Druckdatum: 01.10.98 23:57

Komponentenstandard III:

JavaBeans von SUN

Von

Michael Hurler

Jens Rohde

Thomas Schmitz

Technische Universität Darmstadt

Fachbereich Rechts- und Wirtschaftswissenschaften

Fachgebiet Entwicklung von Anwendungssystemen

Prof. Dr. Erich Ortner

Wirtschaftsinformatik-Seminar SS 1998:

Komponentenorientierte Anwendungssystementwicklung

Thema 5:

Komponentenstandard III: JavaBeans von SUN

Betreuer:

Dipl.-Inf.wiss. Jörg Kalkmann

eingereicht von:

Michael Hurler

Jens Rohde

Thomas Schmitz

Zusammenfassung:

Diese Arbeit stellt eine Einführung in Java und die Java-eigene Komponentenarchitektur JavaBeans dar. Dabei werden sowohl die Grundkonzepte von Java und JavaBeans, als auch die weiterführenden, auf JavaBeans basierenden Konzepte JavaBeans Activation Framework, InfoBus und Enterprise JavaBeans vorgestellt. Im weiteren wird auf die Positionierung Javas in den heute existierenden heterogenen Systemumgebungen eingegangen und ein integriertes Bild bezüglich der Einbindung des JavaBeans-Konzeptes in die existierenden Architekturen zur Anwendungsentwicklung in heterogenen, verteilten Umgebungen, CORBA und DCOM, aufgebaut.

Schlüsselwörter:

Komponentenarchitektur, Java, JavaBeans, Enterprise JavaBeans, Enterprise Beans, CORBA, DCOM, InfoBus, JavaBeans Activation Framework

Summary:

This work is an introduction to Java and Java's component architecture JavaBeans. We present the basic concepts of Java and JavaBeans as well as the advanced, JavaBeans based concepts JavaBeans Activation Framework, InfoBus and Enterprise JavaBeans. In the following we discuss the position of Java in today's heterogeneous computer environments and compose an overall picture on the integration of JavaBeans into the existing architectures for application development in heterogeneous, distributed environments CORBA and DCOM.

Key words:

Component architecture, Java, JavaBeans, Enterprise JavaBeans, Enterprise Beans, CORBA, DCOM, InfoBus, JavaBeans Activation Framework

Inhaltsverzeichnis

Inhaltsverzeichnis I
Abkürzungsverzeichnis III
Abbildungsverzeichnis V
1 Einleitung 1
2 Einführung in die Konzepte von Java 2
2.1 Allgemeines 2
2.2 Interfaces 3
2.3 Packages 4
2.4 Plattformunabhängigkeit 5
2.5 Applets 5
2.6 Internationalization 8
3 Die JavaBeans-Architektur 9
3.1 Designaspekte 9
3.1.1 Komponentengranularität 9
3.1.2 Portabilität 10
3.1.3 Einheitliche, qualitativ hochwertige API 11
3.1.4 Einfachheit 11
3.2 Merkmale von JavaBeans 11
3.3 Public Interface 12
3.3.1 Events 12
3.3.2 Properties 14
3.3.3 Methoden 15
3.4 Introspection 15
3.4.1 BeanInfo-Klasse 16
3.4.2 Die Klasse Introspector 16
3.5 Customization 17
3.5.1 Property Sheet 17
3.5.2 Customizer 18
3.6 Persistenz 19
3.6.1 Serialization 19
3.6.2 Externalization 20

4 Suns Erweiterungen des JavaBean Frameworks ... 21
4.1 JavaBeans Activation Framework ... 21
4.1.1 Architektur ... 22
4.1.2 Anwendungsbereich ... 24
4.1.3 Resümee ... 25
4.2 InfoBus ... 25
4.2.1 Das InfoBus-Protokoll für den Datenaustausch ... 26
4.2.2 Die wichtigsten InfoBus-Schnittstellen ... 29
4.2.3 Die InfoBus-Events ... 31
4.2.4 Resümee ... 31
4.3 Das Enterprise JavaBeans-Komponentenkonzept ... 32
4.3.1 Grundkonzept und Ziele der Enterprise JavaBeans ... 32
4.3.2 Grobarchitektur ... 33
4.3.3 Der EJB-Container aus Sicht des Providers ... 34
4.3.4 Die Enterprise Beans aus Sicht des Providers ... 40
4.3.5 Resümee ... 44
5 Java und JavaBeans in einer heterogenen Umwelt ... 45
6 JavaBeans und Enterprise Beans als Komponenten ... 48
7 Resümee ... 49
7.1 Die Zukunft von Java ... 49
7.2 Die Zukunft von JavaBeans ... 52
7.3 Enterprise JavaBeans ... 53
8 Ausblick ... 54
9 Code-Beispiel ... 54
Literaturverzeichnis ... 57
Stichwortverzeichnis ... 58

Abkürzungsverzeichnis

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AWT	Abstract Window Toolkit
BS	Betriebssystem
bzw.	beziehungsweise
CGI	Common Gateway Interface
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
d.h.	das heißt
DBMS	Datenbank Management System
DCOM	Distributed Component Object Model
EB	Enterprise Bean
EJB	Enterprise JavaBeans
etc.	et cetera
GIOP	General InterORB Protocol
GUI	Graphical User Interface
HTML	HyperText Markup Language
i.d.R.	in der Regel
I/O	Input / Output
IDL	Interface Definition Language
IIOP	Internet InterORB Protocol
JAF	JavaBeans Activation Framework
JDBC	Java Database Connectivity
JDK	Java Development Kit
JFC	Java Foundation Classes (aka Swing)
JNDI	Java Naming and Directory Interface
JVM	Java Virtual Machine
Kap.	Kapitel
MIME	Multipurpose Internet Mail Extension
MS	Microsoft
OLE	Object Linking and Embedding

OMG	Object Management Group
ORB	Object Request Broker
RFC	Request for Comment
RMI	Remote Method Invocation
TMS	Transaktions Management System
u.v.m.	und vieles mehr
UID	Unique Identifier
URL	Uniform Resource Locator
vgl.	vergleiche
VM	(Java) Virtual Machine
WWW	World Wide Web
z.B.	zum Beispiel

Abbildungsverzeichnis

Abbildung 1 Java Virtual Machine als Brücke zwischen Anwendung und System 5
Abbildung 2 Applets und Sicherheit 8
Abbildung 3 Event Konzept...12
Abbildung 4 Property Sheet...17
Abbildung 5 Persistenz einer Bean 19
Abbildung 6 Serialization process...20
Abbildung 7 Externalization process 21
Abbildung 8: Architektur des JavaBeans Activation Frameworks 22
Abbildung 9 Die Verwendung des InfoBuses...26
Abbildung 10 Die Mitglieder des InfoBuses...27
Abbildung 11 EJB-Grobarchitektur 34
Abbildung 12 EJB Runtime Execution Model..35
Abbildung 13 Java in einer heterogenen Welt 45
Abbildung 14 Property Sheet der Multiplicator Bean...56
Abbildung 15 Kombination mit einer Button Bean 56

1 Einleitung

Umfangreiche Applikationen und sich immer weiter verkürzende Lebenszyklen führen zu einem explosionsartigen Anstieg der Entwicklungskosten in der Softwarebranche. Eine Möglichkeit dieser Misere zu entkommen besteht in der Wiederverwendung bereits entwickelter Teile, den Komponenten der Software. Hierdurch läßt sich der Entwicklungsaufwand stark reduzieren, da nicht jede Softwarelösung komplett neu implementiert werden muß. Eine solches Zusammenfügen von Komponenten zu einem größeren Ganzen, wie es z.B. in der Automobilindustrie und anderen Ingenieursdisziplinen längst gang und gäbe ist, steckt bei der Anwendungsentwicklung aber noch in den Kinderschuhen.

