezifikation definierte Umgebung…

»The enterprise bean must not attempt to set the socket factory used by ServerSocket, Socket, or the stream handler factory used by URL.«

Der Applikationsserver benutzt für die Herstellung der Ablaufumgebung die von ihm gesetzten Ressourcen-Factories wie z.B. ServerSocket oder Stream Handler Factory. Die »vorsätzliche« Änderung dieser Einstellung würde auch die Ablaufumgebung des Applikationsservers verändern. Diese wurde jedoch in der J2EE-Spezifikation genau definiert…

»The enterprise bean must not attempt to manage threads. The enterprise bean must not attempt to start, stop, suspend, or resume a thread; or to change a thread's priority or name. The enterprise bean must not attempt to manage thread groups.«

Die meisten Applikationsserver versuchen teure Ressourcen so effizient wie möglich zu verwalten. Die Threads gehören zu dieser Ressourcenkategorie. Das Starten und Stoppen von Threads ist besonders ineffizient. Aus diesem Grund werden die Threads in Threadpools verwaltet. Dabei werden die Threads niemals beendet, vielmehr werden einem »arbeitslosen« Thread andere Aufgaben übergeben.

Die Verwaltung dieser Threads von einer Komponente aus, könnte zu Deadlocks oder Instabilitäten des Applikationsservers führen. Ferner könnten auch versehentlich Systemdienste des Applikationsserver blockiert werden.

»The enterprise bean must not attempt to directly read or write a file descriptor.«

Bei dem `java.io.FileDescriptor` handelt es sich um einen Verweis auf die rechner- und somit auch betriebsystemabhängige Ressource wie Datei, Socket oder eine andere »Byte-Senke«. Der FileDescriptor wird von dem `java.io.FileInputStream` und `java.io.FileOutputStream` direkt verwendet. Sowohl auf den direkten Zugriff auf die Betriebssystemressourcen als auch auf die direkte Arbeit mit Dateien sollte in einer J2EE-Umgebung verzichtet werden. Ansonsten kann die Plattform- und Produktunabhängigkeit der Anwendung nicht gewährleistet werden.

»The enterprise bean must not attempt to obtain the security policy information for a particular code source.«

Diese Funktionalität ist für die Implementierung der Geschäftslogik der Beans irrelevant. Dieses »Wissen« kann nur für Angriffe der EJBs auf die Security des Applikationsserver oder des EJB-Containers verwendet werden.

»The enterprise bean must not attempt to load a native library.«

Das Laden von nativen Bibliotheken kann die Stabilität der JVM gefährden. Ein Applikationsserver läuft auf der gleichen JVM-Instanz wie die J2EE-Anwendung. Ein Absturz der Bibliothek könnte den Absturz des ganzen Applikationsservers mit allen deployten Anwendungen bedeuten. Falls der Zugriff auf eine native Bibliothek für die Abbildung der Geschäftslogik unverzichtbar ist, könnte man den nativen Kern der Anwendung in einer getrennten JVM starten und auf diese »remote« zugreifen.

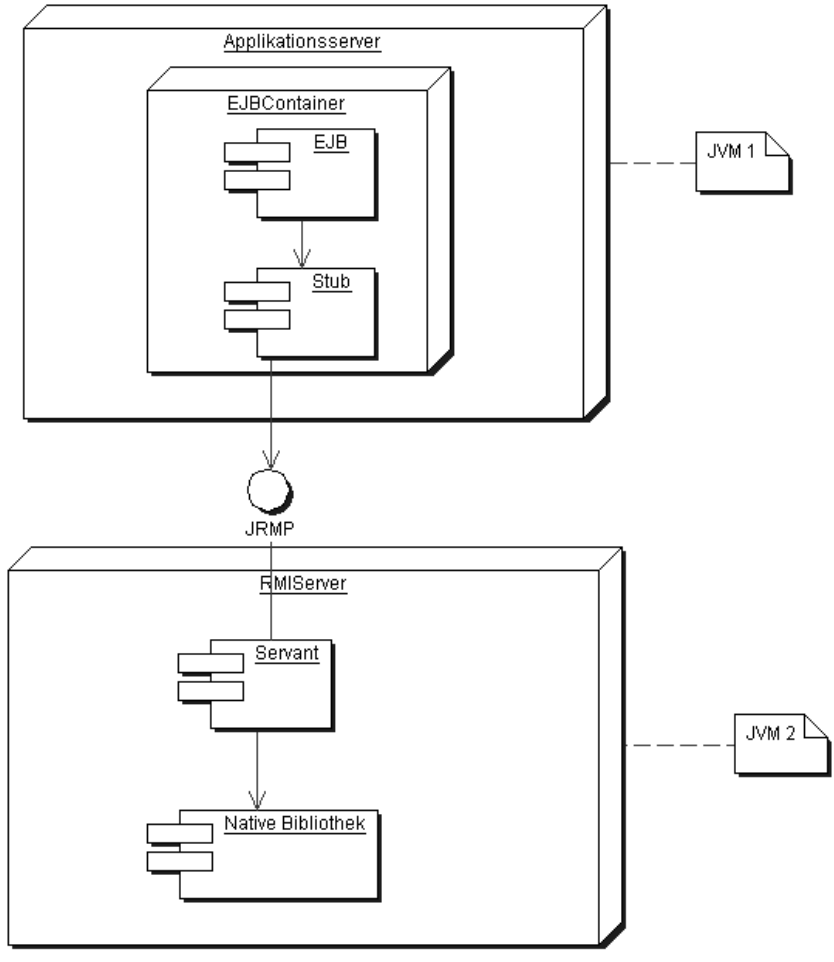


Abbildung 3.64: Die Isolation des nativen Kerns in eine getrennte JVM

Ein weiterer Grund für diese Vorgehensweise ist die Betriebssystemabhängigkeit des nativen Kernes. Mit der räumlichen Trennung der nativen Bibliothek von dem Applikationsserver könnte die native Bibliothek auf einem anderen Betriebsystem ablaufen, als der Applikationsserver.

»The enterprise bean must not attempt to define a class in a package.«

Das Erzeugen von neuen Klasseninstanzen mit Hilfe eines `ClassLoader`s ist nicht erlaubt. Die Methoden `defineClass` dürfen für die Erzeugung von neuen Klassen nicht innerhalb von einer EJB verwendet werden. Grundsätzlich liegen die Instanzierung und das Auflösen von Klassen in dem Zuständigkeitsbereich des Applikationsservers.

»The enterprise bean must not attempt to access or modify the security configuration objects (Policy, Security, Provider, Signer and Identity).«

Die Konfiguration der Sicherheit wurde genau in der J2EE-Spezifikation definiert. Für die Umsetzung der Einstellungen sorgt der Applikationsserver. Eine nachträgliche Änderung der Konfiguration könnte sowohl die Portabilität als auch die Sicherheit der Anwendung gefährden.

»The enterprise bean must not attempt to use the subclass and object substitution features of the Java Serialization Protocol«

Der Aufruf der Methode `replaceObject(Object obj)` ermöglicht den Tausch eines nichtserialisierbaren Objektes mit einem serialisierbaren Objekt. Dies bleibt jedoch nur den »trusted« Streams vorbehalten. Der Tausch der Objekte bei der Serialisierung ermöglicht auch den Tausch des privaten Zustands des Objekts, was auch als sicherheitsbedenklich eingestuft werden sollte.

»The enterprise bean must not attempt to pass this as an argument or method result. The enterprise bean must pass the result of SessionContext.getEJBObject(), SessionContext.getEJBLocalObject(), EntityContext.getEJBObject(), orEntityContext.getEJBLocalObject() instead.«

Ein Client ist niemals in der Lage, mit einer EJB-Instanz direkt zu kommunizieren. Vielmehr werden die beim Deployment generierte Proxies – also die Implementierungen der Local- und Remote-Interfaces - für die Kommunikation verwendet. Die Rückgabe der Referenz auf die Bean selbst (`this`) könnte diese Einschränkungen hier verletzen.

```
public class TestBean implements SessionBean {
    private SessionContext ctx;

    public void setSessionContext(SessionContext context){
        ctx = context;
    }
      //nicht erlaubt!!
    public TestBean getTestBean(){
            return this;
    }
//…
}
```

Die `EJBObject` und `HomeObject` sind für die Bereitstellung zusätzlicher Dienste wie Transaktionen, Security usw. verantwortlich. Die direkte Kommunikation mit der Bean-Instanz würde die zusätzliche Funktionalität des EJB-Containers an dieser Stelle ausschalten. Ferner könnten mehrere Clients mit einer EJB-Instanz unkoordiniert kommunizieren, was die Konsistenz der Daten gefährden könnte.

3.13.3 Die Security-Einstellungen des Applikationsservers

Für die Umsetzung der hier aufgeführten Restriktionen sorgt die Security-Policy des Applikationsservers. Ein EJB 2.1-konformer EJB-Container sollte mit der folgenden Security-Policy-Konfiguration ausgestattet werden, um die hier besprochenen Restriktionen umsetzen zu können:

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

Falls ein EJB-Container oder Applikationsserver mit einer geänderten Security-Policy betrieben wird, könnte es zu Kompatibilitätsproblemen kommen. Eine J2EE-Anwendung wäre nur auf einem Applikationsserver lauffähig.

Die Verletzung der hier aufgeführten Regeln führt zu einer `java.lang.SecurityException`. Bei dieser Exception handelt es sich um eine `java.lang.RuntimeException`. Das Auftreten einer `RuntimeException` führt zu der Zerstörung der EJB-Instanz.

3.14 Was muss bei der Implementierung eines Logging/Tracing-Mechanismus berücksichtigt werden?

3.14.1 Problemstellung

Die Erläuterung der Frage 1.13 hat gerade gezeigt, dass die Benutzung der Funktionalität des Packages `java.io` und der Arbeit mit dem `java.io.FileDescriptor` sowie direkter Zugriff auf Dateien nicht erlaubt sind. Trotzdem ist ein Tracing- oder Logging-Mechanismus für den Betrieb einer Anwendung sehr wichtig. Da das Logging-Format noch nicht standardisiert wurde, sehen die Ausgaben unterschiedlicher Applikationsserver natürlich auch unterschiedlich aus. Das Logformat wurde in der J2EE-Spezifikation nicht definiert – somit ist dieses Herstellerverhalten absolut legitim und auch

J2EE-konform. Allerdings möchte man natürlich, dass die Log-Einträge im eigenen Format geschrieben werden. Ferner könnten eigene Log-Dateien zusätzliche Informationen wie z.B. Zeitstempel, Vorgangsnummer oder eigenen Severities beinhalten. In einer J2SE-Umgebung, hat man mit der Implementierung eines Logging-Mechanismus keine Probleme. Das Lesen und Schreiben von Dateien kann auf einfachste Art und Weise implementiert werden. Eine J2EE-Umgebung ist wesentlich restriktiver – die direkte Benutzung von betriebssystemnahen Ressourcen ist hier sogar verboten. Lässt sich ein eigenes Logging-System für die J2EE-Welt implementieren?

3.14.2 Technischer Hintergrund

Eine EJB-Komponente verlässt sich voll auf ihre Ablaufumgebung, die sie von dem Betriebssystem, der Hardware und sogar dem Betriebssystem abstrahiert. Diese Abstraktion stellt sicher, dass man die Komponente auf verschiedenen Plattformen betreiben kann. Dies ist jedoch nur gegeben, wenn der Entwickler nur die Dienste des EJB-Containers beansprucht und nicht versucht auf eigene Faust an dem EJB-Container »vorbei« mit dem Betriebssystem zu kommunizieren. Diese zusätzliche Koppelung würde sich negativ auf die Betriebssystemunabhängigkeit der Komponente auswirken.

Die Aufgabe des EJB-Containers beschränkt sich nicht nur auf die Abstraktion des Betriebssystems sondern es werden auch an dieser Stelle Services oder Dienste der Komponente angeboten. Für die Thread-Verwaltung und die nebenläufige Ausführung der EJB ist auch der Container verantwortlich – somit fällt die Thread-Synchronisation, Starten oder Stoppen von Threads und das Aufrufen von Methoden einer EJB innerhalb eines Threads in den Aufgabenbereich des EJB-Containers. Aus Rücksicht auf die Stabilität des Applikationsservers (es könnten z.B. Deadlocks entstehen) sollte der Entwickler das Arbeiten mit Threads in einem Container unterlassen.

Die meisten Logging-Systeme schreiben ihre Ausgaben in Dateien. Dieser Vorgang dauert natürlich wesentlich länger als eine Konsolenausgabe. Frameworks wie Log4J (*http://jakarta.apache.org*) bieten die Möglichkeit, an die einzelnen Ausgaben asynchron zu schreiben. Der Benutzer des Logging-Frameworks muss nicht warten, bis der Loggeintrag in eine Datei geschrieben wird – die Arbeit kann sofort fortgesetzt werden. Diese Funktionalität lässt sich natürlich nur mit Hilfe von Threads implementieren und wäre an dieser Stelle nicht J2EE-konform.

3.14.3 Lösung

Ein J2EE 1.3 konformer Applikationsserver muss die JMS-API unterstützen und implementieren. Diese API erlaubt das asynchrone Senden von Nachrichten. Für die Nebenläufigkeit sorgt an dieser Stelle jedoch der Applikationsserver und nicht der Entwickler. Intern werden die Threads des Applikationsservers für die nebenläufige

Ausführung von »Queue-Prozessen« benutzt. Dieser Vorgang ist für den J2EE-Entwickler vollkommen transparent.

Eine weitere nützliche Eigenschaft der JMS-API ist die mögliche physikalische Trennung des Lesers vom Schreiber. Der »Queue-Schreiber« könnte innerhalb des Applikationsservers, der »Queue-Leser« auf einem beliebigen Client ausgeführt werden. Die Kommunikation zwischen den beiden Teilnehmern findet hier über die Rechnergrenzen hinweg statt.

Die vorhandene JMS-Infrastruktur lässt sich hervorragend für den Aufbau eines Logging-Systems verwenden. Die Log-Ausgaben werden in eine `javax.jms.Destination` geschrieben. Die asynchrone Abarbeitung der Messages einer Destination übernimmt hier der Applikationsserver. Für die Implementierung des Logging-Frameworks würde sowohl die Publish-Subscribe, als auch die Point-To-Point-Kommunikation in Frage kommen. Für den schreibenden Zugriff müsste dann entweder eine Instanz der Implementierung des Interfaces `javax.jms.Topic` oder der `javax.jms.Queue` verwendet werden.

