Caching in Stubs und Events mit Enterprise Java Beans bei Einsatz einer objektorientierten Datenbank

Olaf Neumann, Christoph Pohl, Katrin Franze

TU Dresden, Lehrstuhl Rechnernetze
{neumann|pohl|kfranze}@ibdr.inf.tu-dresden.de

Abstract. In diesem Artikel soll gezeigt werden, wie mit Hilfe einer Erweiterung der EJB (Enterprise JavaBeans) - Spezifikation in [7] einfacher zu konstruierende verteilte Client-Server-Anwendungen mit weniger Netzbelastung erstellt werden können. Außerdem wird der Einsatz objektorientierter Datenbanken in EJB-Containern diskutiert. Als Anwendungsbeispiel soll ein Ausschnitt aus dem Dokumentenmodell des am Lehrstuhl Rechnernetze der TU Dresden entstandenen Teleteaching-Projektes genutzt werden.

1 Einführung

Die Einführung der EJB-Spezifikation 1.0 im April 1998 hat die Entwicklung von Middleware nachhaltig beeinflußt. Ziel dieser Spezifikation ist es, häufig auftretende Serverfunktionalität, wie z.B. Load Balancing, Transaktionen und Security Management von der eigentlichen Business-Logik zu trennen. Dieser Standard wird in Zukunft weitere Entwicklung erfahren. So soll dieser Artikel eine Anregung für mögliche Verbesserungen darstellen.

Ausgangspunkt bei der Entwicklung von EJB-Applikationen sind das Home Interface, welches Factory-Methoden zum Erzeugen, Löschen und Finden (nur bei EntityBeans) beinhaltet, das EJB-Remote Interface, das die Business-Funktionen definiert und das Bean, das entweder ein SessionBean (nur für eine Sitzung) oder ein EntityBean (persistentes Bean) darstellt. Mittels spezieller Werkzeuge werden die Remote- und Homeklassen erzeugt und entsprechende Stub- und Skeletonklassen generiert. Die Speicherung von EntityBeans kann entweder vom EntityBean selbst oder vom Container eines Werkzeugeanbieters vorgenommen werden. Wird die Speicherung vom Container vorgenommen, so implementiert der Container die Factory-Methoden selbst. An dieser Stelle sollen die Probleme diskutiert werden, die beim Einsatz objektorientierter Datenbanken entstehen.

Die Clients greifen immer über das Home- oder das EJB-Remote Interface auf das Bean zu. Jeder Methodenaufruf erfolgt somit über eine Netzwerkverbindung. Dieser Artikel stellt dafür ein Proxy-Konzept vor, was gleichzeitig für die Aktualisierung der Proxies bei Änderung auf anderen Clients sorgt. Grundsätzlich werden alle zusätzlichen Informationen, die zur Generierung des Containers und der Stub- und Skeletonklassen notwendig sind, im Deployment Descriptor abgelegt. Bei unseren

vorzunehmenden Erweiterungen werden wir diesen Mechanismus zusammen mit Designpattern verwenden, wie man sie bei JavaBeans vorfindet.

2 Problemstellung

Das Zielsystem für nachfolgend beschriebene Lösung ist JaTeK (Java Based Teleteaching Kit). Diese stellt eine vollständig in Java implementierte Lernumgebung dar, deren abgebildetete Szenarien des Spektrum des asynchronen Selbstlernen bis zur asynchronen und synchronen Gruppenarbeit umfassen.

JaTeK selbst ist als 3-Schichten-Architektur realisiert (siehe Fig. 1), wobei eine objektorientierte Datenbank (OODB) die dritte Schicht darstellt, die zweite der JaTeK-Server und die erste die verteilten JaTeK-Viewer.