Bevor die Wiederverwendung von Komponenten möglich ist, muß erst eine Infrastruktur geschaffen werden. Die bisherigen Beiträge dieser Vortragsreihe beschreiben die notwendigen Bedingungen, die eine Wiederverwendung von Komponenten erst ermöglichen und zwei gängige Ausführungsumgebungen für Komponenten, CORBA und DCOM. Dieser Beitrag beschreibt den von Sun Microsystems entwickelten Komponentenstandard für Java, JavaBeans, und Erweiterungen dieser Architektur.

Das folgende Kapitel 2 gibt eine kurze Einführung in die Sprache Java und ihre Konzepte. Kapitel 3 stellt den Java Komponentenstandard JavaBeans mit seinen Ideen, Konzepten und Konventionen dar. Im 4. Kapitel werden die von Sun Microsystems entwickelten Erweiterungen der Komponententechnologie vorgestellt. Hierzu zählen das JavaBeans Activation Framework, der InfoBus und die Enterprise JavaBeans. Kapitel 5 zeigt die Integration von JavaBeans in die bestehende Informationstechnologie-Infrastruktur. An dieser Stelle wird insbesondere auf die bereits vorgestellten Technologien CORBA und DCOM eingegangen. Ein Vergleich des Konzepts JavaBeans mit den im zweiten Vortrag vorgestellten Definitionen zum Komponentenbegriff findet in Kapitel 6 statt. Das 7. Kapitel faßt die gewonnenen Eindrücke nochmals zusammen und Kapitel 8 versucht mögliche Entwicklungen aufzuzeigen. Abschließend wird in Kapitel 9 ein kurzes Beispiel für eine JavaBean gegeben und ihre Verwendung in einem Entwicklungswerkzeug vorgestellt.

Dieser Beitrag soll lediglich die Sprache Java und die Komponentenkonzepte JavaBeans, JavaBeans Activation Framework, InfoBus und Enterprise JavaBeans vorstellen. Eine ausführliche Beschreibung der umfassenden Spezifikationen und der Programmierung kann

und soll an dieser Stelle nicht vermittelt werden. Zu diesem Zweck sei auf die zur Verfügung stehenden Literaturquellen verwiesen.

2 Einführung in die Konzepte von Java

Die Technologie JavaBeans ist eng im Zusammenhang mit der Sprache Java zu betrachten, da JavaBeans letztendlich lediglich eine Erweiterung der Konzepte von Java darstellen. Zunächst soll deshalb an dieser Stelle eine Einführung in die wesentlichen Konzepte, Strukturen und Möglichkeiten der Programmiersprache Java gegeben werden, um es auch Lesern, die nicht oder nur wenig mit der Sprache vertraut sind, zu ermöglichen, sich über die Vor- und Nachteile von Java ein Bild zu machen und den nachfolgenden Ausführungen folgen zu können.

2.1 Allgemeines

Die Sprache Java wurde von Sun Microsystems im Jahre 1995 eingeführt. Seitdem wurde sie um einige Konzepte, wie zum Beispiel JavaBeans, erweitert. Im Frühjahr 1997 wurde mit der Freigabe des Java Development Kits (JDK) 1.1 der Sprachstandard nochmals modifiziert. Für den Sommer 1998 wird das nächste große Update auf den Standard mit der Version 1.2 erwartet, der dann weitere Erleichterungen für die Entwicklung von fensterbasierten Anwendungen und deutliche Performancegewinne mit sich bringen soll.

Java ist von Anfang an als eine objektorientierte Sprache entwickelt worden. In Java sind deshalb fast alle Dinge Objekte. Nur die einfachen Zahlen, Zeichen sowie boolsche Werte bilden eine Ausnahme. Trotzdem findet man nicht alles in Java, an das man sich bei anderen objektorientierten Programmiersprachen bereits gewöhnt hat. So fehlen zum Beispiel die Mehrfachvererbung, Templates (oder generische Klassen) und das Operator Overloading. Java bedient sich statt dessen anderer Konzepte oder bietet diese Möglichkeiten, wie beim Operator Overloading erst gar nicht an. Auf der anderen Seite erspart dies dem Programmierer zum Beispiel die Beseitigung der bei der Mehrfachvererbung immer wieder auftretenden Namenskonflikte, und im Fall des Operator Overloadings ist es unbestreitbar, daß ein Programmtext ohne überladene Funktionen besser verständlich ist. Zuletzt mag auch das Argument von einfacher zu entwickelnden Compilern und schnellerer Laufzeitausführung eine Rolle gespielt haben.

Auf der anderen Seite findet man in Java aber auch Konzepte, die andere objektorientierte Programmiersprachen vermissen lassen. So verfügt Java über eine Garbage Collection, die nicht mehr referenzierte Objekte aus dem Speicher entfernt. Daher bleibt dem Programmierer die Freigabe nicht mehr genutzten Speichers unter Java erspart.

2.2 Interfaces

Insbesondere die fehlende Möglichkeit der Mehrfachvererbung fordert aber eine Lösung, die diesen Mißstand zumindest zum Teil kompensiert. Java verwendet deshalb das Konzept der Interfaces. Ein Interface ist ein Java-Konstrukt, das lediglich Signaturen von Methoden, nicht aber deren Rümpfe enthält. Unter einer Signatur verstehen wir die Definition des Rückgabetyps, des Methodennamens sowie der Parametertypen und -namen (z.B. `Typ1 myMethod(Typ2 param1, Typ3 param2)`), unter dem Methodenrumpf den ausführbaren Code der Methode. Interfaces enthalten also keinen ausführbaren Code, sie können allerdings Konstanten definieren. Die Verwendung eines Interfaces bedeutet nun, daß man in der Klasse in der man das Interface verwenden, oder besser gesagt, implementieren möchte, dies durch das Anhängen von `implements` *`Interfacename`* hinter den Klassenkopf angibt. In dieser Klasse müssen dann die im Interface deklarierten Methoden ausprogrammiert werden. Natürlich ist es möglich, in einer Klasse mehrere Interfaces zu implementieren. Im Gegensatz zur Mehrfachvererbung ist es hier aber immer nötig, den Code für jede Klasse, die das Interface benutzt, neu zu schreiben. Interfaces eignen sich also hervorragend dazu, Klassen eine bestimmte Schnittstelle aufzuzwingen, wenn diese Klassen zu einem bestimmten Typ von Klassen gehören sollen. So müssen zum Beispiel alle Applets das Interface `Applet` implementieren, dazu unten mehr.

Ferner gibt es bei Java sogenannte abstrakte Klassen, die eine Mischform zwischen gewöhnlichen Klassen und Interfaces sind. Abstrakte Klassen werden durch das Vorstellen des Schlüsselworts `abstract` vor den Klassenkopf kenntlich gemacht und enthalten sowohl ausprogrammierte Methoden als auch Methoden-Signaturen, denen ebenfalls `abstract` vorangestellt ist. Abstrakte Klassen können nicht instanziiert werden, das heißt, es können keine Objekte vom Typ der Klasse erzeugt werden.