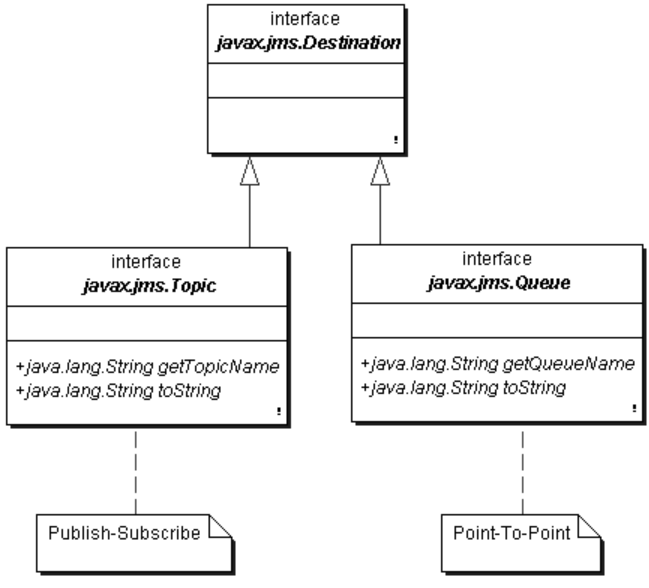


Abbildung 3.65: Publish-Subscribe vs. Point-To-Point

Der Empfänger der Nachrichten könnte sowohl auf dem Applikationsserver als auch außerhalb gestartet werden. Für den Betrieb innerhalb des Applikationsservers eignet sich eine Message Driven Bean für die asynchrone Abarbeitung der Messages. Natürlich gelten hier auch für sie die gleichen Einschränkungen wie für die Session- oder EntityBeans. Eine Message Driven Bean könnte jedoch die Ausgaben nicht in Dateien,

sondern mit Hilfe der Session-Facade in die Datenbank schreiben. Eine Client-Anwendung könnte die aktuellen Log-Ausgaben entweder direkt mit JDBC oder indirekt mit Hilfe einer Session-Facade abholen.

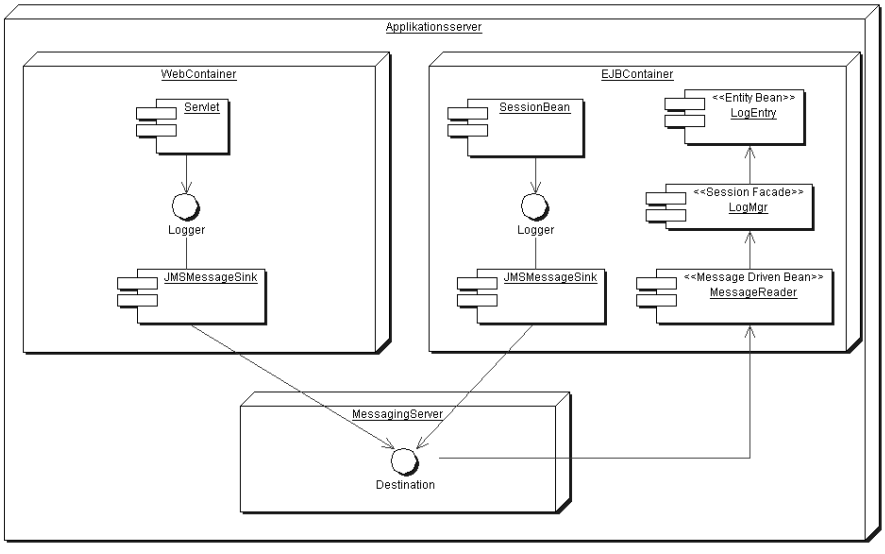


Abbildung 3.66: Message Driven Bean als Message Reader

Auch ein »gewöhnlicher« Client außerhalb des Applikationsservers ist in der Lage die JMS-Nachrichten zu empfangen, indem er sich als `MessageListener` bei einem `javax.jms.QueueReceiver` anmeldet.

3.14.4 Praxis

Bei der Klasse `JMSMessageReceiver` handelt es sich um einen »standalone«-Client, der sich als `javax.jms.MessageListener` bei einer Destination (indirekt) anmeldet. Dieser Client muss nicht in der JVM des Applikationsservers ablaufen, um über neue Nachrichten in einer ausgewählten `javax.jms.Destination` des Applikationsservers benachrichtigt zu werden. Da hier die J2EE-Einschränkungen nicht gelten, darf diese Anwendung in beliebige, lokale Dateien schreiben und ggf. aus diesen lesen.

```
public class JMSMessageReceiver implements MessageListener {
   private QueueReceiver   queueReceiver = null;
   private QueueSession  session         = null;
   private QueueConnection connection    = null;
   private String factoryName                                  = null;
   private String destinationName                              = null;
   private MessageSink messageSink                             = null;
```

```
   private String      sinkProduct                              = null;
   public  final static String DEFAULT_SINK =
"com.abien.j2ee.logging.FileMessageSink";
   public JMSMessageReceiver(String factoryName,String destinationName,String
sinkProduct) throws Exception{
          this.factoryName = factoryName;
          this.destinationName = destinationName;
          this.sinkProduct     = sinkProduct;
          this.init();
    }
   public void init() throws Exception{
      ServiceLocatorEJB locator = new ServiceLocatorEJB();
         LogSinkFactory logSinkFactory = new LogSinkFactory();
         logSinkFactory.setProductClassName(sinkProduct);
         this.messageSink = logSinkFactory.createLogSink();
      QueueConnectionFactory factory    =
locator.getQueueConnectionFactory(this.factoryName);
      this.connection = factory.createQueueConnection();
      this.session    =
this.connection.createQueueSession(true,Session.AUTO_ACKNOWLEDGE);
      Queue   queue      = locator.getQueue(this.destinationName);
      this.queueReceiver = session.createReceiver(queue);
      this.queueReceiver.setMessageListener(this);
   }
   private void processMessage(LogMessageVO logMessage){
         this.messageSink.log(logMessage);
   }
public void onMessage(Message message) {
LogMessageVO log = null;
 if(message instanceof ObjectMessage){
    try{
       ObjectMessage objectMessage = (ObjectMessage)message;
       Object temp                 = objectMessage.getObject();
       if(temp  instanceof LogMessageVO){
              log = (LogMessageVO)temp;
               this.processMessage(log);
                }
           }catch(Exception e){        }
    }
```

Zunächst wird nach einer Instanz vom Typ `javax.jms.QueueConnectionFactory` gesucht. Für die Suche wird ein JNDI-Name einer auf dem Server eingerichteten `QueueConnectionFactory` benötigt. Mit Hilfe dieser Instanz wird die `javax.jms.QueueConnection` erzeugt. Diese wird wiederum für die Erzeugung einer `javax.jms.Session` benötigt.

Ferner wird zusätzlich nach einer Instanz der `javax.jms.Queue` gesucht. Diese Instanz kann mit einer Senke verglichen werden, in die Nachrichten geschrieben werden. Die Methode `createReceiver` der `Session` erzeugt den für die Anmeldung des `javax.jms.Mes-`

sageListeners notwendigen QueueReceiver. Allerdings wird hier die eben gefundene Queue-Instanz als Methodenparameter erwartet. Nun meldet sich die Instanz JMSMessageReceiver selbst als MessageListener bei dem QueueReceiver an.

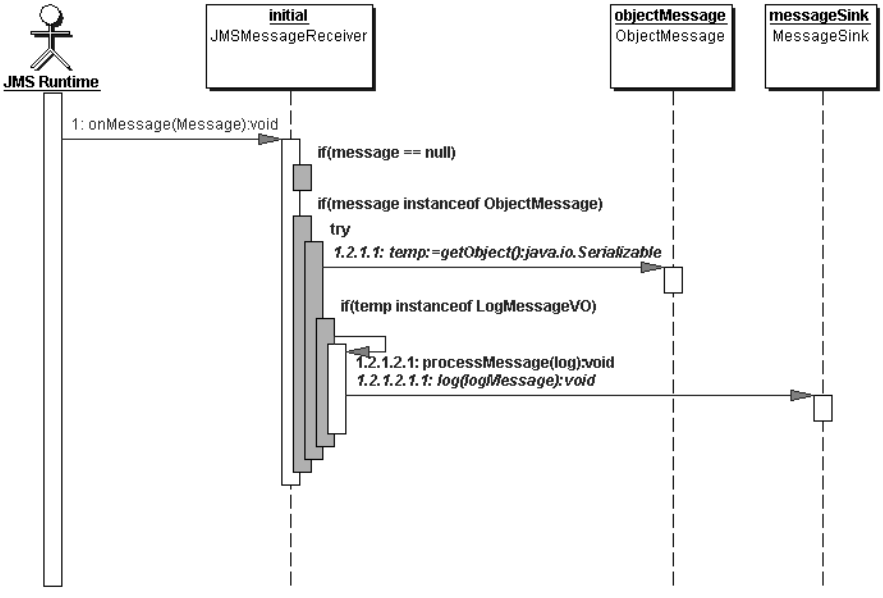


Abbildung 3.67: Delegation des Aufrufs an eine Implementierung des Interfaces MessageSink

Nach der Anmeldung bei der Instanz des QueueReceivers, ist der JMSMessageReceiver für den Empfang von Nachrichten bereit. Die Methode onMessage wird bei Ankunft von Messages asynchron aufgerufen. Als Parameter wird hier eine Instanz vom Typ javax.jms.Message erwartet. In unserem Fall wird die javax.jms.ObjectMessage erwartet. Sie dient als Verpackung der LogMessageVO diese erst einen Log-Eintrag repräsentiert.

Der entpackte Instanz der LogMessageVO wird intern an eine Implementierung der MessageSink Instanz übergeben. Somit ist es möglich, das Schreiben der Log-Einträge von dem Empfangsprozess zu entkoppeln. Die Implementierung der MessageSink wird von der LogSinkFactory erzeugt. Der JMSMessageReceiver benutzt diese Factory um die Implementierung zu erhalten. Somit bleibt der JMSMessageReceiver von der Implementierung der MessageSink unabhängig.

```
public class LogSinkFactory {
      private MessageSinkFactory sinkFactory = null;
      private String productClassName        = null;
      public  final static String DEFAULT_SINK =
"com.abien.j2ee.logging.jms.JMSMessageSink";
```

```
        public LogSinkFactory(){
              this.sinkFactory = MessageSinkFactory.getInstance();
              this.productClassName = DEFAULT_SINK;
        }
  public MessageSink createLogSink(){
  return(MessageSink)this.sinkFactory.createNewMessageSink(getProductClassName());
        }
        public String getProductClassName() {
               return productClassName;
        }
        public void setProductClassName(String productClassName) {
              this.productClassName = productClassName;
        }
  }
```

Eine mögliche Implementierung des Interfaces `com.abien.j2ee.logging.MessageSink` ist die Klasse `FileMessageSink`. Diese Instanz nimmt die Log-Einträge entgegen und reicht diese an eine Logger-Instanz des Packages `java.util.logging` (ab JDK 1.4.X) weiter. Für die weitere Verarbeitung der Einträge ist nun die Logging API der J2SE zuständig. Die Inhalte können sofort geschrieben, oder für die Weiterverarbeitung verschickt werden.

```
public class FileMessageSink implements MessageSink {
      private static Logger logger = null;
      static {
            logger = Logger.getLogger("com.abien.j2ee.starfinder");
      }
public void log(LogMessageVO log) {
  if (log.isError())
      logger.log(Level.SEVERE,log.getMessage(),log.getThrowable()b);
  else
      logger.log(Level.INFO,log.toString());
      }
}
```

Nachdem wir jetzt kennengelernt haben, wie der Empfang und die Verarbeitung der Log-Einträge aussehen, schauen wir uns nun den Schreibprozess genauer an. Für das Verschicken der Log-Einträge wird auch hier die JMS-API verwendet. Der Mehraufwand für das Verschicken einer Nachricht ist ziemlich hoch, sodass dieser Prozess vor dem J2EE-Entwickler versteckt wurde. Die Schnittstelle eines Logging-Systems sollte einfach ausfallen, damit diese auch akzeptiert wird. In unserem Fall reicht die Verwendung der statischen Methoden der Klasse InitialLogContext, um in die Logs zu schreiben. Es wurden zunächst die statischen Methoden `log(String message)`, `trace(String message)` und die überladene Methode `err` bereitgestellt. Für das Schreiben einer Tracing-Ausgabe reicht der folgende Aufruf `InitialLogContext.log("Tracing Ausgabe")`.