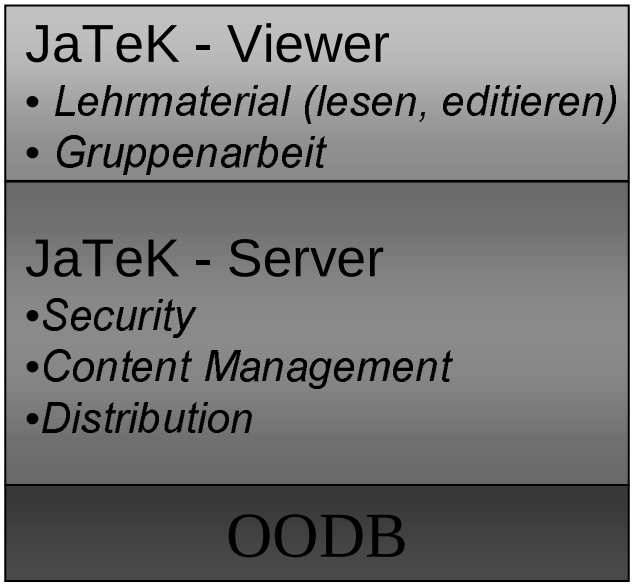

Fig. 1 - 3-Schichten Architektur von JaTeK

In dem zu betrachtenden Beispiel (siehe Fig. 2) wird davon ausgegangen, daß die Clients u.a. eine Baumstruktur von Kapiteln innerhalb eines Kurses mittels der Swing-Klasse JTree anzeigen. Der JTree greift auf ein eigenes TreeModel zu [8]. Das TreeModel benutzt Methoden des Business-Interfaces, um die Kapitelstruktur auf dem Server abzufragen. Der Server verwaltet die Kapitelstruktur und andere Security-relevanten Informationen. In der OODB werden die einzelnen Kapitel in eigenen Objekten gespeichert. Zum Zeitpunkt, wenn der JTree auf dem Client neu gezeichnet wird, werden Daten von dem TreeModel abgefragt. Das TreeModel seinerseits fragt zu jedem Blattknoten wenigstens den Namen ab. Bei einer größeren Anzahl von Blattknoten kann somit der Netzwerkverkehr erheblich ansteigen. Aus diesem Grund verwendet man sogenannte Proxies, um Daten nicht immer wieder erneut abzufragen.

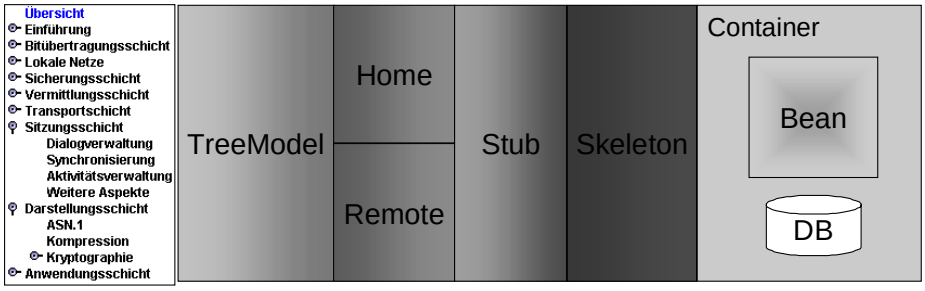

Fig. 2 - Aufbau der Komponenten (JTree, Model, Interface, Stub, Skeleton, Bean, DB)

Die Szenarien unseres Systems beinhalten, daß mehrere Benutzer an einer Struktur gleichzeitig arbeiten. Dafür müssen die Proxy-Objekte jedoch bei Änderungen aktualisiert werden. Sowohl der Anwender als auch der Entwickler sollten so wenig wie möglich mit solchen Architekturfragen belästigt werden.

Ein Ansatz dafür wäre die Integration eines Rückaufrufmechanismusses beim Client. Dieser ist jedoch in der EJB-Spezifikation nicht integriert. In [5], wie auch im Corba-Eventservice [6] wird ein Vorschlag für eine derartige Realisierung gemacht. Der Ansatz in [5] baut gegenüber unserem Konzept auf den EJB auf. Die Skalierbarkeit wird aber in diesem Ansatz nicht ausreichend berücksichtigt, da die Events seriell an die einzelnen Empfänger verschickt werden. Auch [4] kann ein Ausgangspunkt für derartige Überlegungen sein.

Die Daten auf Serverseite werden in einer objektorientierten Datenbank gespeichert. Der Einsatz einer objektorientierten Datenbank hat gezeigt, daß sich der Quelltext zur Abspeicherung von Daten etwa um 50% verringert hat gegenüber dem Einsatz einer relationalen Datenbank, da keine Typkonvertierungen mehr notwendig ist. Da die Operationen zur Speicherung der Daten fast immer dieselben sind, bietet sich das Konzept der Container Managed Persistence (CMP) von EJB an. Dabei müssen die Spezifika objektorientierter Datenbanken, wie eine einheitliche ObjectID und das Spezifizieren von Eintrittspunkten (Roots), berücksichtigt werden.