2.3 Packages

Im Java-Standard ist eine relativ große Anzahl von Standard-APIs enthalten. Für Java sind die Klassen und Interfaces dieser APIs in sogenannte Packages eingeteilt. Dies gibt der schon ziemlich großen Anzahl von Klassen und Interfaces, die zum Java-Standard gehören, eine übersichtliche Struktur. Das dafür angewendete Schema ermöglicht eine internetweite Namensgebung. Die Java Standard-Packages werden nach dieser Konvention mit `java.`*`packagename`* bezeichnet. Klassen und Methoden können mit voll-qualifizierten Namen bezeichnet werden. So kann z.B. die Methode `start()` der Klasse `Applet` aus dem Package `java.applet` mit `java.applet.Applet.start()` eindeutig angesprochen werden. Packages, die nicht zum Standardumfang von Java gehören sollen wie folgt benannt sein:

`internet-domain-postfix.internet-firmen-name.firmenintern-eindeutiger-package-name.class-name`,

so z.B. `com.sybase.jdbc.SybDriver` [Flanagan, 1997].

Die wichtigsten, zu Java gehörenden APIs sind:

- das Abstract Window Toolkit (AWT)-API und seine Erweiterung Java Foundation Classes (JFC), die die Erstellung GUI-basierter Software unterstützt
- das Java Database Connectivity (JDBC)-API zur standardisierten Anbindung von Java-Anwendungen an ein relationales Datenbanksystem
- das I/O-API, das Schnittstellen zum Dateisystem, zum Drucker usw. bietet
- das Net-API, das einfache Schnittstellen zur Benutzung von Netzen bietet
- das Remote Method Invocation (RMI)-API, zur Entwicklung von verteilten Anwendungen, die auf verschiedenen Virtual Machines (dazu unten mehr) und damit auf verschiedenen Rechnern laufen können
- das Reflection-API, das es ermöglicht, Felder, Methoden und Konstruktoren von Objekten zur Laufzeit abzufragen. Die Werte der Felder können dann geändert, die Methoden aufgerufen und mit den Konstruktoren neue Objekte erzeugt werden.
- das Bean-API, das die Konzepte der komponentenorientierten Anwendungsentwicklung in Java einführt und Gegenstand dieser Arbeit ist.

2.4 Plattformunabhängigkeit

Der Grund, warum Java bereits so kurze Zeit nach seiner Einführung eine dermaßen große Popularität genießt, ist aber ein anderer, nämlich die Plattformunabhängigkeit. Plattformunabhängigkeit bedeutet, daß sich ein einmal geschriebenes Java-Programm auf jedem Rechner unter jedem Betriebssystem ausführen läßt, auf dem eine Java-Implementierung, die das Kernstück, die Java Virtual Machine (JVM) beinhaltet, zur Verfügung steht. Was heißt das? Zunächst übersetzt der Programmierer seine Klassen mit einem Compiler, der den sogenannten Bytecode erzeugt. Im Gegensatz zu anderen Programmiersprachen besteht dieser Bytecode aber nicht aus Maschinenbefehlen für eine bestimmte Hardware, und er enthält auch keine Aufrufe für ein bestimmtes Betriebssystem, sondern muß von einem bestimmten Programm, nämlich der JVM interpretiert werden. Diese JVM ist wiederum plattformabhängig, so daß sie die besonderen Eigenschaften des Systems, für das sie entwickelt worden ist, ausnutzen kann. Daher muß der Entwickler nur eine Version seines Programms veröffentlichen, ohne den Quellcode herauszugeben.

Diese Vorgehensweise hat aber auch ihre Nachteile. Insbesondere ist der Programmablauf bei interpretiertem Code gewöhnlich langsamer als bei maschinenspezifischem Code.

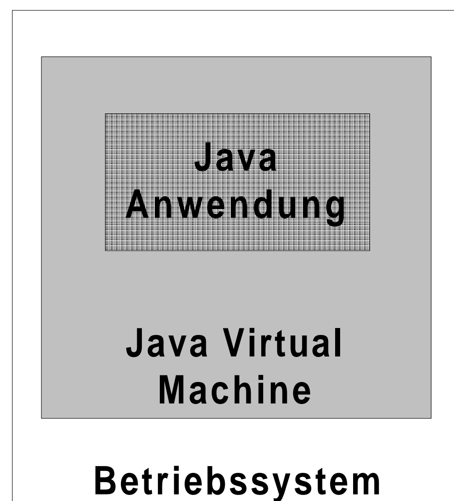

Abbildung 1 Java Virtual Machine als Brücke zwischen Anwendung und System

2.5 Applets

In einer Welt der Informationsverarbeitung, in der das Internet eine so große Rolle spielt wie heute, ist Plattformunabhängigkeit ein wichtiger Faktor. Wo sich HTML zum einfachen

Formatieren von Text und Grafik eignet, findet es dennoch seine Grenzen, wenn es darum geht, Inhalte von Webseiten interaktiv zu gestalten. Java bietet hier eine Möglichkeit, die interaktiven Elemente einer Seite auf den Client-Rechner zu laden und dort auszuführen. Dadurch wird die Belastung des Servers auf die Bereitstellung dieser interaktiven Elemente beschränkt. Andere Konzepte, wie z.B. CGI, führen Code auf der Server-Maschine aus, so daß dort leicht Kapazitätsengpässe entstehen können. Die in Java entwickelten interaktiven Elemente werden Applets genannt. Dabei handelt es sich um gewöhnliche Java-Applikationen, die aber eine definierte Schnittstelle zum Webbrowser bieten. Java-Applets implementieren deshalb das Interface `java.applet.Applet`. So stellen Klassen, die dieses Interface implementieren, unter anderem die Methoden `start()` und `stop()` zur Verfügung, über die der Webbrowser das Applet starten und anhalten kann. Applets sind also ganz gewöhnliche Java-Anwendungen, die lediglich eine zusätzliche Schnittstelle zum Browser haben. Ausgeführt werden sie von der JVM, die in den Browser integriert sein muß. Diese Applets können theoretisch von beliebig großer Komplexität sein, wobei dies in der Praxis durch die Zeit, die zum Laden der Applets über das Netz benötigt wird, seine Grenzen findet. Da Applets auf der Client-Maschine ablaufen, wurde in Java ein umfangreiches Sicherheitskonzept integriert, um zu verhindern, daß für den Benutzer unerwünschte Operationen auf seinem Rechner ausgeführt werden. Java-Applets können mit einer Signatur versehen sein, wobei der Benutzer festlegen muß, ob er dem Aussteller der Signatur traut oder nicht. Gewöhnlich sind Applets aber nicht signiert und der Benutzer traut keinem Aussteller. Dann können Applets unter anderem folgende Operationen nicht ausführen [Flanagan, 1997]:

- Dateien lesen
- Verzeichnisse abfragen
- die Existenz von Dateien überprüfen
- das Änderungsdatum oder die Größe von Dateien abfragen
- die Lese- und Schreibrechte abfragen
- überprüfen, ob ein Dateiname für eine Datei oder ein Verzeichnis steht
- Dateien schreiben
- Dateien löschen
- Verzeichnisse anlegen

- Dateien umbenennen
- Netzwerkverbindungen zu einem anderen Rechner als zu dem, von dem es geladen wurde, herstellen
- auf Netzwerkverbindungen an privilegierten Ports mit einer Port-Nummer kleiner als 1025 horchen
- Netzwerkverbindungen an Ports mit einer Port-Nummer kleiner als 1025 akzeptieren, wenn sie nicht von dem Rechner, von dem das Applet geladen wurde, ausgehen.
- Multicast-Sockets verwenden
- den Java-Interpreter beenden
- neue Prozesse starten
- dynamisch Native Code (plattformabhängige) Libraries laden
- Druckaufträge initiieren
- auf die Zwischenablage zugreifen
- mit den Methoden der Reflection API auf andere Objekte zugreifen, es sei denn, daß ihre Klassen von dem gleichen Host wie das Applet geladen wurden
- u.v.m.