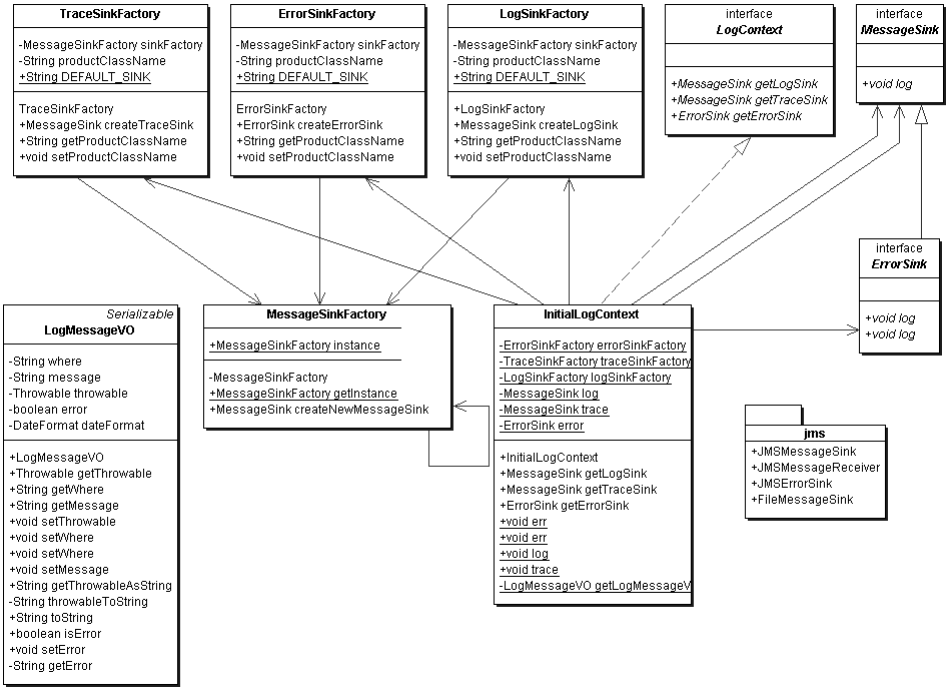


Abbildung 3.68: Die beteiligten Klassen des Packages `logging`.

```
public class InitialLogContext implements LogContext{

//…
      static{
            new InitialLogContext();
      }

      public InitialLogContext(){
            errorSinkFactory = new ErrorSinkFactory();
            traceSinkFactory = new TraceSinkFactory();
            logSinkFactory   = new LogSinkFactory();
            error = errorSinkFactory.createErrorSink();
            trace = traceSinkFactory.createTraceSink();
            log   = logSinkFactory.createLogSink();
      }
      //… "getter"      -  Methoden
      public static void err(Object where, String why, Throwable t) {
            error.log(where,why,t);
      }
      public static void err(String message, Throwable t) {
            error.log(message,t);
      }
      public static void log(String message) {
```

```
            log.log(getLogMessageVO(message));
      }
      public static void trace(String message) {
            trace.log(getLogMessageVO(message));
      }
      private static LogMessageVO getLogMessageVO(String message){
            LogMessageVO messageVO = new LogMessageVO();
            messageVO.setMessage(message);
            return messageVO;
      }
}
```

Die Klasse `InitialLogContext` ist lediglich für die Entgegennahme und die Weiterleitung der Log-Einträge zuständig. Abhängig von den aufgerufenen Methoden wird der Logeintrag an unterschiedliche Instanzen der `MessageSink` weitergeleitet. Auch hier wurden die Instanzen von den Factories erzeugt, um den `InitialLogContext` von der Implementierung zu entkoppeln. Da der `IntialLogContext` lediglich die Interfaces `MessageSink` und nicht ihre Implementierungen kennt, ist die Austauschbarkeit der Implementierungen ziemlich hoch. Für das Verschicken von Nachrichten mit der JMS-API ist die Klasse `JMSMessageSink` zuständig. Sie kann als das Gegenstück des `JMSMessageReceivers` gelten. Auch in dieser Klasse wird zunächst nach einer `QueueConnectionFactory` gesucht. Mit Hilfe der `QueueConnectionFactory` wird eine Instanz der `QueueConnection` erzeugt. Diese ist für die Erzeugung der `javax.jms.Session` zuständig. Danach wird wiederum nach einer Queue-Instanz in dem Naming-Service (JNDI) des Applikationsservers gesucht, da diese für die Erzeugung des `QueueSenders` benötigt wird.

```
public class JMSMessageSink implements MessageSink {
     public static final String JMS_CONNECTION_FACTORY        =
"com.abien.j2ee.log.JMSFactory";
     public static final String QUEUE                          =
"com.abien.j2ee.log.LogQueue";
     private ObjectMessage message                             = null;
     private QueueSender   queueSender                         = null;
     private QueueSession  session                             = null;
     private QueueConnection connection                        = null;
     private String factoryName                                = null;
     private String destinationName                            = null;
    public JMSMessageSink() throws Exception{
          this.init();
    }
   public void init() throws Exception{
      ServiceLocatorEJB locator = new ServiceLocatorEJB();
      QueueConnectionFactory factory    =
locator.getQueueConnectionFactory(this.factoryName);
      this.connection = factory.createQueueConnection();
      this.session    =
this.connection.createQueueSession(true,Session.AUTO_ACKNOWLEDGE);
      Queue   queue      = locator.getQueue(this.destinationName);
```

```
        this.queueSender = session.createSender(queue);
        this.message = session.createObjectMessage();
        connection.start();
    }
     public void setDestinationName(String destinationName){
         this.destinationName = destinationName;
     }
     public void setQueueConnectionFactoryName(String factoryName){
         this.factoryName = factoryName;
     }
      public void commit(){
         try{
         this.session.commit();
         }catch(Exception e) {        }
     }
public void log(LogMessageVO logMessageVO) {
   try{
         message.setObject(logMessageVO);
         queueSender.send(message);
         this.commit();
      }catch(JMSException exception) {    }
}
```

Mit dem `QueueSender` ist es nun möglich, einzelne Nachrichten zu verschicken. Der `JMSMessageReceiver` erwartet eine Instanz der `javax.jms.ObjectMessage`, sodass auch diese von der `JMSMessageSink` verschickt werden sollte. Als Inhalt dient hier eine Instanz der Klasse `LogMessageVO`.

3.14.5 Fazit

Obwohl die J2EE-Spezifikation das Arbeiten mit Dateien innerhalb des EJB-Containers verbietet, lässt sich trotzdem ein Logging-Mechanismus implementieren. Allerdings müssen an dieser Stelle weiterhin die geltenden Einschränkungen der J2EE-Spezifikation berücksichtigt werden. Die JMS-Spezifikation ermöglicht die Kommunikation des Applikationsservers mit anderen JMS-Clients, die auch auf anderen JVMs gestartet werden können. Der schreibende Zugriff auf Dateien muss nicht mehr innerhalb einer J2EE-Umgebung stattfinden. Für die Auslagerung der Schreibzugriffe könnten auch RMI, CORBA oder sogar SOCKETS dienen. Allerdings müsste man sich bei der Benutzung dieser APIs um die Ausfallsicherheit, Transaktionen und das Clustering selbst kümmern. Die JMS-API erfüllt alle diese nichtfunktionalen Anforderungen und noch zusätzlich die Möglichkeit, asynchron die einzelnen Log-Einträge schreiben zu können. Der Benutzer dieses Loggingsystems muss nicht warten, bis die Einträge in die Dateien geschrieben werden – die Ablage in eine `javax.jms.Destination` reicht völlig.

Bei der Benutzung von persistenten Queues lässt sich die Ausfallsicherheit des Systems weiterhin erhöhen. Die Inhalte einer persistenten Queue werden für die Weiterverarbeitung im Sekundärspeicher abgelegt. Bei diesem kann es sich entweder um

einfache Dateien oder Datenbanktabellen handeln. Bei einem Ausfall des EJB-Containers, oder des JMS-Servers gehen die Inhalte somit nicht verloren. Nach dem erneuten Hochfahren des Applikationsservers stehen die »alten« Inhalte der Queue sofort zur Verfügung. Obwohl diese Variante die Ausfallsicherheit des Logging-Systems erhöht, sollte diese nur bei Bedarf verwendet werden. Die Performance dieser Lösung ist wesentlich schlechter als die Performance einer einfachen Queue. Beide Varianten lassen sich rein konfigurativ, ohne einen Eingriff in den JMS-Code, einstellen.

Man sollte also den Systemadministrator regelmäßig auf eine Pizza einladen, damit diese Einstellungen einfach und bequem umgesetzt werden können ☺.

3.15 Warum gibt die ejbCreate Methode einer EntityBean ein Primary Key zurück?

Bei der Einführung der EntityBeans in den EJB-Schulungen fragen sich die Schulungsteilnehmer oft, warum die `ejbCreate` Methode einer EntityBean die übergebene Primary-Key-Instanz wieder zurückgibt. Da diese Frage beinahe in jeder Schulung gestellt wird, wurde sie auch in dieses Buch aufgenommen.

3.15.1 Problemstellung

Bei der Betrachtung des Home-Interfaces einer Bean stellt man fest, dass sich hier um ein Factory-Pattern oder sogar um ein Abstract-Factory-Pattern (GoF) handelt. Die Hauptaufgabe des Home-Interfaces einer EJB ist die Erzeugung oder Bereitstellung einer Instanz des Remote-Interfaces. Diese Factory ist für die Dienste des EJB-Containers von essenzieller Bedeutung, da sich nur so der Lebenszyklus einer EJB bestimmen lässt.

```
public interface StarHome extends EJBHome{
 public Star  create(String catalogPrefix, String zoneSign, int declinationZone,
int numberWithinZone, String note,float magnitude,int rightAscensionHours,int
rightAscensionMinutes,float rightAscentionSeconds,String signOfDeclination,int
declinationDegrees,float declinationMinutes) throws
CreateException,RemoteException;
public Star findByPrimaryKey(StarPK pk) throws FinderException,RemoteException;
}
```

Bei der Methode `ejbCreate` handelt es sich um eine indirekte Implementierung der Methode `create` aus dem Home-Interface. Dabei verlässt sich die Implementierung des `StarHome`-Interfaces auf die »ejb-Prefix-Konvention« und ruft die Methode `ejbCreate` der Bean-Klasse auf. Allerdings unterscheidet sich der Rückgabewert der beiden Methoden. Die Methode des Home-Interfaces gibt ein Remote-Interface zurück, die Methode der Bean-Klasse den Primärschlüssel der Komponente.

```
   public StarPK ejbCreate(String catalogPrefix, String zoneSign, int
declinationZone, int numberWithinZone, String note,float magnitude,int
rightAscensionHours,int rightAscensionMinutes,float rightAscentionSeconds,String
signOfDeclination,int declinationDegrees,float declinationMinutes)throws
CreateException{
        this.setCatalogPrefix(catalogPrefix);
        this.setZoneSign(zoneSign);
        this.setDeclinationZone(declinationZone);
        this.setNumberWithinZone(numberWithinZone);
        this.setNote(note);
        this.setMagnitude(magnitude);
        this.setRightAscensionHours(rightAscensionHours);
        this.setRightAscensionMinutes(rightAscensionMinutes);
        this.setRightAscentionSeconds(rightAscentionSeconds);
        this.setSignOfDeclination(signOfDeclination);
        this.setDeclinationDegrees(declinationDegrees);
        this.setDeclinationMinutes(declinationMinutes);
        return null;
    }
```

Der Aufruf der Methode `ejbCreate` lässt sich mit dem INSERT-Statement einer relationalen Datenbank vergleichen. Ein INSERT-Statement benötigt zumindest den Primärschlüssel für die Erzeugung eines neuen Datensatzes in der Datenbanktabelle. Die Methode `ejbCreate` der Bean benötigt zumindest genügend Informationen für die Bildung des Primärschlüssels, damit der Datensatz in die Datenbank überhaupt geschrieben werden kann. Dabei spielt es keine Rolle, ob der Schlüssel aus einer SQL-Sequence stammt, oder als Parameter von »außen« übergeben wird. Umso mehr stellt sich an dieser Stelle die Frage, warum die bereits vorhandene oder sogar gerade übergebene Information wieder zurückgegeben wird.

3.15.2 Technischer Hintergrund

Der EJB-Container speichert den Zustand der EntityBean erst nach dem Beenden der Methode `ejbCreate` in dem Sekundärspeicher ab. Bei einer relationalen Datenbank wird an dieser Stelle das INSERT-Statement ausgeführt. Dem EJB-Container steht es frei, auch länger mit der persistenten Ablage des Zustandes zu warten. Die Methode `ejbCreate` wird unmittelbar nach dem Aufruf der Methode `create` des Clients ausgeführt. Für diesen Vorgang wird eine EntityBean-Instanz aus dem Pool verwendet.

Wie bereits erwähnt, wird die Methode `create` in dem Home-Interface definiert. Die Implementierung dieses Interfaces – das `HomeObject` - wird beim Deployment automatisch generiert. Diese Implementierung greift auf die EntityBean-Instanz zu. Für den Aufruf der Methode `create` kann eine beliebige Instanz der Bean-Klasse aus dem Pool verwendet werden.

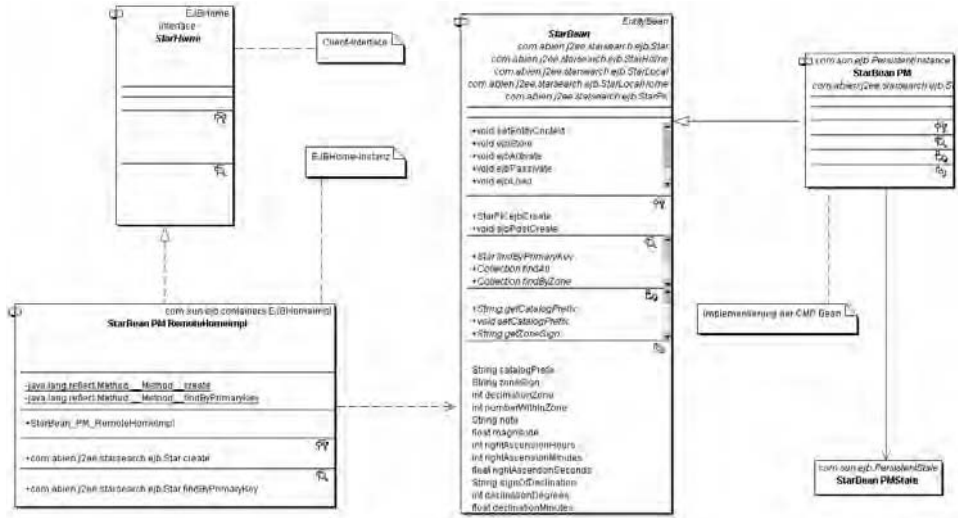


Abbildung 3.69: Eine EntityBean mit den beim Deployment generierten Klassen

Nachdem der Datensatz in der Datenbank erfolgreich angelegt wurde, gibt die Implementierung der abstrakten Bean-Klasse den Primärschlüssel an den EJB-Container zurück. Nun kann die Instanz des EJB(Local)Objects an den Client zurückgegeben werden. Der Client greift auf diese Instanz entweder direkt (die Local-Variante) oder indirekt (mit Hilfe der Stubs und Skeletons) zu.

Der Schlüssel wird für die Erzeugung der Implementierung des Remote-Interfaces und die Assoziation der Bean-Klasse mit dem Primärschlüssel benötigt. Die Primärschlüssel werden unabhängig von den Bean-Instanzen verwaltet. Eine Bean-Instanz, die sich gerade in dem Pool befindet, ist nicht mit einem Primärschlüssel verbunden. Erst vor der ersten Benutzung wird sie mit einem Primärschlüssel assoziiert, um die Bean mit den Daten aus der Datenbank laden zu können. Zu diesem Zweck wird die Methode `ejbLoad` der Bean aufgerufen.

```
    public void ejbLoad(){
    StarPK currentKey =      (StarPK)this.entityContext.getPrimaryKey();
//erst mit dem Primärschlüssel ist das Laden der Bean möglich…
    this.loadData(currentKey);
    }
```

Das Zusammenspiel der Remote- und der Home-Interfaces erinnert an das Abstract-Factory-Pattern (GoF). Dabei definiert das Home-Interface die Schnittstelle der Factory. Das Remote-Interface ist für die Festlegung eines konkreten Produkts zuständig. Die konkrete Fabrik und das konkrete Produkt werden beim Deployment generiert. Beide Implementierungen sind applikationsserverabhängig. Da sie jedoch nicht das Bestand-

teil des Enterprise-Archives (EAR) sind, bleibt eine Bean trotzdem von dem jeweiligen Applikationsserver-Hersteller unabhängig. Die Existenz der beiden Implementierungen ist für einen EJB-Entwickler vollkommen transparent.