3 Lösung

Der Event-Mechanismus in JaTeK basiert auf Channels und macht so die Filterung von Events möglich. Ein Channel faßt (wie im Corba Event Service [6]) eine Untermenge aller angeschlossen Clients zusammen, die auf dieselben Daten zugreifen und über Änderungen dieser Daten informiert werden. Andere oder ähnliche Ansätze sind in [4], [5] und [3] verfolgt. Eine weitere Eigenschaft von JaTeK ist es, daß die Stubs beim Client als Proxies dienen, um Daten in einem Cache zwischenzuspeichern, und so die Performance der Zugriffe zum Server zu erhöhen. Proxies werden z.B. auch in BEA Weblogic [2] verwendet, dienen dort jedoch der Bündelung der Aufrufe über einen Socket. Beide Mechanismen, das Eventhandling und das Caching können zusammen verwendet werden, um die Entwicklung von verteilten Applikationen zu vereinfachen.

3.1 Lösungsansatz

Bei EJB können Stubs, Skeleton und der Container automatisch erzeugt werden. So könnten im Stub automatisch Attribute XXX generiert werden, die als Proxy für die JavaBeans-konformen Accessor-Methoden setXXX() und getXXX() dienen. Die Attribute erhalten eine zusätzliche Eigenschaft, die besagt, ob das Attribut:

- non cachable (Alle Methoden greifen direkt auf die Remote-Schnittstelle zu.)
- pull cachable (Das Attribut wird zwischengespeichert und per Pull-Aktion in bestimmten Zeitabständen aktualisiert. Dazu wird ein Thread verwendet. Die Accessor-Methoden müssen eine Exception bei einem inkosistenten Zustand werfen. Es müssen Methoden zum An- und Abmelden eines Event-Listeners vorhanden sein.)
- pullWait cachable: (Es wird eine Methode auf dem Server aufgerufen. Diese wird mittels wait() in den Wartezustand geschickt. Bei Veränderung des Attributs wird ein notify() ausgelöst. Dieses Vorgehen benötigt zusätzlich einen Referenzzähler für jedes Attribut, da zwischenzeitlich mehrere Events auf dem Server eingetroffen sein können. Es wird ein RMI-Mechanismus benötigt, der keine Sockets bei einem Methodenaufruf offenläßt, was zu Engpässen führen könnte. Das von BEA implementierte RMI mit Bündelung aller Aufrufe über einen Socket kann hierfür genutzt werden.)
- push cachable (Das Attribut wird zwischengespeichert und bei Änderung per Push-Aktion in den anderen Clients aktualisiert. Dazu wird eine Callback-Methode eingerichtet, die ein Event entgegennimmt. Die Accessormethoden müssen eine Exception bei einem inkosistenten Zustand werfen. Es müssen Methoden zum An- und Abmelden eines Event-Listeners vorhanden sein.)

Die Accessor-Methoden erhalten Exceptions, wenn die Aktualität bei der Ausführung der Operation nicht gewährleistet war. Außerdem können in dem Remote Interface zusätzlich sog. verteilte Events generiert werden, die an alle angeschlossenen Clients verteilt werden. Die Event-Klassen werden von dem selben Werkzeug erzeugt, welches auch den Container etc. generiert. Es wird also einerseits ein verteilter Event-Mechanismus und andererseits eine Möglichkeit eingeführt, Attribute im Stub cachable zu gestalten, wobei deren Inhalt automatisch aktualisiert wird, wenn andere Clients den Inhalt ändern. Dieses geht über den Ansatz des Corba Event Service hinaus, indem Proxyfunktionalität mit der Möglichkeit zur Aktualisierung ergänzt wird.

Auf der Serverseite werden die Events vom Container verteilt. Dieser erzeugt für jeden angeschlossenen Client einen Thread, um die Events zu verschicken. So wird verhindert, daß erst auf einen Timeout gewartet werden muß, bis ein Event zum nächsten Client geschickt werden kann. Um zu verhindern, daß zu viele Threads benötigt werden, kann ein Threadpool eingerichtet werden, was allerdings bisher im JaTeK-System nicht nötig war. Neben dem Aktivieren und Deaktivieren von Enterprise Beans mittels des Mechanismus' der EJB können auch die Container mit Hilfe von RMI-Activation [1] deaktiviert werden, um zusätzlich die Serverbelastung zu mindern. Diese Maßnahme hat jedoch Performance-Auswirkungen, die zu beachten sind.