Es kann also festgestellt werden, daß das Sicherheitskonzept für Java unerwünschte Effekte auf das System des Anwenders verhindert, da Applets diese Operationen gewöhnlich nicht benötigen.

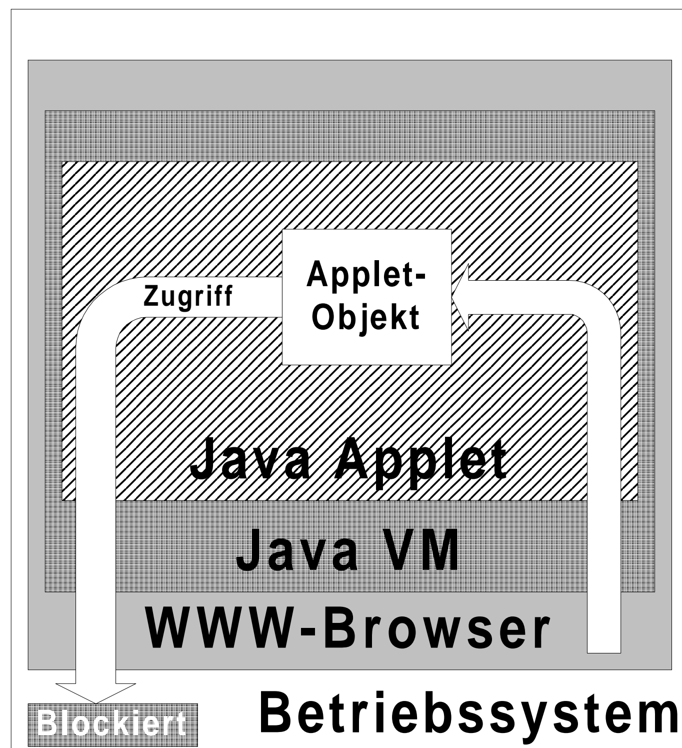

Abbildung 2 Applets und Sicherheit

In der Vergangenheit aufgetretene Probleme lagen immer in Implementierungsfehlern der JVM begründet, die dieses Sicherheitskonzept umsetzen muß.

2.6 Internationalization

Ein anderes Problem im weltweiten Datenverkehr stellen oft lokal unterschiedliche Sprachen und die unterschiedlichen Formatierungen von Datum und Zahlen dar. Java unterstützt daher das Konzept der Internationalization. Aus diesem Grund benutzt Java die Unicode Zeichencodierung. Seit der Version 1.1 wird Unicode 2.0 verwendet, vorher Unicode 1.0. Unicode wurde vom Unicode Consortium (http://unicode.org/) eingeführt und codiert jedes Zeichen mit 16 Bit; zur Zeit stehen 38.885 verschiedene Zeichen aus den wichtigsten Schriften zur Verfügung. Es ist allerdings zu beachten, daß viele Plattformen Unicode nur unzureichend oder gar nicht unterstützen. Auch sind kaum Fonts verfügbar, die alle Zeichen darstellen können. Die Unicode-Zeichen 32 bis 126 sind aber mit denen des ASCII-Codes identisch, so daß eine gewisse Kompatibilität gewährleistet ist.

Zur Formatierung, z.B. von Datum und Zahlen, bieten die Klassen des Packages `java.text` zahlreiche Methoden an, die es ermöglichen, zur Laufzeit beim jeweiligen Betriebssystem die lokalen Konventionen abzufragen und dann ausgegebene Daten entsprechend zu formatieren.

Weiterhin ist es möglich Anzeigen für den Benutzer in der jeweils lokalen Sprache anzugeben. Wenn dies gewünscht ist, wir anzuzeigender Text nicht wie üblich im Quelltext „hart verdrahtet“, sondern der Programmierer kann für die Anzeige Default-Werte vorgeben, sowie Werte in anderen Sprachen, die dann die Default-Werte überschreiben, wenn eine andere Sprache die lokale ist.

3 Die JavaBeans-Architektur

Zunächst soll in diesem Abschnitt auf allgemeine Designaspekte des Konzepts JavaBeans eingegangen und dann die Kernkonzepte vorgestellt werden.

Wir beginnen mit der Definition einer JavaBean, wie sie in der Spezifikation von JavaBeans, Version 1.01 von Sun Microsystems vorgegeben ist [Hamilton, 1997]:

> *„A JavaBean is a reusable software component that can be manipulated visually in a builder tool.“*

Dies ist eine sehr allgemeine Definition und im folgenden wird aufgezeigt, wie Sun Microsystems diese Idee in der Spezifikation der JavaBeans umgesetzt hat.

3.1 Designaspekte

Das Ziel von Sun Microsystems war es, mit dem JavaBean-API eine Softwarekomponentenarchitektur für die Programmiersprache Java zu manifestieren. Die entwickelten Beans sollen sich dann einfach zu mehr oder weniger komplexen Anwendungen mit Hilfe eines Buildertools bzw. Entwicklungswerkzeuges zusammenstellen lassen. Dabei ist es von Vorteil, ein solches Buildertool zu verwenden, man kann auf seinen Einsatz jedoch auch verzichten. Die API muß hierzu einigen Aspekten genügen, die im folgenden erläutert werden.

3.1.1 Komponentengranularität

JavaBeans soll es ermöglichen, Komponenten unterschiedlicher Größe und Komplexität (Granularität) zu entwickeln. Dabei soll es genauso möglich sein einen kleinen Baustein, zum Beispiel einen Button oder ein Eingabefeld, als Bean zu entwickeln, wie auch eine große Komponente, die einer vollständigen Anwendung ähnelt, wie sie in Compound Documents

Verwendung findet. Hier kann eine Tabellenkalkulation, die in eine HTML-Seite eingebettet ist, als Beispiel betrachtet werden. JavaBeans ist jedoch primär für die Entwicklung kleiner und mittlerer Komponenten gedacht, ohne sich darauf zu beschränken.

Um Verwechslungen zu vermeiden, möchten wir an dieser Stelle noch einmal hervorheben, daß die Konzepte von JavaBeans und Applets orthogonal zueinander stehen. Während Beans Komponenten im Sinne der JavaBeans-Spezifikation darstellen, handelt es sich bei Applets um kleine Anwendungen, die von der JVM eines Webbrowsers ausgeführt werden und eine Schnittstelle zu diesem bieten. Aber natürlich können Java Applets, genau wie Stand Alone Applikationen aus JavaBeans zusammengesetzt werden.