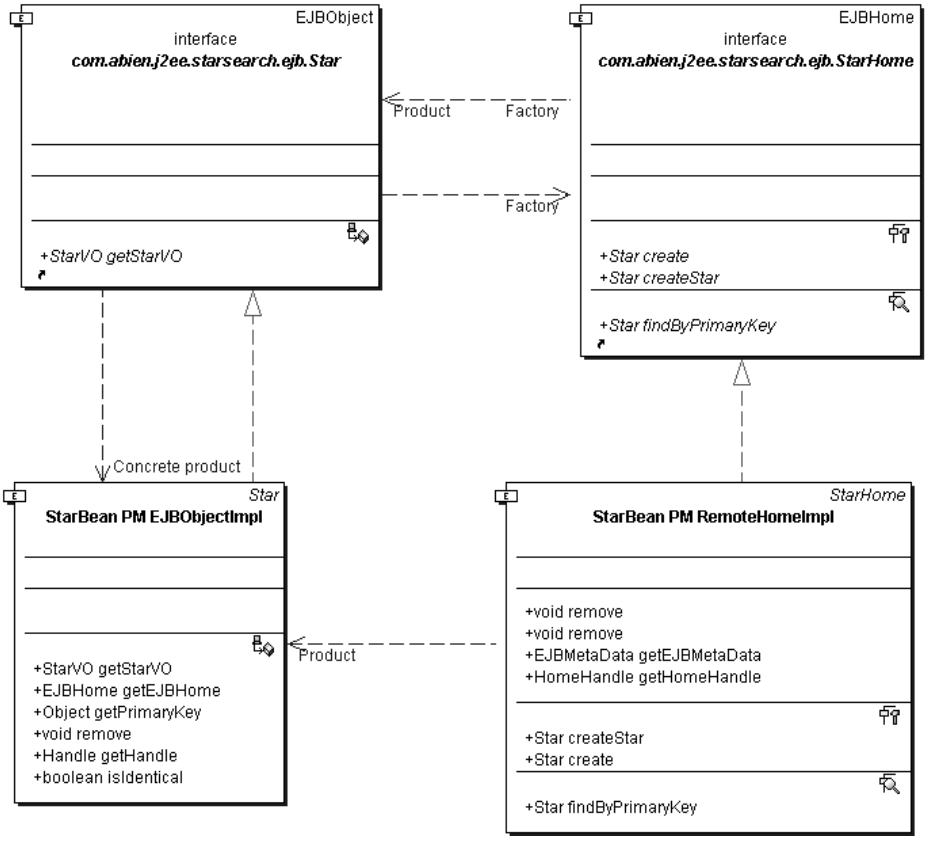


Abbildung 3.70: Das Abstract-Factory-Pattern am Beispiel von StarHome

3.15.3 Lösung

Die Methoden `ejbCreate(…)` einer EntityBean geben eine Instanz des Primärschlüssels zurück, da dieser von dem EJB-Container für interne Zwecke benötigt wird. Für die Erzeugung einer EntityBean holt sich der EJB-Container eine bereits existierende Bean-Instanz aus dem Pool. In dem Instanzen-Pool befinden sich EntityBeans im undefinierten Zustand – diese wurden also noch nicht mit dem Primärschlüssel assoziiert. Erst nachdem der Zustand der Bean in einer Datenbank persistent abgelegt wurde, erhält der EJB-Container von der Implementierung der Bean den Primärschlüssel. Dieser Primärschlüssel ist für die eindeutige Identifizierung einer Bean-Instanz zuständig. Die Instanz einer Bean-Klasse kann jedoch jederzeit gepoolt werden – sie verliert in dem

Fall wieder ihre Identität. Bei dem nächsten Zugriff des Clients »reaktiviert« der EJB-Container die Bean mit dem `ejbActivate` Aufruf und versetzt diese in einen gültigen Zustand. Dies kann wiederum mit dem Aufruf der Methode `ejbLoad` erreicht werden. Die Bean-Instanz benötigt allerdings den Wert des Primärschlüssels, um ihre Daten aus der Datenbank laden zu können. Die Instanz des Primärschlüssels kann aus dem `javax.ejb.EntityContext` mit dem Aufruf der Methode `getPrimaryKey` geholt werden.

Die Methode `ejbCreate` der Bean-Klasse gibt jedoch lediglich einen `null`-Wert zurück. Obwohl die EJB-Spezifikation diese Vorgehensweise vorschreibt, ist der eigentliche Rückgabewert an dieser Stelle vollkommen egal.

```
public abstract class StarBean implements EntityBean{
   public StarPK ejbCreate(String catalogPrefix,
        return null;
    }
}
```

Die beim Deployment generierte Implementierung dieser Klasse überschreibt diese Methode und gibt einen sinnvollen Wert zurück. Die Implementierung ruft die Methode der Superklasse (hier `StarBean`) auf, ignoriert jedoch völlig den Rückgabewert.

```
//Implementierung der J2SDKEE 1.3.1 RI
public final class StarBean_PM extends StarBean implements
com.sun.ejb.PersistentInstance {
     private javax.ejb.EntityContext __entityContext = null;
   public com.abien.j2ee.starsearch.ejb.StarPK ejbCreate(java.lang.String param0,
...) throws javax.ejb.CreateException {
      com.sun.ejb.Partition partition =
com.sun.ejb.PersistenceUtils.getPartition(this);
      partition.beforeEjbCreate(this);
      //Rückgabewert wird ignoriert...
      super.ejbCreate(param0, param1, param2, param3, param4, param5, param6,
param7, param8, param9, param10, param11);
      return (com.abien.j2ee.starsearch.ejb.StarPK)
partition.afterEjbCreate(this);
    }
    public void ejbPostCreate(java.lang.String param0, …) throws
javax.ejb.CreateException {
      com.sun.ejb.Partition partition =
com.sun.ejb.PersistenceUtils.getPartition(this);
      partition.ejbPostCreate(this);
      super.ejbPostCreate(param0, param1, param2, param3, param4, param5, param6,
param7, param8, param9, param10, param11);
    }
  public void ejbRemove()  {
     super.ejbRemove();
```

```
    com.sun.ejb.Partition partition =
com.sun.ejb.PersistenceUtils.getPartition(this);
     partition.ejbRemove(this);
  }
 public void ejbLoad()  {
  com.sun.ejb.Partition partition =
com.sun.ejb.PersistenceUtils.getPartition(this);
       partition.ejbLoad(this);
       super.ejbLoad();
  }
 public void ejbStore()  {
     super.ejbStore();
     com.sun.ejb.Partition partition =
com.sun.ejb.PersistenceUtils.getPartition(this);
     partition.ejbStore(this);
   }
//…
}
```

Das `EJBHomeObject` wartet schon ungeduldig auf den Primärschlüssel, um diesen dem EJB-Container zu übergeben. Nun kann auch die Methode `ejbPostCreate` der Bean aufgerufen werden, da der Primärschlüssel bereits an den Container übergeben wurde und somit auch der Bean zur Verfügung gestellt werden kann.

```
//Implementierung der J2SDKEE 1.3.1 RI
public final class StarBean_PM_LocalHomeImpl extends
com.sun.ejb.containers.EJBLocalHomeImpl implements
com.abien.j2ee.starsearch.ejb.StarLocalHome {

   public com.abien.j2ee.starsearch.ejb.StarLocal create(java.lang.String
param0,…) throws javax.ejb.CreateException {
//…
      try {
            this.getContainer().preInvoke(i);
      com.abien.j2ee.starsearch.ejb.StarBean_PM ejb =
(com.abien.j2ee.starsearch.ejb.StarBean_PM) i.ejb;
java.lang.Object primaryKey = ejb.ejbCreate(param0, param1, param2, param3,
param4, param5, param6, param7, param8, param9, param10, param11);
//Primärschlüssel für den Aufruf der Methode ejbPostCreate notwendig…
 this.getContainer().postCreate(i, primaryKey);
 ejb.ejbPostCreate(param0, param1, param2, param3, param4, param5, param6, param7,
param8, param9, param10, param11);
      } catch(Throwable c) {}
//…
      return
(com.abien.j2ee.starsearch.ejb.StarLocal)((com.sun.ejb.containers.EJBLocalObjectIm
pl)i.ejbObject);
    }
}
```

Der Methode `ejbCreate` wird der Primärschlüssel von »außen« übergeben. An den Rückgabewert ist jedoch der EJB-Container interessiert, da er die Instanz für die Identifikation einzelner Bean-Instanzen benötigt. Der EJB-Container wartet jedoch ab, bis die Methode `ejbCreate` erfolgreich zurückkehrt – die Abarbeitung dieser Methode könnte auch schief gehen (z.B. `javax.ejb.DuplicateKeyException`). Erst nach dem erfolgreichen Abschluss merkt er sich endgültig den Primärschlüssel der Bean.

3.16 Was muss in der Design- und Implementierungsphase berücksichtigt werden?

Obwohl diese Frage nur selten in den Schulungen und Seminaren gestellt wird, ist die Antwort für die erfolgreiche Produktionseinführung umso wichtiger. Obwohl die Clusterfunktionalität völlig von dem Applikationsserver-Hersteller übernommen wird, ist die Existenz des Clusters für den EJB-Entwickler oder -Architekten keineswegs transparent. Da in der Praxis meistens auf einzelnen Server-Knoten entwickelt wird, lässt sich die Anwendung erst recht spät im Clusterbetrieb testen. Nicht nur für die Stabilität, sondern auch für die Performance und Konsistenz der Daten ist das Verständnis der Funktionalität eines Clusters absolut notwendig.

3.16.1 Problemstellung

Bei einem Applikationsserver handelt es sich im Prinzip um eine (möglicherweise »pure«) Java-Anwendung. Für den Betrieb des Applikationsservers ist die Installation einer JVM unbedingt notwendig. Ein Applikationsserver bietet seinen Komponenten eine definierte Ablaufumgebung an. Die J2EE-Komponenten sind erst auf dem Applikationsserver lauffähig. Die J2EE-Komponenten können sowohl innerhalb des Web-Containers (JSPs und Servlets), als auch auf dem EJB-Container (Sesssion-, Message Driven- und EntityBeans) ablaufen.

Im »clusterlosen« Betrieb könnten alle diese Komponenten auf einer JVM ablaufen. Die Zustände der Komponenten (nur wenige J2EE-Anwendungen sind zustandslos) können im Speicher gehalten werden. Diese Vorgehensweise ist sehr performant, da die Bereitstellung des Zustandes einer Komponente mit der Performance eines Methodenaufrufs gleichzusetzen ist. Beim Absturz dieser JVM sind alle Komponenten betroffen – die Anwendung ist für die Clients nicht mehr verfügbar.

Die Verwendung einer geclusterten Umgebung kann die Ausfallsicherheit einer Anwendung deutlich erhöhen. Dabei werden die gleichen Komponenten auf mehreren Servern redundant installiert. Bei dem Ausfall eines Serverknotens, kann die Arbeit von anderem Knoten automatisch übernommen werden. Der Applikationsserver wird auf mehrere Rechner zusammen mit den installierten Anwendungen verteilt. Da es sich bei

dem Applikationsserver um eine ganz gewöhnliche JAVA-Anwendung handelt, muss diese auch auf jedem Server einzeln gestartet werden. Die gleichen Komponenten laufen auf mehreren JVMs gleichzeitig und können miteinander wahllos kommunizieren!

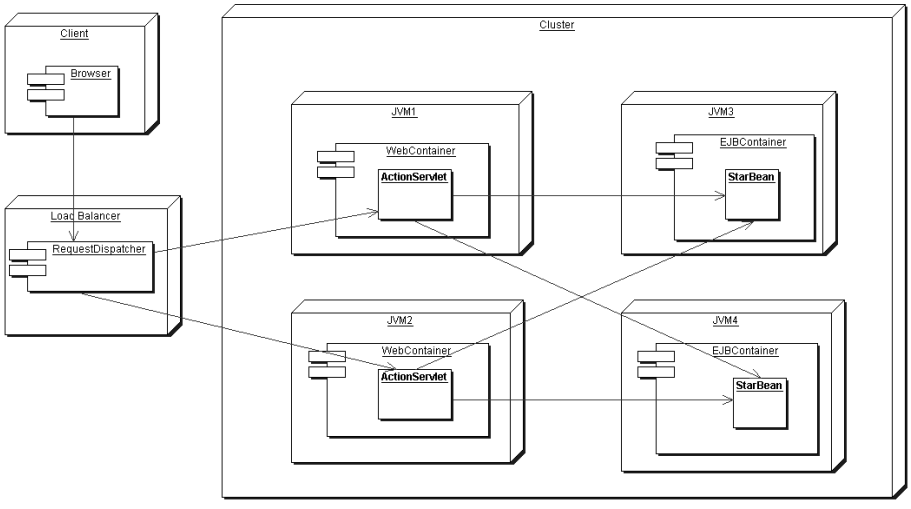


Abbildung 3.71: Die Verteilung eines Applikationsservers auf mehrere JVMs – das Clustering

Die Verteilung der Komponenten schließt die Benutzung des gemeinsamen Speichers für die Ablage des Zustandes komplett aus. Jeder Request des Clients kann auf einem anderen Knoten landen. Der Speicher der einzelnen Clusterknoten und der JVMs ist voneinander völlig unabhängig. Was passiert an dieser Stelle? Wie verhält sich eine Anwendung in dieser Umgebung?