Für das am Anfang dargestellte Beispiel bedeutet dies, daß das TreeModel auf ein Remote Interface zugreift, welches "cachable" Attribute des Stubs enthält. Wird in einem anderen Client der JTree aktualisiert, so wird ein entsprechendes verteilte Event erzeugt und allen angemeldeten Clients zur Verfügung gestellt. Danach wird die fireTreeStructureChanged()-Methode aufgerufen und der JTree aus dem TreeModel neu erzeugt, in welchem nun bereits alle Daten auf dem Client vorliegen. Der Vorgang wird in der Fig. 3 näher verdeutlicht.

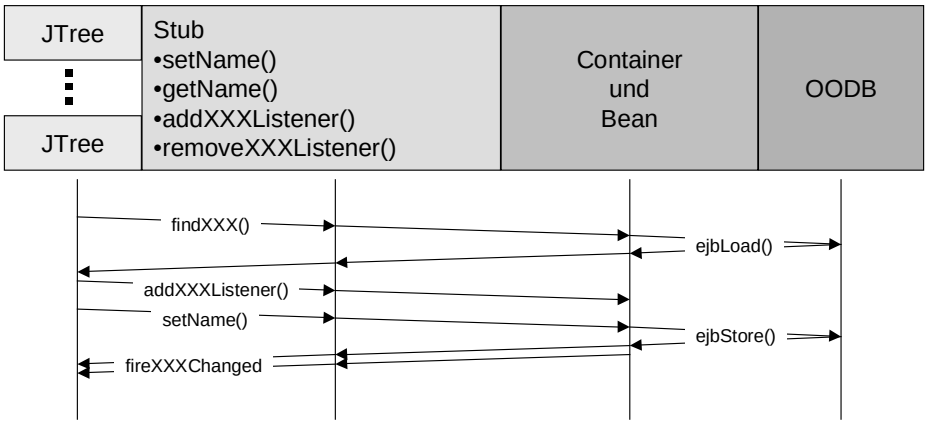

Fig. 3 - Szenario des "cachable" Stub

Eine Performanceerhöhung kann nur erwartet werden, wenn wenige Änderungen bei einer großen Datenmenge auftreten. Das Verfahren wird kontraproduktiv, wenn viele Änderungen auf einer kleinen Datenmenge erfolgen.

Das Verfahren kann bei einem weitverzweigten JTree zu einem erhöhten Verwaltungsaufwand führen, da im einfachsten Falle jeder Knoten ein Listener wäre, der angemeldet werden müßte. In diesem Falle wird der gesamte JTree als Listener angemeldet und der TreePath zum geänderten Objekt im JTree verwendet. Ähnliche Überlegungen sind bei anderen komplexen Strukturen anzustellen, wobei der beim JTree verwendete TreePath abgewandelt zur Lokalisierung der Änderung geeignet ist.

3.2 Behandlung von Ausnahmefällen

Bei dem eben vorgestellten Szenario können eine Reihe von Ausnahmefällen auftreten. In diesem Abschnitt sollen die Ausnahmefälle und deren mögliche Lösungen skizziert werden.

- Gleichzeitiges Erstellen von Objekten im Baum: Es werden beide Operationen serialisiert und in der zeitlichen Reihenfolge abgearbeitet, so daß ein erstellter Knoten eher erscheint, als der andere. (siehe Fig. 4)
- Löschen eines Objekts bei gleichzeitigem Schreiben des Objekts: Die Accessor-Methoden enthalten Exception, die geworfen werden, wenn auf ein verändertes Objekt zugegriffen wird. Es wird optimistisches Sperren (Locking) durchgeführt, d.h. die Änderung wird erst nach der Operation bei allen gleichzeitig durchgeführt.