Sun Microsystems beabsichtigt aber nicht, eine Integration von Beans in komplexe Dokumente, wie zum Beispiel jene von Microsoft Office voranzutreiben. Bei der Spezifikation ist eher an einfache Compound Documents, wie eben Webseiten gedacht worden. Dennoch gibt es APIs, die analog z.B. zu den OLE Control APIs oder den ActiveX-APIs sind, die eine Verwendung in diesen High-End Compound Documents ermöglichen.

3.1.2 Portabilität

Die größte Stärke der Programmiersprache Java ist ihre Plattformunabhängigkeit. Dieser Aspekt darf deshalb auch bei dem Entwurf einer Komponentenarchitektur nicht unberücksichtigt bleiben, soll auch diese Komponentenarchitektur plattformunabhängig sein. Eine Bean soll grundsätzlich auf allen Plattformen die gleiche Funktionalität bieten.

Dennoch findet die Plattformunabhängigkeit ihre Grenze, wo Beans in bestehende Komponentenmodelle der aktuellen Plattform integriert werden müssen. Das ist zum Beispiel dann der Fall, wenn eine Bean von einer Anwendung benutzt werden soll, die ausschließlich ein plattformspezifisches Komponentenmodell verwendet. So müßte eine JavaBean, die von Microsoft Word genutzt werden soll, in die ActiveX/COM-Architektur eingebettet werden. Dies gilt allerdings nur für die sogenannte Root-Bean, nämlich die Bean, die in der Aufrufhierarchie an erster Stelle steht. Beans, die von dieser Bean benutzt werden, müssen nicht integriert werden.

Die Integration geschieht mit Hilfe einer „Bridge“, so daß Aufrufe zwischen JavaBeans und ActiveX / COM möglich sind. Die Bridge ist für den Entwickler der JavaBeans nicht sichtbar. Sun Microsystems hat beim Design der JavaBeans-API allerdings die Notwendigkeit der Bridges berücksichtigen müssen. Insbesondere ist sichergestellt, daß zu den drei wichtigsten

Komponentenmodellen, nämlich OpenDoc, OLE/COM/ActiveX und LiveConnect Bridges entwickelt werden können.

3.1.3 Einheitliche, qualitativ hochwertige API

Auf verschieden Plattformen ist es unter Umständen nicht möglich, die volle Funktionalität der JavaBeans-API zu unterstützen. In diesen Fällen muß eine alternative Funktionalität geschaffen werden, die der ursprünglich beabsichtigten am ehesten entspricht. Wenn zum Beispiel auf einer Plattform das Verschmelzen von mehreren Menüleisten nicht möglich ist und eine JavaBean als Komponente in einem Compound Document eine Menüleiste unterstützt, dann kann das Menü als Alternative in einem eigenen Fenster über der Anwendung erscheinen, statt in ihre Menüleiste integriert zu werden.

Da sich die Entwickler von JavaBeans nicht mit den Möglichkeiten der jeweiligen Plattformen auseinandersetzen sollen, muß diese alternative Funktionalität von der plattformabhängigen Implementierung zur Verfügung gestellt werden.

3.1.4 Einfachheit

Wie schon oben beschrieben, ist die JavaBeans-API hauptsächlich für kleine und mittlere Komponenten konzipiert worden, ohne sich darauf zu beschränken. Dies macht es auch möglich, die API relativ einfach zu halten, so daß der Entwickler sich schnell in die API einarbeiten und somit auch relativ schnell zu Ergebnissen kommen kann.

3.2 Merkmale von JavaBeans

Die Merkmale einer JavaBean lassen sich in vier Punkten darstellen [Jubin et al., 1997, S. 10]:

- Public Interface
- Introspection
- Customization
- Persistenz

In den folgenden Abschnitten werden diese Charakteristiken ausführlicher erläutert.

3.3 Public Interface

Das Public Interface (öffentliche Schnittstelle) einer Bean definiert, wie eine Bean mit anderen Beans oder auch einer Applikation interagiert. Es läßt sich in die drei im folgenden beschriebenen Gruppen einteilen.

3.3.1 Events

Die Events bzw. Ereignisse stellen den Benachrichtigungsmechanismus der JavaBeans dar. Diese Benachrichtigung geschieht in Java über Methodenaufrufe. Die Namen der Methoden werden in einem Event Listener Interface definiert. Ein Event Listener, der dieses Interface implementiert, ist nach der Registrierung bei der Event Source bereit, Events zu empfangen und zu verarbeiten. Die Event Source feuert ein Ereignis durch Erstellen eines neuen Event Objects und dem Aufruf der Eventverarbeitungsmethoden aller registrierten Event Listener.

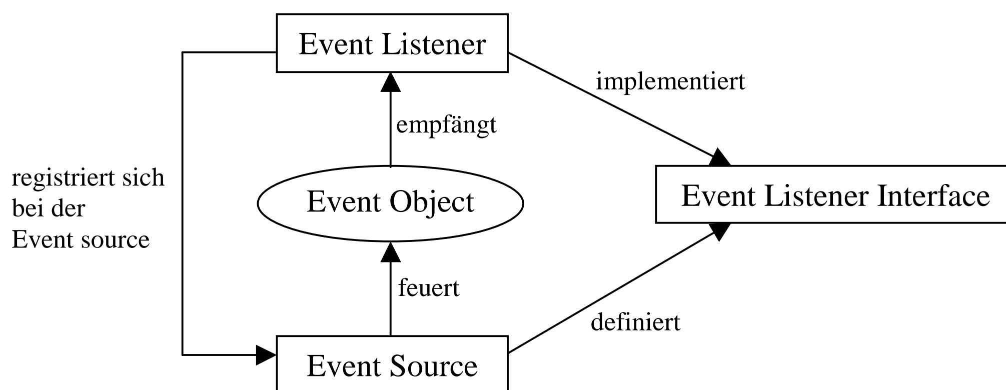

Abbildung 3 Event Konzept

Die folgenden bereits genannten vier Elemente sind also im JavaBeans-Eventkonzept involviert [Jubin et al., 1997, S. 24 ff.]:

- Event Object
- Event Listener
- Event Listener Interface
- Event Source

3.3.1.1 Event Object

Das Event Object kapselt die Informationen, die eine Event Source den registrierten Event Listeners übergibt. Ein Event Object wird von der Event Source instanziiert und jedem registrierten Event Listener gesendet, wenn ein Ereignis ausgelöst wird.

Event-Objekte erben von der Klasse `java.util.EventObject` und implementieren zumindest die Methoden `toString()` und `getSource()`. Die erste Methode liefert eine Stringrepräsentation des Events, die zweite Methode liefert eine Referenz auf die Event Source.

3.3.1.2 Event Listener

Event Listener sind Objekte, die Ereignisse empfangen und verarbeiten können. Um dies zu ermöglichen, müssen sie ein oder mehrere Event Listener Interfaces implementieren und sich bei der Event Source registrieren.

3.3.1.3 Event Listener Interfaces

Da die Eventbenachrichtigung in Java über Methodenaufrufe erfolgt, muß sichergestellt sein, daß der Event Source die Methodennamen zur Ereignisbearbeitung der Event Listener bekannt sind. Über ein sogenanntes Event Listener Interface wird dies erreicht. Ein Event Listener, der ein bestimmtes Event empfangen und bearbeiten will, muß das entsprechende Interface implementieren.

Ein Event Listener Interface erbt von der Klasse `java.util.EventListener`. Der Name des Interfaces soll mit `...Listener` enden. Die definierten Methoden haben als einzigen Parameter ein Event-Objekt.