Ein ähnliches Problem tritt bei der Verwendung des Singleton-Patterns (GoF) auf. In einer nicht geclusterten Umgebung kann sich der Entwickler darauf verlassen, dass nur eine einzige Instanz des Singletons erzeugt werden kann. Intern arbeitet ein Singleton mit einer statischen Referenz auf sich selbst.

```
public class Singleton {
      public  static Singleton instance = null;
      private Singleton (){}
      public final static Singleton getInstance(){
            if(instance==null)
                  instance = new Singleton ();
            return instance;
      }
}
```

Bei den statischen Variablen handelt es sich um einen instanzunabhängigen Speicher – dieser »funktioniert« wieder nur innerhalb einer JVM. Singletons eignen sich hervorragend für die Erzeugung von Primärschlüsseln, da hier einfach eine »globale« Variable hochgezählt werden kann. Bei der Verteilung eines Singletons auf mehrere JVMs (z. B. im Clusterbetrieb …) ist die globale Variable nicht mehr global – und die Primärschlüssel sind nicht mehr primär …

Gute Applikationsserver sind in der Lage, die Daten der Datenbank zu cachen. Das verhindert das Laden der Datensätze aus der Datenbank und die Konvertierung von einfachen Datentypen in »echte« Objekte. Bei den Objekten handelt es sich um EntityBeans, die mit gültigen Daten im Speicher bereitgehalten werden können. Der tatsächliche Datenbankzugriff kann auf ein Minimum reduziert werden. Da im Cluster mehrere EJB-Container gleichzeitig aktiv sind, ist der Instanzen-Cache ebenfalls mehrfach vorhanden. Muss man sich an dieser Stelle um die Konsistenz der Daten selbst kümmern oder wird diese Aufgabe von dem Applikationsserver automatisch übernommen?

Ähnliche Fragestellungen ergeben sich bei der Betrachtung von einmal vorhandener Ressourcen wie `javax.servlet.ServletContext`, `javax.servlet.http.HttpSession`, `javax.sql.DataSource`, `javax.jms.Destination` und sogar Stateful SessionBeans.

3.16.2 Lösung

Leider wurden die Funktion und das Verhalten eines Clusters nicht in der J2EE-Spezifikation festgelegt. Somit bleibt die Implementierung des Clusters völlig dem Applikationsserver-Hersteller überlassen. Von der Güte der Implementierung hängen die Performance und die tatsächliche Ausfallsicherheit der deployten J2EE-Anwendung ab. Betrachten wir nun mal die kritischen Bereiche eines Applikationsservers.

Web-Container

In einem Web-Container ist die Ablage von gemeinsamen Ressourcen in dem `javax.servlet.ServletContext` sehr beliebt. Die Instanz des `ServletContext` wird von allen Komponenten einer Web-Anwendung geteilt. Die Methoden `public void setAttribute (String name, Object object)` und `public Object getAttribute(String name)` ermöglichen die Ablage von beliebigen Objekten. Die Instanz des `ServletContext`s eignet sich somit auch hervorragend für das Caching von Objekten beliebigen Typs. Diese Vorgehensweise verhindert unnötige Zugriffe auf die Geschäftslogikschicht, somit lässt sich die Performanz der Gesamtanwendung deutlich erhöhen. Allerdings kann diese Optimierung nur bei statischen Inhalten angewendet werden.

Leider wird im Cluster die Instanz des `ServletContext` überhaupt nicht repliziert. Der `ServletContext` läuft redundant auf allen Clusterknoten gleichzeitig – die jeweiligen Instanzen kennen sich gegenseitig nicht. Falls benutzerabhängige Informationen innerhalb eines Requests in dem `ServletContextS` abgelegt werden, werden die Nachbarsknoten über die

Änderung des Zustandes nicht benachrichtigt. Die meisten Applikationsserver benutzen Load-Balancer (Hardware oder Software), die für gleichmäßige Verteilung der Last auf die jeweiligen Knoten zuständig sind. Es könnte also passieren, dass der nächste Request auf einem anderen Clusterknoten landet – auf diesem befindet sich aber das gerade abgelegte Objekt nicht. Falls es sich hier dabei um einen Cache handelt, ist diese Tatsache nicht so schlimm – die Daten können immerhin nochmals geholt werden. Anders sieht der Sachverhalt bei der Verwendung des ServletContexts als Zwischenablage aus. In dem Fall wird die Anwendung im Cluster nicht funktionieren. Das Auffinden von solchen Fehlern kann Ihnen Kopfschmerzen (oder zumindest schlaflose Nächte) bereiten. Je nach eingestellter Strategie des Load-Balancers, kann ferner passieren, dass der Fehler nur unter bestimmten Umständen (Konfigurationen) auftritt.

Ein wenig transparenter ist die Verwendung von Instanzen der `javax.servlet.http.HttpSession`. Diese wird von den meisten Applikationsservern repliziert. Es werden dabei unterschiedliche Strategien für die Replizierung verwendet. Obwohl die Inhalte der HttpSession repliziert werden, müssen trotzdem einige Besonderheiten im Clusterbetrieb berücksichtigt werden. Der Abgleich der Session-Daten kann entweder mit der Benutzung einer gemeinsamen Datenbank oder mit der Verteilung der Zustände auf alle Cluster-Teilnehmer sichergestellt werden.

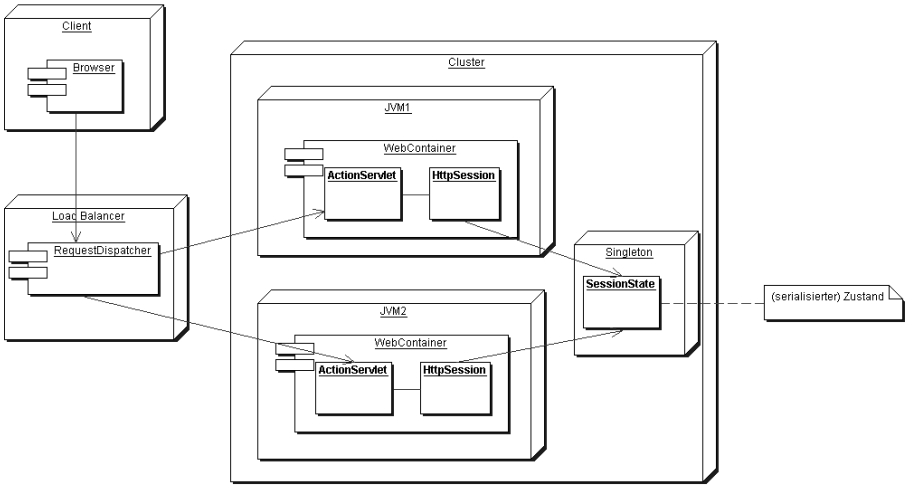


Abbildung 3.72: Die Replizierung der HttpSession

Die einfachste Methode ist die Ablage der Session-Daten in einer gemeinsamen Datenbank. Dies kann nur erfolgen, wenn die Inhalte der Session das `java.io.Serializable` Interface implementieren. Bei einer Änderung der Sessioninhalte auf einem beliebigen Knoten, werden die Daten der Session unmittelbar in die gemeinsame Datenbank abgelegt. Zu diesem Zweck werden meist BLOB oder RAW Spalten einer Datenbank-

tabelle benutzt. Bei der Serialisierung einer Instanz werden ihre Daten und nicht die »echte« Referenz auf diese Instanz in einen »Byte-Strom« geschrieben. Der Inhalt des Bytestroms wird in die gemeinsame Datenbank geschrieben. Prinzipiell ist dieses Verhalten gewollt, allerdings können einige Seiteneffekte entstehen.

Falls in die `HttpSession` eine Referenz auf ein Objekt abgelegt und dieses nachträglich verändert wird, wird die Änderung nicht in die Datenbank geschrieben. Um über die Änderung des Objekts den ganzen Cluster zu benachrichtigen, muss der Inhalt der `HttpSession` nochmals in die Datenbank geschrieben werden. Dies kann lediglich mit dem Überschreiben der aktueller Inhalte – also dem Aufruf der Methode `public void setAttribute(String key,Object value)` forciert werden. Auch dieser Unterschied kann die Funktionsfähigkeit einer Anwendung beeinträchtigen. Obwohl eine Anwendung in der Entwicklungsumgebung getestet wurde (meistens in einem Clusterknoten), muss diese noch lange nicht im Cluster funktionieren.

Nicht nur die Funktionstüchtigkeit, sondern auch die Performanz kann durch die Benutzung eines Clusters beeinträchtigt werden. In einem einzelnen Knoten ist die Replizierung der `HttpSession` nicht notwendig. Der Inhalt der Session kann einfach im Speicher gehalten werden. Der Zugriff auf die im Speicher vorhandenen Daten ist sehr performant (kaum messbar), die Serialisierung der Session-Daten und ihre Ablage in einer JDBC-Ressource ist deutlich langsamer (und auf jeden Fall messbar). Aus diesem Grund schlagen viele Hersteller die Einschränkung der Session-Größe auf ca. 4-8 kB vor. Allerdings hängt die Performanz eines Serialisierungsvorgangs nicht unbedingt von der Größe des zu serialisierenden Objekts ab, sondern vielmehr von seiner Komplexität. Baumstrukturen lassen sich deutlich langsamer serialisieren als deutlich größere, aber dafür einfache Objekte. Diese Tatsache lässt sich deutlich mit der Implementierung einer eigenen Serialisierungsstrategie entschärfen. Anstatt das `java.io.Serializable` muss das `java.io.Externalizable` Interface implementiert werden.

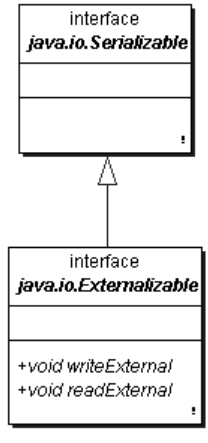


Abbildung 3.73: Das Interface `java.io.Externalizable`

Dabei müssen die Methoden `writeExternal(ObjectOutput output)` und `readExternal(ObjectInput input)` von dem Entwickler richtig ausimplementiert werden. Dabei ist es dem Entwickler überlassen, welche Daten persistent abgelegt werden.

Bei der »Session-Affinität« handelt es sich um eine besondere Konfiguration des Dispatchers oder Load-Balancers. Anhand der im Request enthaltenen Informationen erkennt dieser, um welche Session es sich handelt. Anstatt die Requests wahllos zu verteilen, landen die Requests eines Benutzers (oder einer Session) auf dem gleichen Knoten. Diese Optimierung erlaubt das Caching der Session-Daten auf einem Knoten. Bei einem schreibenden Zugriff jedoch muss die Session weiterhin in einer zentralen Datenbank abgelegt werden.

Neben der Ablage der Session-Daten in einer zentralen Datenbank, wird oft eine andere Strategie, die sog. »In-Memory-Replizierung« verwendet. Dabei befindet sich die Session, einem »Masterknoten«. Die »Session-Affinität« stellt sicher, dass Requests innerhalb einer Session immer auf dem gleichen Serverknoten landen. Die Zustände der Session werden zusätzlich auf einem anderen Knoten gehalten (sog. Replika). Bei einem Ausfall des Hauptknotens wird automatisch der Benutzer auf den »Replika-Knoten« weitergeleitet. Aus den Daten der Replika wird eine neue Session erzeugt. Dieser Vorgang bleibt für den Internetbenutzer vollkommen transparent. Allerdings werden an die Session-Daten ähnliche Anforderungen gestellt – sie müssen serialisierbar sein. Diese Anforderung kann mit der Übertragung der Daten von dem Master- auf den Replika-Knoten erklärt werden. Diese Art der Session-Replizierung kommt ohne eine Datenbank aus und kann (hängt von Hersteller ab) eine bessere Performance aufweisen als die Session-Persistenz.

EJB-Container

Leider wirkt sich eine Cluster-Umgebung nicht nur auf die Funktionalität der Web-, sondern auch auf das Verhalten der EJB-Komponenten aus. Meistens wird die Cluster-Fähigkeit der EJBs bereits in ihren Stubs implementiert. Dabei wird oft von »Smart-Stubs« oder »Cluster-Aware-Stubs« gesprochen. Ein clusterfähiger Stub kennt die IP-Adressen oder DNS-Namen aller beteiligten Server. Zu den Aufgaben der Stubs gehört nicht nur die Bereitstellung der Ausfallsicherheit, sondern auch die Übernahme des Load-Balancings.

Am einfachsten lassen sich die Stateless SessionBeans clustern. Aufgrund der Zustandslosigkeit dieser Komponenten sind die einzelnen Instanzen austauschbar. Ein Client, der gerade mit einer Instanz kommuniziert, könnte beim nächsten Methodenaufruf bereits eine andere Instanz zugewiesen bekommen. Da der Zustand der SLSB nicht repliziert werden muss, können die einzelnen Methodenaufrufe auf die Clusterknoten verteilt werden. Bei diesem Load-Balancing ist nicht nur die Erhöhung der Ausfallsicherheit, sondern auch der Performance möglich. Allerdings ist die Cluste-

rung dieser zustandslosen Komponenten für den Entwickler auch nicht ganz transparent. Es muss unterschieden werden, ob die Methoden der Bean »idempotent« sind oder nicht. Die Aufrufe einer idempotenten Methode mit gleichen Parametern ergeben immer das gleiche Ergebnis. Ferner sind die »Auswirkungen« einer idempotenten Methode nie persistent. Ein idempotenter Aufruf kann mit einer SQL »Select« Abfrage, oder dem Aufruf `Math.sin(float a)` verglichen werden. Die Unterscheidung, ob eine Methode idempotent ist oder nicht, ist besonders wichtig für das Verhalten des Clusters bei einem Absturz der gerade benutzten Instanz während der Abarbeitung einer Methode. Bei einer nicht-idempotenten Methode kann einfach auf die Instanz des Nachbarsknoten ausgewichen und die Methode nochmals aufgerufen werden. Der Ausfall einer Stateless SessionBean wäre für den EJB-Client vollkommen transparent. Bei einer idempotenten Methode ist dieses Verhalten fatal für die Anwendung. Es könnte passieren, dass ein Datensatz zweimal bei einem Aufruf der Methode in die Datenbank hinzugefügt wird. Einmal bei dem regulären Aufruf (vor dem Absturz) und das zweite Mal bei dem »failover« Aufruf. Die Idempotenz einer Methode ist lediglich für das Failover-Verhalten eines Applikationsservers während eines Methodenaufrufs wichtig.