Geschrieben wird immer sofort auf dem Server (write through), gelesen hingegen aus den im Cache zwischengespeicherten Variablen. (siehe Fig. 4)

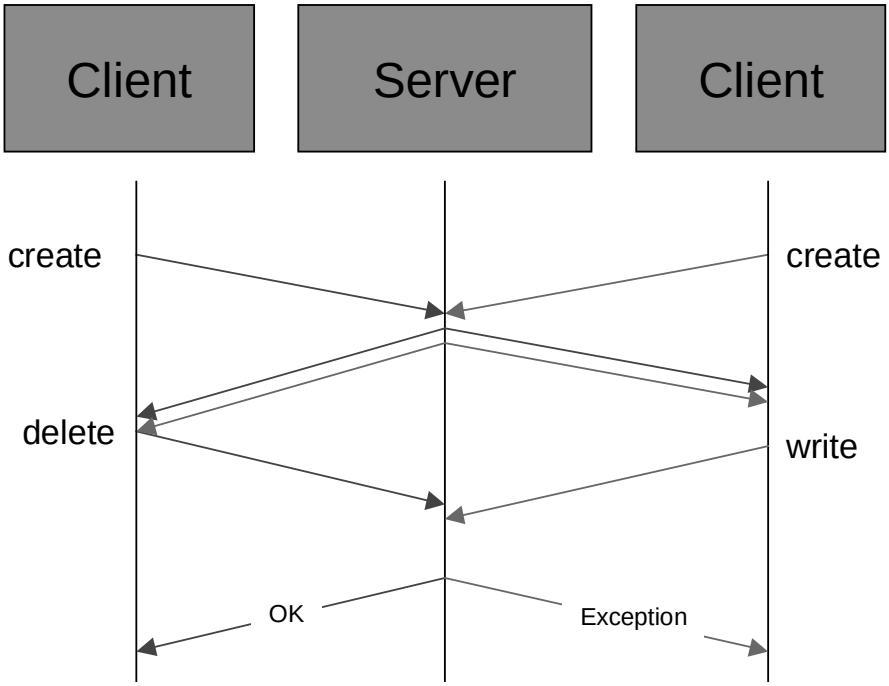

Fig. 4 - Ausnahmefälle

3.3 Datenbankeinsatz

Um die Persistenz der Entity Beans zu gewährleisten, können die Daten entweder in ein File serealisiert werden bzw. mit Hilfe eines RDBMS oder eines ODBMS gespeichert werden. Hier sollen die auftretenden Probleme der Speicherung mittels eines RDBMS und eines ODBMS bei der Verwendung von Container Managed Persistence (CMP) diskutiert und Lösungsvorschläge vorgestellt werden. Die Serialisierung in einer Datei wird nicht weiter betrachtet.

Bei CMP steht die Forderung, daß der Container sowohl die Speicherung als auch das Laden vornimmt. Das impliziert, daß der Container auch das Home-Interface mit seinen create()- und finderXXX()-Methoden implementieren muß.

Das Problem kann dabei entweder von der Datenbank oder von dem Enterprise Bean aus angegangen werden. Nachfolgende Tabelle erläutert dieses näher.

	RDBMS	OODBMS
Datenbank zu Bean	Ausgehend von einer vorhandenen Tabellenstruktur und einem Enterprise Bean wird mit Hilfe des Container Generators ein Mapping von Tabellenspalten zu Attributen im Enterprise Bean vorgenommen.	Ausgehend von einer vorhandenen Objektstruktur wird mit Hilfe des Container-Generators ein Mapping von Attributen vorgenommen.
Bean zu Datenbank	Ausgehend von den Attributen innerhalb der Enterprise Beans wird eine entsprechende Tabelle in der Datenbank definiert. Der Primary Key wird entweder zusätzlich generiert oder ausgewählt.	Ausgehend von den Attributen des EnterPrise Beans wird eine persistente Klasse erzeugt.
create()-Methoden	Es wird ein insert()-Statement auf einer Tabelle ausgeführt.	Es wird ein persistentes Objekt erzeugt und in eine Collection eingeordnet, um Erreichbarkeit zu gewährleisten.
finderXXX()-Methoden	Anhand der Auswahl von Attributen werden innerhalb einer Klasse finderXXX()-Methoden definiert.	Anhand der Attribute in der persistenten Klasse werden die finderXXX()-Methoden definiert und entsprechende Collections in der Datenbank angelegt.
Primary Key	Es wird automatisch eine Klasse erzeugt, die den Primary Key der entsprechenden Tabelle repräsentiert. Dieser ist entweder „int“, String oder eine Kombination aus mehreren Spalten	Es wird die OID verwendet. Die Primary-Key-Klasse wird von einem Werkzeug automatisch generiert und ist für alle Klassen gleich.

Table 1. Gegenüberstellung der Datenbankparadigmen bei der Realisierung von Persistenz der Entity Beans.