3.3.1.4 Event Source

Mit Event Source werden alle Objekte bezeichnet, die Ereignisse senden können. Die Event Source ist dafür verantwortlich, für jedes Ereigniss, daß von ihr gefeuert werden kann, eine Liste der registrierten Event Listener zu halten. Mit Hilfe dieser Listen und der durch das Event Listener Interface definierten Methoden, kann die Event Source die Event Listener lokalisieren und die entsprechenden Methoden aufrufen.

Um die Registrierung zu ermöglichen, muß die Event Source für jedes Ereigniss „add“- und „remove“-Methoden bereitstellen, wobei der ListenerType dem Listener Interface des Ereignisses entspricht:

```
public void addListenerType (ListenerType aListener);
public void removeListenerType (ListenerType aListener);
```

3.3.2 Properties

Ein Property ist ein einzelnes öffentliches Attribut. Diese Attribute bestimmen das Verhalten und das Erscheinungsbild einer JavaBean. Mögliche Zugriffsarten auf Properties sind read/write, read-only oder write-only. Get und set-Methoden ermöglichen den Zugriff und legen die Zugriffsart fest. Ist beispielsweise lediglich eine get-Methode für ein Property implementiert, liegt ein read-only Zugriff vor. Die unterschiedlichen Typen von Properties werden in den folgenden Abschnitten dargestellt [DeSoto, 1997, S. 2-1 ff.].

3.3.2.1 Simple Properties

Ein Simple Property repräsentiert einen einzigen Wert. Der Zugriff auf diesen Wert erfolgt über einfache get- und set-Methoden. Der Methodenname orientiert sich an dem Namen des Properties. Auf ein Property mit Namen „x“ wird über die Methoden setX und getX zugegriffen. Attribute des Typs Boolean können zusätzlich die Methode isX enthalten.

Beispiel:

```
String name;
String getName();
void setName(String aName);
```

3.3.2.2 Indexed Properties

Ein Indexed Property stellt einen Vektor dar. Mittels der get- und set-Methoden können einzelne Werte des Vektors über einen Index verändert werden. Weitere, optionale get- und set-Methoden erlauben das Lesen bzw. Schreiben des gesamten Vektors.

Beispiel:

```
String name[];
```

```
String getName(int anIndex);
void setName(int anIndex, String aName);
String[] getName();
void setName (String[] names);
```

3.3.2.3 Bound Properties

Ein Bound Property benachrichtigt andere Objekte, wenn es seinen Wert ändert. Bei jeder Veränderung des Wertes wird ein PropertyChange Ereignis gefeuert, welches den Property-Namen, sowie die alten und neuen Werte beinhaltet. Interessierte Objekte müssen sich bei der JavaBean explizit anmelden (vgl. Kap. 3.3.1). Die Benachrichtigungsgranularität richtet sich nach der Bean, nicht nach dem Property; d.h. PropertyChangeListener melden sich bei der Bean und nicht bei den einzelnen Properties an.

3.3.2.4 Constrained Properties

Ein Objekt mit Constrained Properties erlaubt anderen Objekten ein Veto gegen die Änderung des Wertes eines Constrained Properties auszusprechen. Ein Veto wird durch eine PropertyVetoException des ConstrainedPropertyListeners eingelegt. In diesem Fall behält das Property seinen alten Wert bei.

3.3.3 Methoden

Die Methoden der öffentlichen Schnittstelle enthalten die Logik einer Bean. Sie sind anderen Beans oder der Applikation zugänglich. Für Methoden sind keine Designpatterns vorgesehen, sie können also beliebige Namen und Parameterlisten erhalten. Private Methoden können ebenso in einer Bean vorkommen, sie zählen aber nicht zur öffentlichen Schnittstelle und sind lediglich der Bean selbst zugänglich.

3.4 Introspection

Introspection ist der Prozeß, den Java nutzt, um die öffentliche Schnittstelle einer Bean zu analysieren. Zu dieser öffentlichen Schnittstelle gehören die Properties, die öffentlichen Methoden und die Events. Introspection arbeitet auf Objekten, d.h. der Source-Code einer Bean wird nicht benötigt. Ein Entwicklungswerkzeug kann mit ihrer Hilfe ein Objekt im Bytecode und ohne zusätzliche Informationen des Entwicklers analysieren.

3.4.1 BeanInfo-Klasse

Alle Informationen über die Properties, die Methoden und die Events einer Komponente befinden sich in einem BeanInfo-Objekt [Hamilton, 1997, S. 59 ff.]. Über diverse Methoden dieses Objekts ist es einem Entwicklungswerkzeug möglich, die Informationen zu erhalten. Ein Bean-Entwickler kann diese Methoden explizit in einer BeanInfo-Klasse implementieren. Es ist jedoch nicht nötig, alle Methoden, die diese Informationen enthalten, explizit anzugeben. Ist eine Methode nicht angegeben, wird die Information durch den Introspector ergänzt. Die folgenden Informationen können aus den Methoden (bzw. Deskriptoren) eines BeanInfo-Objekts ermittelt werden:

- `getBeanDescriptor()`
  ermittelt den Namen, die Java Klasse, etc. der JavaBean.
- `getPropertyDescriptors()`
  ermittelt die Properties der öffentlichen Schnittstelle der JavaBean.
- `getMethodDescriptors()`
  ermittelt die Methoden der öffentlichen Schnittstelle der JavaBean.
- `getEventSetDescriptors()`
  ermittelt die Events der JavaBean.
- `getCustomizer()`
  ermittelt den Customizer der JavaBean.
- `getIcon()`
  ermittelt das Icon zur Repräsentation der JavaBean im Entwicklungswerkzeug.
- Weitere Methoden für zusätzliche Informationen.

Zu jeder BeanInfo-Klasse existiert genau eine zugehörige Bean-Klasse. Der Name der BeanInfo-Klasse entspricht dem Namen der Bean erweitert um `...BeanInfo`.

3.4.2 Die Klasse Introspector

Die Klasse `java.beans.Introspector` ist ein Weg für eine Applikation, Informationen über eine Bean zu erhalten. Eine Instanz dieser Klasse sucht zu einer Bean die korrespondierende BeanInfo-Klasse. Ist diese nicht vorhanden, so nutzt der Introspector das Reflection-API von Java und die Designpatterns der Namensgebung von Methoden und

Events der JavaBeans-Spezifikation, um die benötigte Information zu generieren. Kann eine BeanInfo-Klasse gefunden werden, so werden nur die fehlenden Deskriptoren ergänzt. Das Ergebnis der Introspection ist eine BeanInfo-Klasse, die nun teilweise benutzerdefinierte und durch den Introspector generierte Informationen enthält. Kann keine BeanInfo-Klasse gefunden werden, generiert der Introspector alle Informationen und liefert sie in einem BeanInfo-Objekt zurück.

3.5 Customization

Customization dient der Veränderung des Verhaltens und des Erscheinungsbilds einer Bean. Über Customization können die Properties an individuelle Bedürfnisse angepaßt werden. Es gibt zwei Arten, eine Bean anzupassen: das Property Sheet und den Customizer. Während das Property Sheet alle editierbaren Properties der Bean darstellt und unabhängig voneinander Änderungen erlaubt, kann ein Customizer selektiv Properties darstellen und Interdependenzen bei Wertänderungen erlauben. Die beiden folgenden Abschnitte beschreiben diese beiden Möglichkeiten [Jubin et al., 1997, S. 58 ff.].