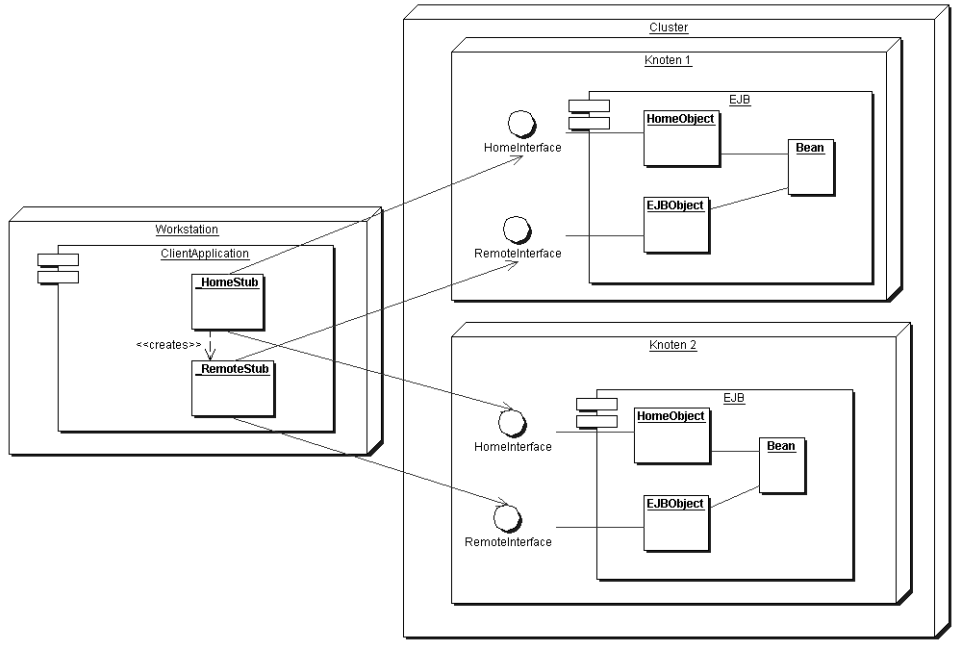


Abbildung 3.74: Das Clustering von EJBs (Skeletons wurden nicht berücksichtigt)

Die Clusterung von Stateful SessionBeans (SFSB) ist hier ein wenig aufwändiger. Bei jeder Änderung der Attribute einer SFSB muss der Zustand mit allen beteiligten Knoten repliziert werden. Meistens wird auch hier die »In-Memory-Replication« verwendet. Dabei wird immer nur der gleiche Knoten bzw. die gleiche Instanz einer SFSB angesprochen. Der Zustand wird nach jeder Transaktion mit dem Sekundärknoten abgeglichen. Sollte der Primärknoten ausfallen, landet der nächste Methodenaufruf automatisch auf dem Sekundärknoten. Dieser Vorgang ist für den EJB-Client vollkommen transparent. Da hier auch Daten über das Netzwerk verschickt werden müssen, sollten auch alle Attribute einer SFSB serialisierbar sein. Diese Anforderungen hängen natürlich von dem jeweiligen Applikationsserverhersteller ab. Leider hat die Ausfallsicherheit der SFSB im Cluster meistens einige Einschränkungen. Das Problem: Die Daten werden meistens nach einem »commit« repliziert. Sollte der Primärknoten sofort nach dem commit, aber noch vor der Replizierung der Daten ausfallen, greift der Client auf veralteten Zustand der SLSB zu. Der Zustand stammt noch aus der Zeit vor dem letzten »commit«.

Ferner könnte es passieren, dass der Primärknoten sofort nach der Erzeugung und noch bevor der Zustand repliziert werden könnte, ausfällt. In dem Fall versucht der Client mit einer nicht existierenden SFSB auf dem Sekundärknoten zu kommunizieren …

Bei der Benutzung der EntityBeans muss sich vor allem der Deployer mit dem Cluster auseinandersetzen. Da die EntityBeans kein Konversationsgedächtnis besitzen, werden sie auf ähnliche Art und Weise geclustert wie die Stateless SessionBeans. Da eine EntityBean Instanz mit einem PK jetzt sich gleichzeitig auf verschiedenen Clusterknoten befinden kann, muss die Konsistenz der Daten gewährleistet werden. Die Clusterung der Entities hat oft die Abschaltung des Caches zu Folge. Bei dem Cache handelt es sich um EntityBean Instanzen, die bereits mit dem PK assoziiert sind – sie können also jederzeit von dem Client verwendet werden, ohne den Aufruf der Methode `ejbLoad`. An dieser Stelle werden die Datenbankexperten hellhörig – die Datenbank cached doch selbst! Man darf allerdings nicht vergessen, dass in der Datenbank nur einfache Datentypen gecached werden, diese bei jedem Zugriff zuerst in Objekte konvertiert werden müssen. Die Erzeugung von Objekten ist allerdings im Verhältnis sehr teuer – der Aufruf `new` ist wesentlich langsamer als ein gewöhnlicher Methodenaufruf. Das Caching von gültigen Entity-Instanzen kann grosse Performance-Vorteile mit sich bringen.

Bisher war das Caching nur möglich, wenn der Applikationsserver als einziger Client der Datenbank agiert hat. Für das Caching im Cluster reicht diese Einstellung leider nicht aus. Da die Caches nur in Ausnahmefällen von dem Applikationsserver repliziert werden (herstellerabhängig), lassen sich diese betreiben, wenn nur lesend auf die Datenbank zugegriffen wird. Ein schreibender Zugriff könnte die Konsistenz der Daten gefährden, da sich die Daten der einzelnen Clusterknoten unterscheiden könnten.

3.16.3 Fazit

Die Ausfallsicherheit des Applikationsservers im Cluster wird durch die redundante Auslegung von kritischen Ressourcen erreicht. Genau an dieser Stelle liegt das Problem. Bisher liefen die alle Anwendungen innerhalb einer JVM. Man konnte sich auf die Existenz gemeinsamer Speicherbereiche verlassen. Dadurch konnte die Funktion von z. B. Singletons sichergestellt werden.

In einer Clusterumgebung läuft der Applikationsserver gleichzeitig auf mehreren JVMs ab. Ressourcen die bisher einmal vorhanden waren, können in einem Cluster-Verbund mehrfach existieren (pro Knoten einmal). Besondere Vorsicht ist bei der Benutzung von Singletons geboten. Diese können in einer J2EE-Umgebung zu Dateninkonsistenzen führen. Aus diesem Grund sollte man lediglich lesend auf diese zugreifen.

Neben der Implementierung von eigenen Komponenten sollte man die Resource-Factories des Applikationsservers näher betrachten und auch testen. Eine nicht clusterfähige Instanz eines Datenbank-Connection-Pools könnte sich mehrmals mit der Datenbank verbinden (Anzahl der Knoten * Anzahl der eingestellten Connections). Aber nicht nur JDBC-Ressourcen, sondern auch JMS-, und JCA-Factories sollten überprüft werden.

3.17 Berechtigungen auf dem Applikationsserver

3.17.1 Problemstellung

Wie werden auf dem Applikationsserver Berechtigungen vergeben?

Darf ein Applikationsserver Dateien lesen? Socket-Verbindungen aufbauen?

3.17.2 Technischer Hintergrund

Die verbreitetste Möglichkeit, auf dem Server für den registrierten Anwender Berechtigungen zu vergeben, ist der Ansatz über den J2EE-Deskriptor. Darin können Berechtigungen in Abhängigkeit vom authentifizierten Principal auf Methodenebene vergeben werden. Der Aufruf einer Methode einer J2EE-Komponente wird stets vom Container kontrolliert, der damit die volle Kontrolle über die Ausführung der Methode hat.

J2EE-Komponenten haben auf Grund des Selbstschutzes des Applikationsservers nur eingeschränkte Rechte. So darf bekanntermaßen eine EJB keine Threads starten oder beliebige Dateien lesen und schreiben. Diese Beschränkung ist in der J2EE-Spezifikation ausführlich begründet und aufgelistet. Trotzdem kann jede EJB noch durch den Aufbau von Socket-Verbindungen Kontakt mit Prozessen erlangen, die sich außerhalb

der Sicherheitsumgebung des Applikationsservers befinden. Dies ist beispielsweise dann notwendig, wenn die Bean eine Datenbankverbindung aufbaut oder Kontakt mit einem RMI-Server aufnehmen muss.

Web-Komponenten haben etwas größere Berechtigungen, so den Zugriff auf das Dateisystem, das Starten eigener Threads und das Laden nativer Bibliotheken.

Hat der Applikationsserver Zugriff auf kritische Ressourcen des Unternehmens, muss der Deployer die Methoden, die den Zugriff durchführen, kennen und im Deskriptor deklarativ den Aufruf nur bestimmten Benutzergruppen erlauben. Dies ist bei Eigenentwicklungen zumindest aufwändig, bei zugekauften oder extern entwickelten Programmteilen nur durch Analyse des Quellcodes ersichtlich und somit faktisch unmöglich zu garantieren.

Es ist notwendig, die Aktionen, für die der Applikationsserver selber Berechtigungen hat, in diesem Zusammenhang feiner zu definieren. Dafür ist jedoch die ausgefeilte Sicherheits-Architektur des JRE bereits vollständig geeignet. Die Virtuellen Maschinen eines Applikationsservers werden mit einem aktivierten Security Manager gestartet, der mit so genannten Policy-Dateien konfiguriert werden kann. Diese Policy- oder »Berechtigungs«-Dateien liegen für den Security-Provider von Sun, der der Laufzeitumgebung beigefügt wurde, in Form von einfach lesbaren Textdateien vor. Berechtigungen werden innerhalb der Policy-Datei an Hand von drei unterschiedlichen Kriterien vergeben:

- Die »codeBase«, also die URL, von der die Klasse geladen wurde. So kann in einer einfachen Sicherheitseinstellung lokal installierten Klassen der volle Zugriff auf die Ressourcen erlaubt werden, allen über das Netz geladenen Bytecodes wird der Zugriff verboten.
- Bytecode kann signiert werden, indem der JAR-Datei ein Schlüssel bzw. ein Zertifikat beigefügt wird. Signiertem Bytecode können dann in Abhängigkeit vom Unterzeichner weitere Rechte eingeräumt werden.
- Als dritte Möglichkeit ist ab der Version 1.4 eine Berechtigung an Hand des java.security.Principal, also der Identität des Aufrufenden, vorgesehen. Dies ist die Grundlage des bereits besprochenen JAAS.

Alle Klassen der Java-Laufzeitumgebung prüfen vor Ausführung potenziell sicherheitsrelevanter Aktionen mit Hilfe der Klasse `java.security.AccessController`, ob sie überhaupt die notwendigen Berechtigungen dafür besitzen. Berechtigungen sind als eine erweiterbare Klassenhierarchie modelliert:

Eine Policy-Datei ordnet einer CodeBase, einer Signatur oder einem Principal eine oder auch mehrere Berechtigungen zu. Dazu kann ein beliebiger Editor oder auch das Policy-Tool der Java Laufzeitumgebung dienen:

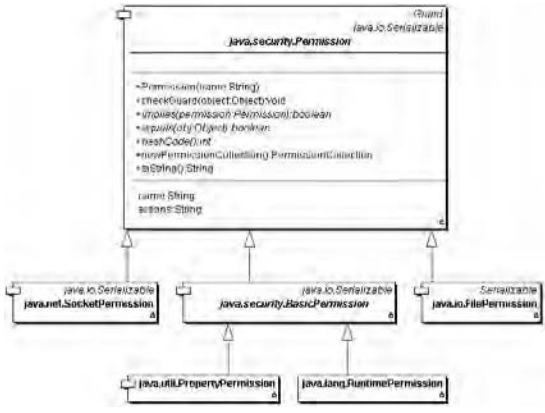


Abbildung 3.75: Die Hierarchie der Permission-Klassen

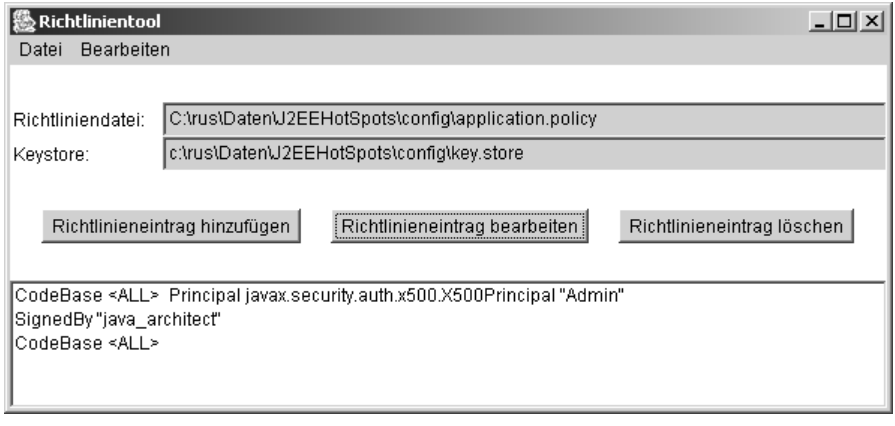


Abbildung 3.76: Das Policy-Tool zur Erstellung von Policy-Dateien

Die damit erstellte Policy-Datei erhält die Vollberechtigung für alle als Administrator authentifizierten Benutzer, eine Lese- und Schreibberechtigung für alle Klassen, die vom Alias »java_architect« signiert wurden. Alle anderen Klassen haben nur Lese- und Schreibberechtigung für das Temp-Verzeichnis. Dies kann sich der Anwender auch ohne Verwendung des Policy-Tool durch Betrachten der generierten Datei überzeugen:

```
keystore "c:\rus\Daten\J2EEHotspots\config\key.store";
grant      principal javax.security.auth.x500.X500Principal "Admin" {
  permission java.security.AllPermission;
 };
grant signedBy "java_architect" {
  permission java.io.FilePermission "<<ALL FILES>>", "read, read, write, delete,
execute";
```

```
};
grant {
  permission java.io.FilePermission "file:/c:/temp/-", "write";
  permission java.io.FilePermission "file:/c:/temp/-", "read";
};
```

Der Keystore mit den Schlüsseln der Aliasse wird hier in der Datei key.store gesucht. Zur Verwaltung eines Keystores (Anlegen, Löschen von Aliassen, Generierung von Schlüsseln und Zertifikaten, Import und Export) kann das »keytool« des JDK verwendet werden.