Darüber hinaus treten folgende Probleme auf:

- Erzeugen eines neuen Objekts: Objekte, die in einer objektorientierten Datenbank erzeugt werden, müssen zu einem Objekt zugeordnet werden, damit sie nicht bei dem Lauf eines Persistent Garbage Collectors zerstört werden. Um hier ein ähnliches Verhalten wie bei der Verwendung von relationalen Datenbanken zu erreichen, sollte jedes neu erzeugte Objekt in eine Collection eingeordnet werden. Damit ist allerdings kein Persistent Garbage Collection mehr möglich und die Daten in der Datenbank können nur noch explizit gelöscht werden.

- finderXXX()-Methoden: Objekte können über das Home-Interface und die finderXXX()-Methoden gefunden werden. In objektorientierten Datenbanken gibt es Wurzelknoten (Entrypoints), die in der Form in der Klassenhierarchie der Anwendung keine Entsprechung finden. Beim automatischen Generieren des Containers muß also spezifiziert werden, welche Objekte einer Klasse als Wurzelknoten verwendet werden. Wurzelknoten werden gewöhnlich immer durch die Angabe eines Strings identifiziert, so daß also für diese Klasse gleichzeitig eine Suche durch die Angabe eines Strings möglich sein muß. Bei CMP sollte das Interface um finderXXX()-Methoden, die der Container Generator erzeugt, erweitert werden. Es ist nicht sinnvoll, den Entwickler finderXXX()-Methoden spezifizieren zu lassen, die eine Datenbank nicht umsetzen kann.
- Primary Key: In CMP ist es generell sinnvoll, die Primary Key-Klasse von dem Generator des Containers generieren zu lassen. Die Implementierung einer Primary Key-Klasse selbst vorzunehmen, erscheint eigentlich nur bei Bean Managed Persistence (BMP) zweckmäßig.

4 Ausblick

Dieser Artikel zeigt Möglichkeiten zur Integration der Corba Event Service-Funktionalitäten in EJB, wie auch der Artikel [5] im Java Developer's Journal. Er erläutert die Möglichkeit, EJB so zu erweitern, daß Proxies einfach erzeugt werden können. Er zeigt eine Alternative beim Pull-Mechanismus auf, diskutiert die Schwierigkeiten, die beim Einsatz einer OODB mit Container Managed Persistence auftreten und gibt Schritte zu deren Lösung an.

Die vorgeschlagenen Lösungen erweitern das EJB-Konzept, so daß verteilte Anwendungen einfacher erstellt werden können. Außerdem können die Netzwerklast gesenkt und verteile Events ermöglicht werden. Auch wenn objektorientierte Datenbanken bis jetzt nur vereinzelt eingesetzt werden, sollte der EJB Standard auch diese Datenbanken geeignet unterstützen, da kein Impedance Mismatch mehr auftritt, und so einerseits der Quelltext reduziert und anderseits die Fehleranfälligkeit gesenkt werden kann.

Der hier vorgestellte Algorithmus kann in der Form ausgebaut werden, daß er zusätzlich eine einstellbare Tiefe von Baumstrukturen beim Zwischenspeichern im Cache berücksichtigt. Des weiteren ist die vorgestellte Lösung für andere verteilte Applikationen, wie z.B. für den in JaTeK integrierten, verteilten Texteditor auf der Basis des JTextPane geeignet, der ebenfalls baumartige Objektstrukturen enthält.

Literatur

[1] (RMI Specification)
http://www.javasoft.com/products/jdk/1.2/docs/ guide/rmi/spec/rmiTOC.doc.html

[2] (Overview of RMI in the WebLogic Framework)
http://www.weblogic.com/docs/techoverview/rmi.html

[3] Kirchner L., Meissner K., Wehner F.; Kooperative multimediale Anwendungen: Basis für virtuelle Arbeitsumgebungen – GeNeMe98, Reihe: Telekommunikation und Mediendienste – Band 2, 1998, S. 109

[4] (Java Shared Data Toolkit) http://www.sun.com/software/jsdt/index.html

[5] (Event Managegement & EJB)
http://www.sys-con.com/java/feature/4-1/eventManagementEnterpriseJavaBeans/

[6] (Corba Event Spezification)
ftp://www.omg.org/pub/docs/formal/97-12-11.pdf

[7] (EJB – Specifikation)
ftp://ftp.javasoft.com/docs/ejb/ejb.10.pdf

[8] (MVC in Swing)
http://www.javasoft.com/products/jfc/tsc/archive/what_is_arch/swing-arch/swing-arch.html