3.5.1 Property Sheet

Jede Bean besitzt ein Property Sheet. Es enthält für jedes Property einen Property Editor, über den der Wert eines Properties gelesen und verändert werden kann. Wenn eine Bean im Container des Entwicklungswerkzeuges den Focus erhält, so wird automatisch das zugehörige Property Sheet geöffnet.

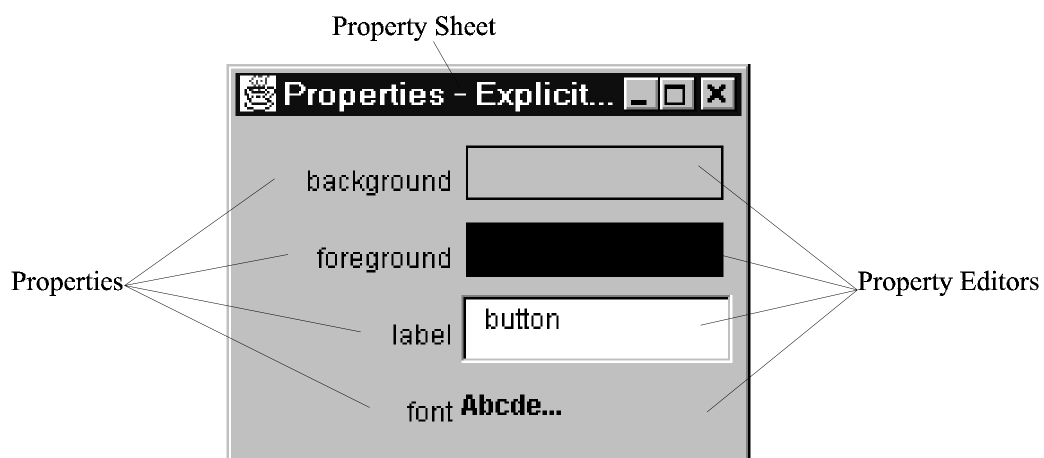

Abbildung 4 Property Sheet

Für die Generierung des Property Sheets ist ebenfalls das Entwicklungswerkzeug zuständig. Zunächst werden aus dem Property Descriptor des BeanInfo-Objekts die Properties und ihre zugehörigen Property Editoren ermittelt. Ist zu einem Attribut kein Property Editor explizit angegeben, so muß zu dem Typ des Attributs ein korrespondierender Property Editor ermittelt werden. Die zu verschiedenen Datentypen existierenden Property Editoren werden über einen Property Editor Manager registriert und stehen dann dem Werkzeug zur Verfügung. Die JavaBeans-API stellt 10 Default Property Editoren bereit:

- boolean
- byte
- short
- int
- long
- float
- double
- String
- Color
- Font

Eine Klasse, die einen Property Editor darstellt, muß das Interface `java.-beans.PropertyEditor` implementieren oder die Klasse `java.beans.Property-EditorSupport` erweitern. Im Property Descriptor des Attributs, das diesen Editor verwenden soll, muß nun noch diese Klasse angegeben werden. Das Entwicklungswerkzeug baut nun diesen Editor für das dazugehörige Property in sein Property Sheet ein.

3.5.2 Customizer

Ein Customizer eignet sich für Situationen in denen eine höhere Logik beim Editieren der Properties notwendig ist. Dies ist z.B. bei Interdependenzen zwischen einzelnen Properties der Fall. Das Property Sheet kann diese nicht berücksichtigen. Der Customizer, sofern dieser angeboten werden soll, muß vom Entwickler der Bean implementiert werden. Im Gegensatz zu den Property Editoren, die für jeweils ein Property gelten, existiert ein Customizer für die gesamte Bean. Dieser muß nicht zwangsweise alle Properties enthalten, er kann z.B. auch nur die interdependenten Properties behandeln. Das Entwicklungswerkzeug muß den Customizer

nicht automatisch bei Fokussierung einer Bean öffnen. Es muß lediglich dafür sorgen, daß dieser zugänglich ist.

Eine Customizer Klasse muß das Interface `java.beans.Customizer` implementieren. Über die Methode `getCustomizerClass()` der BeanInfo Klasse muß die Cunstomizer Klasse bekannt gegeben werden.

3.6 Persistenz

Persistenz ist die Sicherung des Zustands eines Objektes, d.h. seiner zustandsbestimmenden Attribute. Hierdurch kann ein Objekt jederzeit wiederhergestellt werden. Es wird hierzu neu instanziiert und mit dem gesicherten Zustand initialisiert [Jubin et al., 1997, S. 78 ff.].

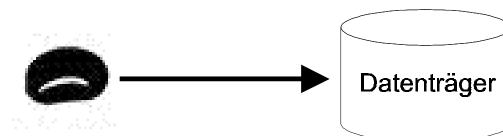

Abbildung 5 Persistenz einer Bean

Um die Persistenz einer Bean zu sichern sieht die JavaBeans-Spezifikation zwei Möglichkeiten vor: Serialization und Externalization. Serialisierbare Klassen speichern ihre Felder automatisch über den Object Serialization Process. Externalisierbare Klassen müssen das Format des Streams, der den Zustand eines Objektes der Klasse repräsentiert, und die darin enthaltenen Daten selbst bestimmen.

Unter Deserialization wird der Prozeß verstanden, der den Zustand eines Objektes aus dem Bytestream wiederherstellt.

3.6.1 Serialization

Durch den Prozeß der Serialization wird ein Objekt in einen Bytestream, der den momentanen Zustand repräsentiert, transformiert. Die Serialization ermittelt eine eindeutige Repräsentation einer Klasse (UID) und schreibt diese gemeinsam mit dem Namen der Klasse an den Anfang des Streams. Danach werden die Namen, die Typen und die Werte aller Attribute notiert.

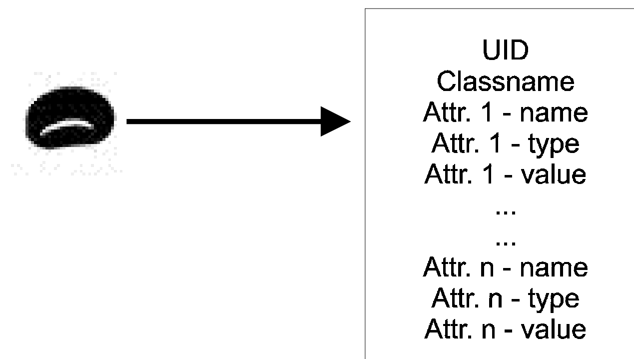

Abbildung 6 Serialization process

Um eine Klasse serialisierbar zu machen, muß diese das Interface `Serializable` implementieren. Eine solche Klasse kann über die `writeObject(Object serObject)`-Methode der Klasse `ObjectOutputStream` serialisiert werden. Eine serialisierbare Klasse kann aber auch eine eigene writeObject-Methode definieren, die z.B. weitere Informationen, wie ein Serialisierungsdatum enthalten kann. Attribute, die mit dem Schlüsselwort `transient` gekennzeichnet sind, werden bei der Serialisierung übergangen.