Diese Policy-Datei wird beim Aufruf der Virtuellen Maschine als weitere Umgebungsvariable gesetzt:

```
java -Djava.security.manger -Djava.security.policy=application.policy Application
```

3.17.3 Lösung

Die »Programmierrestriktionen für J2EE-Komponenten« werden bei der Referenzimplementierung von Sun durch die folgende Policy-Datei (server.policy im lib\security-Verzeichnis) umgesetzt:

```
// Standard extensions get all permissions by default

grant codeBase "file:${java.home}/lib/ext/-" {
    permission java.security.AllPermission;
 };

grant codeBase "file:${java.home}/../lib/tools.jar" {
    permission java.security.AllPermission;
 };

grant codeBase "file:${com.sun.enterprise.home}/lib/-" {
    permission java.security.AllPermission;
 };

grant codeBase "file:${jms.home}/classes/" {
    permission java.security.AllPermission;
 };

// Drivers and other system classes should be stored in this
// code base.
 grant codeBase "file:${com.sun.enterprise.home}/lib/system/-" {
    permission java.security.AllPermission;
 };

// additional permissions for EJBs
grant codeBase "file:${com.sun.enterprise.home}/ejb_impls/-" {
```

```
    permission java.lang.RuntimePermission "queuePrintJob";
    permission java.io.FilePermission "${com.sun.enterprise.home}${/}repository${/
}-", "read";
};

// additional permissions for servlets
grant codeBase "file:${com.sun.enterprise.home}/public_html/-" {
    permission java.lang.RuntimePermission "loadLibrary.*";
    permission java.lang.RuntimePermission "accessClassInPackage.*";
    permission java.lang.RuntimePermission "queuePrintJob";
    permission java.lang.RuntimePermission "modifyThreadGroup";
    permission java.io.FilePermission "<<ALL FILES>>", "read,write";
};

// additional permissions for standalone resource adapters
grant codeBase "file:${com.sun.enterprise.home}/connector/adapters/-" {
    permission javax.security.auth.PrivateCredentialPermission "* * \"*\"",
"read";
    permission java.io.FilePermission "${com.sun.enterprise.home}${/}logs${/}-",
"read,write";
};

// permissions for other classes
grant codeBase "file:${com.sun.enterprise.home}/repository/-" {
    permission java.lang.RuntimePermission "loadLibrary.*";
    permission java.lang.RuntimePermission "accessClassInPackage.*";
    permission java.lang.RuntimePermission "queuePrintJob";
    permission java.lang.RuntimePermission "modifyThreadGroup";
    permission java.io.FilePermission "<<ALL FILES>>", "read,write";
    permission javax.security.auth.PrivateCredentialPermission "* * \"*\"",
"read";
};

// permissions for default domain
grant {
    permission java.net.SocketPermission "*", "connect";
    permission java.util.PropertyPermission "*", "read";
    // workaround missing doPrivileged blocks in javamail
    permission java.io.FilePermission "${com.sun.enterprise.home}${/}lib${/
}j2ee.jar", "read";

    permission javax.security.auth.PrivateCredentialPermission
"javax.resource.spi.security.PasswordCredential * \"*\"", "read";
    permission javax.security.auth.PrivateCredentialPermission
"javax.resource.spi.security.GenericCredential * \"*\"", "read";

};
```

3.17.4 Praxis

Jedes installierte Bean kann eine direkte JDBC-Connection durch Einladen des Treibers und Registrierung beim DriverManager erhalten:

```
try{
 Class.forName("COM.cloudscape.core.RmiJdbcDriver");
 Connection lCon = DriverManager.getConnection("jdbc:cloudscape:rmi://localhost/
Test;create=true");
Statement lStatement = lCon.createStatement();
lStatement.executeUpdate("create table TEST (col_id varchar(255))")
 }
 catch(Exception pException){
  pException.printStackTrace();
}
```

Hier versucht das Enterprise Bean durch Ansprechen eines RMI-fähigen Typ 4 JDBC-Treibers der Cloudscape-Datenbank eine Tabelle anzulegen. Dies ist ohne weiteres möglich, da innerhalb der Server-Policy die Erlaubnis zur Verbindung mit Sockets explizit für alle Klassen gegeben wurde.

Um dies sicher zu verhindern genügt es, diese Erlaubnis nicht zu gewähren, was sich zur Demonstration am einfachsten durch Kommentieren der folgenden Zeile und anschließenden Neustart des Applikationsservers erreichen lässt:

```
grant {
    //permission java.net.SocketPermission "*", "connect";
    permission java.util.PropertyPermission "*", "read";
…
};
```

Jetzt darf ein installiertes Enterprise Bean keine Socket-Verbindungen mehr aufbauen, was sich in diesem Beispiel beim Aufruf der Methode in einer Exception in der Konsole des Applikationsservers äußert:

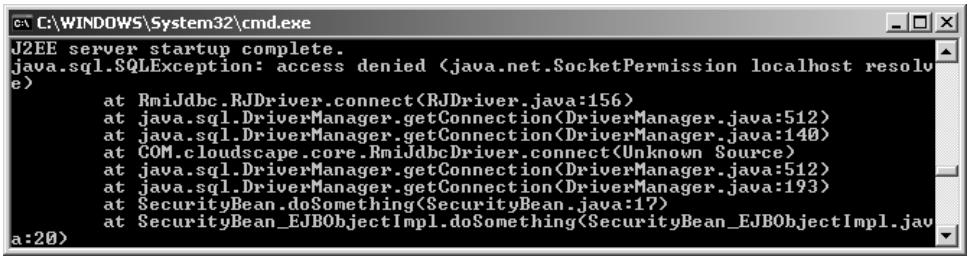


Abbildung 3.77: Die nicht gewährte SocketPermission führt beim Versuch des Aufbaus einer SQL-Connection zu einer Exception.

Diese Einstellung kann anhand der Policy-Datei-Syntax des Security Providers von Sun natürlich noch wesentlich feiner eingestellt werden, so dass der Zugriff auf bestimmte Sockets möglich ist oder aber an Hand der Signatur oder CodeBase des Archivs die Erlaubnis wieder gegeben werden kann.

Somit kann durch das Anpassen der Server-Policy der Applikationsserver in die Sicherheits-Architektur des Unternehmens integriert werden.

Der Weblogic wird zwar mit einer Berechtigungsdatei gestartet. Das Installationsverzeichnis des Applikationsserver enthält im Unterverzeichnis »lib« die Datei weblogic.policy, die zwar etwas andere Einträge enthält, jedoch im Wesentlichen die gleichen Berechtigungen enthält. Die Konfiguration von Berechtigungen für Ressourcen ist jedoch eleganter gelöst: Jeder Ressource kann eine individuelle Berechtigungsdatei zugeordnet werden:

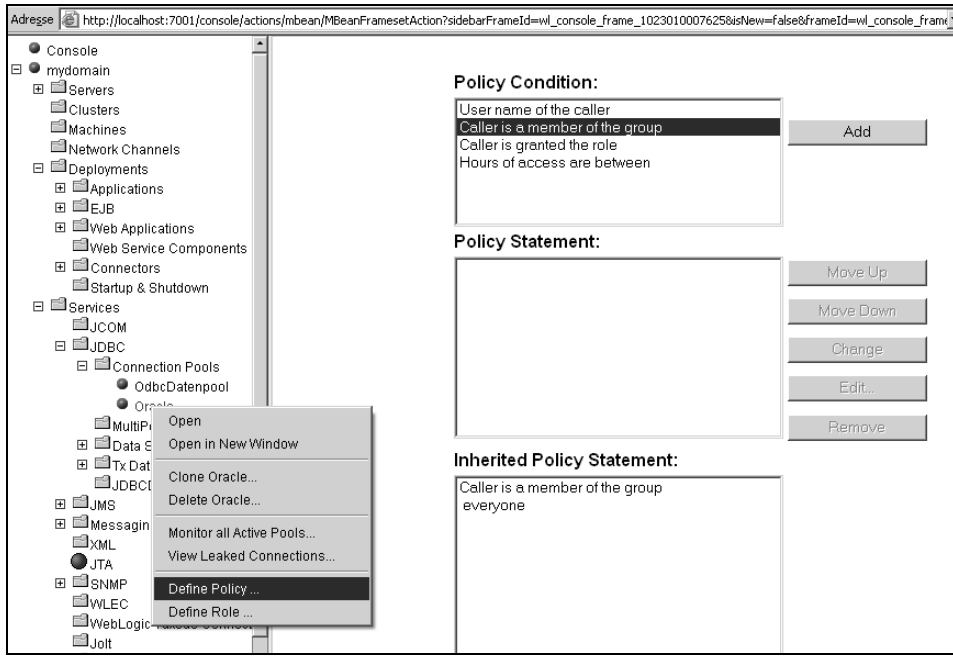


Abbildung 3.78: Das Erstellen von Berechtigungen für den BEA Weblogic 7.0 am Beispiel eines Connection Pools

Der JBoss-Applikationsserver legt seine Sicherheitseinstellungen komplett in die Hand des Administrators. Die im conf-Verzeichnis des Servers abgelegte Datei server.policy enthält als einzigen Eintrag die Gewährung der `java.security.AllPermission`. Hier ist es alleinige Aufgabe des Administrators, nur die benötigten Berechtigungen zu definieren. Insbesondere dürfen in der Standard-Installation des JBoss EnterpriseBeans im Gegensatz zur J2EE-Spezifikation »alles«.

3.18 Zugriff auf RMI-Server

3.18.1 Problemstellung

Wie kann eine EnterpriseBean auf einen externen RMI-Server zugreifen?

3.18.2 Technischer Hintergrund

Eine EnterpriseBean-Komponente kann Zugriff auf andere RMI-Server benötigen. Dies ist beispielsweise dann der Fall, wenn Altsysteme mit nativen Bibliotheken mit einem RMI-Wrapper umgeben werden und die Bean die nativen Methoden aufrufen will. Nachdem die J2EE-Spezifikation Java RMI als Protokoll verlangt, kann eine EnterpriseBean als »normaler« RMI-Client aufgefasst werden.

3.18.3 Lösung

Die EnterpriseBean holt sich eine entfernte Referenz über JNDI. Das Bean baut damit einen einen eigenen JNDI-Kontext auf den RMI-Server auf. Die Angabe der InitialContextFactory und der URL erfolgen im Deskriptor.

3.18.4 Praxis

Die `InitialContextFactory` auf die rmiregistry ist die Klasse `com.sun.jndi.rmi.registry.RegistryContextFactory`, die in der Klassenbibliothek der J2SE enthalten ist. Der Aufbau der Verbindung erfolgt bei der Initialisierung der Bean:

```
try{
  InitialContext serverContext = new InitialContext();
  Object obj = serverContext.lookup("java:comp/env/NamingFactory");
  if (obj != null){
    namingFactory = obj.toString();
  }
  obj = serverContext.lookup("java:comp/env/NamingUrl");
  if (obj != null){
    namingUrl = obj.toString();
  }
 }
  catch(Exception pException){
    throw new EJBException(pException);
}
}
private Context getRemoteContext(){
try{
  if (remoteContext == null){
   if (namingFactory != null && namingFactory.length() > 0){
    Hashtable lTable = new Hashtable();
```

```
      lTable.put(Context.INITIAL_CONTEXT_FACTORY, namingFactory);
      lTable.put(Context.PROVIDER_URL, namingUrl);
      remoteContext = new InitialContext(lTable);
     }
     else{
      remoteContext = new InitialContext();
     }
    }
    return remoteContext;
   }
   catch(Exception pException){
    throw new EJBException(pException);
  }
  }
```

Die rmiregistry kann nun über einen normalen JNDI-Lookup durchsucht werden.

4 Die Integrationsschicht

4.1 MQSeries als JMS-Provider

4.1.1 Problemstellung

Wie können Java-Programme in eine MQSeries-Architektur integriert werden?

Wie greift ein Nicht-Java-Client auf die JMS-Messages zu?

Wie sendet ein Nicht-Java-Client JMS-Messages?

4.1.2 Technischer Hintergrund

IBM MQSeries ist ein in der Industrie weit verbreitetes und stabil laufendes Messaging System, das eine Point-to-Point Queue verwaltet. Das Produkt integriert eine Vielzahl von Enterprise Information Systemen und ist somit ein mächtiges Werkzeug zur Enterprise Application Integration. So sind Anbindungen an SAP, Lotus Domino und CICS enthalten. Ab Version 5 ist eine eigene Implementierung des *Java Messaging Services* (JMS) enthalten.

4.1.3 Lösung

Die Installation von MQSeries enthält eine Reihe von Java-Archiven, die die direkte Kommunikation mit der Queue abbilden:

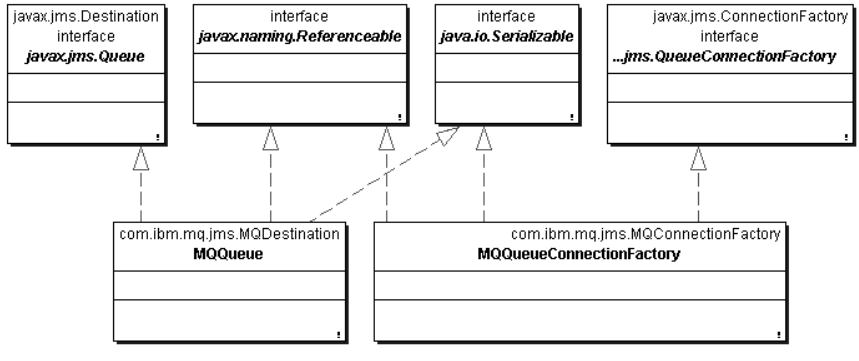


Abbildung 4.1: MQQueue und MQQueueConnectionFactory

Daneben sind natürlich auch Implementierungen des `javax.jms.Session`-Interfaces sowie `javax.jms.QueueSender` und `QueueReceiver` vorhanden.

Java-JMS-Clients können `MQQueueConnectionFactory` und `MQQueue` durch Instanzierung und Setzen der Eigenschaften direkt verwenden. Eine externe Konfiguration und Ablage in einem JNDI-Namensraum ist natürlich wesentlich flexibler. MQSeries enthält ein Administrationsprogramm namens »JMSAdmin«, das diese Aufgabe übernimmt. Dieses Programm ist jedoch leider beschränkt auf wenige JNDI-Provider (File-System, LDAP-Server und Websphere Applikationsserver), eine allgemeinere Version zeigt exemplarisch die folgende Implementierung:

```
package com.hotspots.client.mq_series;
import com.ibm.mq.jms.*;
import javax.naming.*;
public class MQJMSConfigurator{
 public static void main(String[] args) throws Exception{
 String defaultHostname = "localhost";
 int  defaultPort = 1415;
 String defaultQueueManager = "Demo_Warteschlangen_Manager";
 String defaultQueue = "hotspot";
 String jndiNameConnectionFactory = "MQConnectionFactory";
 String jndiNameQueue = "mq_hotspot";
 MQQueueConnectionFactory factory = new MQQueueConnectionFactory();
factory.setQueueManager(defaultQueueManager);
factory.setHostName(defaultHostname);
factory.setPort(defaultPort);
MQQueue queue = new MQQueue(defaultQueue);
InitialContext ctx = new InitialContext();
ctx.rebind(jndiNameConnectionFactory, factory);
ctx.rebind(jndiNameQueue, queue);
 }
}
```

Der Administrator muss für diese Konfiguration MQ-Series entsprechend einstellen (siehe Abbildung 4.2).

Die Factory kann durch Setzen von Properties MQ-spezifisch konfiguriert werden. Auch Methoden der MQMessage-Implementierungen der javax-jms.Message-Interfaces können hierzu benutzt werden.

An diese Warteschlange kann nun ein JMS-Client sofort beliebige Nachrichten schicken:

```
 String jndiNameConnectionFactory = "MQConnectionFactory";
 String jndiNameQueue = "mq_hotspot";
 try {
  jndiContext = new InitialContext();
  queueConnectionFactory = (QueueConnectionFactory)
jndiContext.lookup(jndiNameConnectionFactory);
  queue = (Queue) jndiContext.lookup(jndiNameQueue);
  queueConnection =
```

```
    queueConnectionFactory.createQueueConnection();
      queueSession =
    queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
      queueSender = queueSession.createSender(queue);
      message = queueSession.createTextMessage();
      message.setText("A Java-Hotspot using MQSeries");
      message.setStringProperty("Key", "Value");
      queueSender.send(message);
     }
```

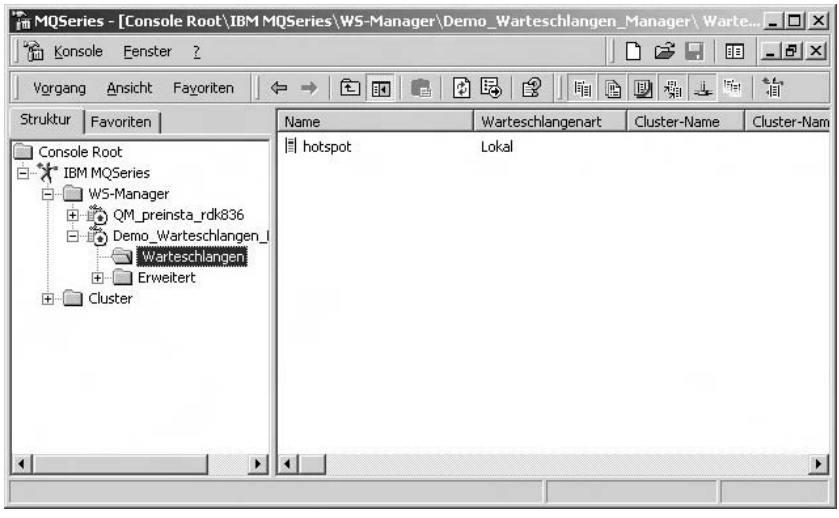


Abbildung 4.2: Einstellungen in der MQSeries-Administration

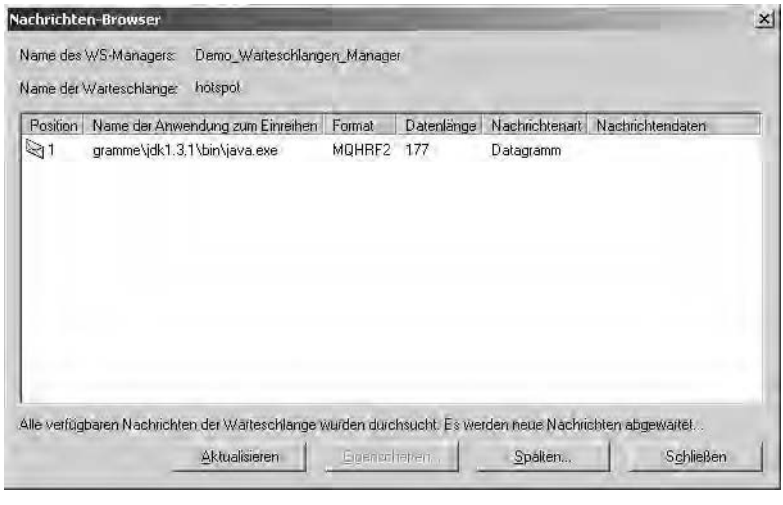


Abbildung 4.3: Die angekommene javax.jms.TextMessage

Die Nachricht wird bis zur ihrer Abholung in der Warteschlange gehalten (siehe Abbildung 4.3).

Auch ein nativer C-Client kann natürlich Nachrichten senden. Dazu müssen nur die MQSeries-Bibliotheken eingebunden werden. Der vollständige Quellcode des aufgerufenen Programms `NativeMQSender` ist auf der Begleit-CD enthalten:

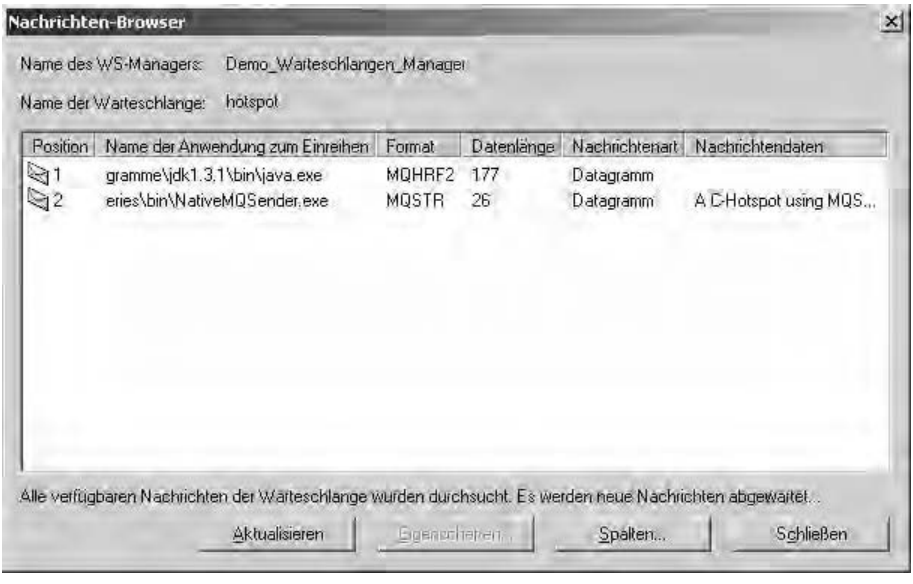


Abbildung 4.4: Eine javax.jms.TextMessage und eine einfache MQ-Message

Zu beachten sind hier die unterschiedlichen Nachrichten-Formate. Eine MQSeries-Nachricht besteht aus einem Message Descriptor und dem Message Body mit den darin enthaltenen Daten. JMS-spezifische Informationen wie übertragene Objekte und Properties werden in einem gesondert strukturierten Bereich mit einem Header (Typ RF_HEADER_2) und einem oder mehreren Datenblöcken übertragen. Jeder Datenblock ist mit einfachen XML-Tags strukturiert:

```
<mcd><Msd>jms_text</Msd></mcd>
<jms>
<Dst>queue://Demo_Warteschlangen_Manager/hotspot</Dst></jms>
<usr><Key>Value</Key></usr>
A Java-Hotspot using MQSeries
```

Die Bedeutung der drei Umgebungen ist evident: `<mcd>` enthält beschreibende Elemente der Nachricht, `<jms>` die JMS-Properties sowie die MQ-Erweiterungen, `<usr>` hält JMS-Properties.

Name	Java-Typ	Umgebung	Name der Property	C-Typ/Wert
JMSDestination	Destination	jms	Dst	string
JMSExpiration	long	jms	Exp	i8
JMSPriority	int	jms	Pri	i4
JMSDeliveryMode	int	jms	Dlv	i4
JMSCorrelationID	String	jms	Cid	string
JMSReplyTo	Destination	jms	Rto	string
JMSType	String	mcd	Type	string
JMSXGroupID	String	jms	Gid	string
JMSXGroupSeq	int	jms	Seq	i4
xxx(frei definierbar)	beliebig	usr.xxx	xxx	any
		mcd	Msd	jms_none jms_text jms_bytes jms_map jms_stream jms_object

Tabelle 4.1: Die Umgebungen des Formats MQHRF2

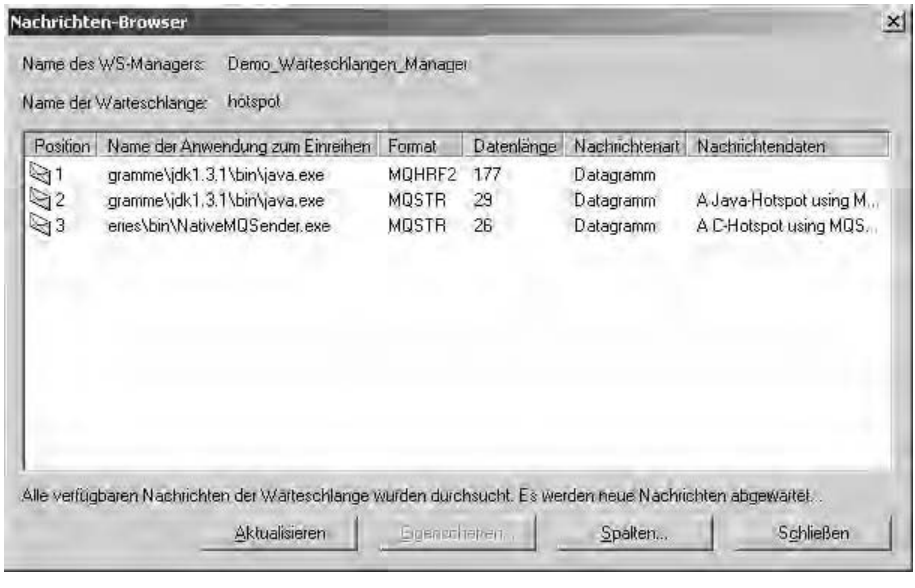


Abbildung 4.5: Eine javax.jms.TextMessage übertragen mit der Option JMSC.MQJMS_CLIENT_NONJMS_MQ

Ein nativer MQ-Client kann nun direkt mit diesem Overhead nichts anfangen. Die Java-Message enthält über 100 Bytes mehr als die C-Nachricht. Um dieses Problem zu umgehen, kann:

- Der Java-Client die Erzeugung des MQHRF2-Datenblocks unterdrücken. Dies erfolgt in der Konfiguration der MQDestination durch die Anweisung:

```
queue.setTargetClient(JMSC.MQJMS_CLIENT_NONJMS_MQ);
```

  Damit können aber keine Message-Properties übertragen werden.

- der C-Client den Datenbereich auswerten. Dazu definiert die MQ-Bibliothek die Struktur »MQRFH2«. Diese Auswertung erfordert aber in der Regel eine Änderung bereits vorhandener Programme und ist deshalb nicht immer möglich.

4.2 Ansprechen nativer Methoden aus einer J2EE-Komponente

4.2.1 Problemstellung

Können J2EE-Komponenten native Bibliotheken einbinden?

4.2.2 Technischer Hintergrund

- Das *Java Native Interface* (JNI) bildet eine plattform-unabhängige Schnittstelle für den Zugriff auf native Bibliotheken. Jedoch führt das Einbinden solcher Bibliotheken innerhalb eines Java Servers zu einigen Problemen:

- Die Virtuelle Maschine kann bei nativen Methodenaufrufen keinerlei Laufzeitfehler abfangen.

- Die Java-Sicherheitsarchitektur greift nicht.

Die J2EE-Spezifikation erlaubt deshalb für EnterpriseBeans kein Ansprechen von nativen Methoden. Genauer: Eine EnterpriseBean darf keine nativen Bibliotheken laden, dies ist »alleinige Aufgabe des Containers«.

4.2.3 Lösung

Ist es für eine Applikation wirklich entscheidend, dass eine EnterpriseBean direkt native Methoden anspricht, so kann die Erlaubnis zum Laden der Bibliothek vom Administrator erteilt werden. Für die Referenz-Implementierung von Sun sind dafür nur zwei Modifikationen durchzuführen:

- In der Server-Policy wird der Eintrag

```
permission java.lang.RuntimePermission "loadLibrary.*";
```

  eingeführt.

- Beim Aufruf muss die Umgebungsvariable java.library.path auf das Verzeichnis zeigen, in dem die Bibliothek vorhanden ist. Dazu wird der Aufruf geändert.

  Dann kann eine EnterpriseBean z. B. seine Remote-Methoden native deklarieren:

```
public class JNIBean implements SessionBean{
 private SessionContext cSessionContext;
 public native double doCalculation(double pSummand1, double pSummand2);
public JNIBean(){}
 public void ejbCreate(){
  try{
    System.loadLibrary("com_hotspots_java_jni_JNIBean");
  }
  catch(Throwable t){
   t.printStackTrace();
  }
 }
 //Methoden des SessionBean-Interfaces
 //...
}
```

- Dann kann die mit dem Werkzeug »`javah`« des JDK generierte Header-Datei

```
#include <jni.h>
#ifndef _Included_ com_hotspots_java_jni_JNIBean
#define _Included_ com_hotspots_java_jni_JNIBean
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_hotspots_java_jni_JNIBean
 * Method:    doCalculation
 * Signature: (DD)D
 */
JNIEXPORT jdouble JNICALL Java_ com_hotspots_java_jni_JNIBean_doCalculation
  (JNIEnv *, jobject, jdouble, jdouble);
//Restliche Methoden
```

- etwa in C++ implementiert und eine Bibliothek erstellt werden. Diese SessionBean integriert dann wie gewünscht eine native Bibliothek.

4.2.4 Praxis

Für die Referenz-Implementierung muss wie in produktiven Systemen die Berechtigung zur Verwendung der nativen Bibliotheken explizit gegeben werden. Dies erfolgt durch den Eintrag

```
permission java.lang.RuntimePermission "loadLibrary.*";
```

in der Server-Policy-Datei. Weiterhin muss beim Start der Virtuellen Maschine die Option

```
-Djava.library.path=…
```

gesetzt oder die Bibliothek in ein spezielles Verzeichnis kopiert werden.

Das Deployment des JNIBeans in den JBoss Applikationsserver gestaltet sich völlig unproblematisch, da der Server, wie in einem vorhergehenden Hotspot erwähnt, die `java.security.AllPermission` gewährt.

BEA Weblogic verwendet ein eigenes Verfahren zur Integration in den Applikationsserver: jCOM. jCOM erlaubt den Zugriff von Microsoft ActiveX-Komponenten auf Java-Objekte und andersherum. Konzepte und Verwendung von jCOM sind analog der in Kapitel 1 beschriebenen CAS-COM-Bridge von Sun.

Index