Zur Deserialisierung einer Klasse wird die Methode `readObject()` verwendet. Hierbei werden zunächst der Klassenname und die UID verglichen, danach werden die Felder eingelesen. Die Reihenfolge der Attribute spielt keine Rolle. Fehlende Attribute im Bytestream werden im Objekt mit ihrem Defaultwert initialisiert. Sind Attribute im Bytestream, nicht aber im Objekt enthalten, werden diese übergangen. Da im Bytestream neben den Attributwerten auch ihre Namen und Typen gespeichert sind, können alle Werte eindeutig zugeordnet werden.

3.6.2 Externalization

Der Prozeß der Externalization transformiert ebenfalls ein Objekt in einen Bytestream, der den momentanen Zustand enthält. Hierbei ist jedoch der Bean-Entwickler für das Format des Bytestreams und die zu speichernden Informationen selbst zuständig. Es ist auf diese Weise z.B. möglich, lediglich die Attributwerte zu sichern, um Speicherplatz und Zeit einsparen zu können. Der Entwickler muß jedoch sicherstellen, daß der Prozeß der Deserialization diesen Stream verstehen kann.

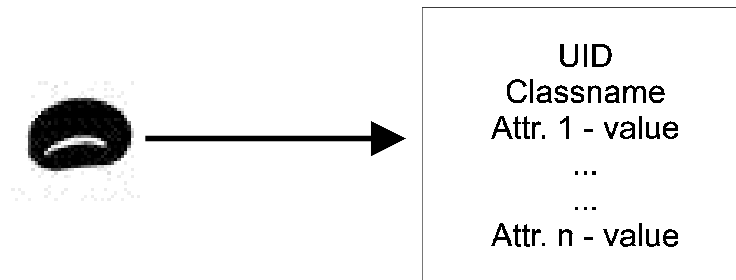

Abbildung 7 Externalization process

Um eine Klasse externalisierbar zu machen, muß diese das Interface `Externalizable` implementieren. Eine solche Klasse muß ebenfalls die `writeExternal(ObjectOutput anOutput)`-Methode implementieren und kann über sie externalisiert werden.

Die Dezerialisation erfolgt über die `readExternal(ObjectOutput anOutput)`-Methode der Klasse. An dieser Stelle ist darauf zu achten, daß entsprechend der Konventionen der writeExternal-Methode die Werte und andere Informationen den Attributen zugeordnet bzw. verarbeitet werden. Eine automatische Zuordnung von Werten zu Attributen, wie bei der Serialization, kann hier nicht erfolgen.

4 Suns Erweiterungen des JavaBean Frameworks

4.1 JavaBeans Activation Framework

Das JavaBeans Activation Framework (JAF) ist im März 1998 in der aktuellen Version 1.0 von Sun Microsystems veröffentlicht worden [Calder et al., 1998]. Es stellt eine Ergänzung zum JavaBeans-Komponentenmodell dar, die aber insbesondere für Entwickler interessant sein dürfte, die Stand-Alone-Komponenten zum Editieren und Generieren von Daten entwickeln.

Dieses Framework stellt ein Modell zur Verfügung, das folgendes leisten soll:

- eine konsistente Strategie zur Typisierung von Daten
- eine Methode, die es ermöglicht, die von einer Komponente unterstützten Datentypen festzustellen
- eine Methode, die Daten mit einem gewissen Typ an eine dafür vorgesehene Komponente bindet
- Architektur und Implementierung der oben genannten Features

Um die genannten Leistungen zur Verfügung stellen zu können, implementiert das JAF die folgenden Dienste:

- Bestimmung des Typs von beliebigen Daten
- Kapselung des Zugriffs auf diese Daten
- Feststellung der möglichen Operationen auf diesen Daten
- Instanziierung der Softwarekomponente, die eine gewünschte Operation auf bestimmten Daten ausführen kann

4.1.1 Architektur

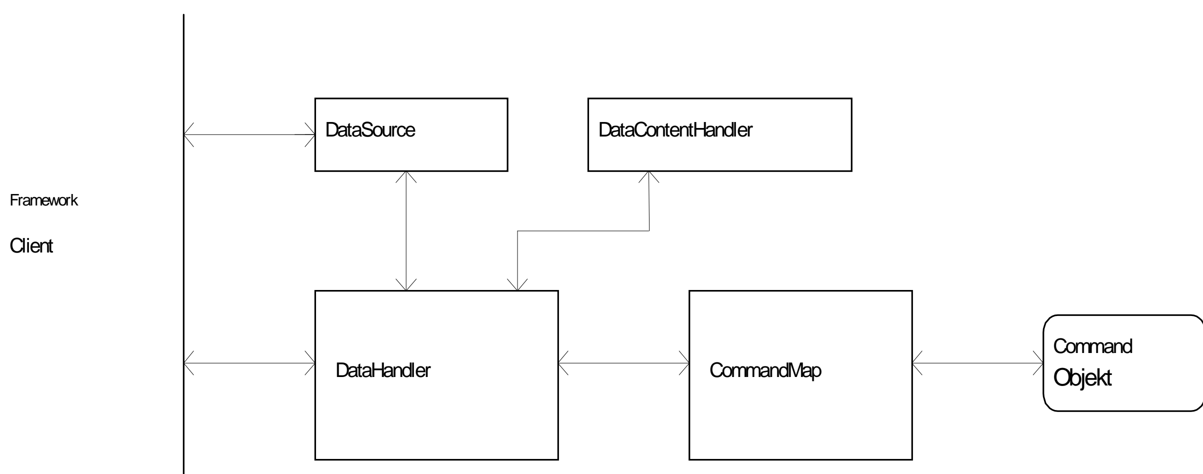

Abbildung 8: Architektur des JavaBeans Activation Frameworks

4.1.1.1 Die DataHandler-Klasse

Ein Objekt der `DataHandler`-Klasse stellt die Schnittstelle eines Clients zu den Subsystemen des Frameworks dar. Es bietet den Zugriff auf die Daten an und stellt die Methoden zur Verfügung, die auf diesen Daten arbeiten können. `DataHandler`-Objekte können nur zusammen mit Daten instanziiert werden. Die Daten können dabei in Form von Objekten, die das `DataSource`-Interface implementieren (die bevorzugte Variante), in Form von Objekten mit einem assozierten `DataContentHandler`-Objekt oder als `java.net.URL`-Objekte vorliegen.

Ein `DataHandler`-Objekt ist genau einer Datenquelle zugeordnet.

4.1.1.2 Das DataSource-Interface

Ein Objekt, das das `DataSource`-Interface implementiert, bietet eine Schnittstelle, die von der tatsächlichen Speicherstruktur der Daten abstrahiert ist. Diese Schnittstelle stellt dem Benutzer dieser Daten, nämlich z.B. der `DataHandler`-Klasse, den MIME-Typ [Borenstein et al., 1993] der Daten, sowie die Daten selbst als `java.io.InputStream`-Objekt und somit in einer standardisierten Form zur Verfügung.

Ein `DataSource`-Objekt entspricht genau einer Datenquelle, das heißt einer Datei oder einer URL.

4.1.1.3 Das DataContentHandler-Interface

Wenn es sich bei den zu behandelnden Daten schon um ein Objekt handelt, das bereits instanziiert worden ist und im Hauptspeicher liegt, macht es nicht notwendigerweise Sinn, dieses Objekt in ein `InputStream`-Objekt zu konvertieren, wenn die Bean, die mit diesen Daten arbeiten soll, auch das im Speicher liegende Objekt selbst verarbeiten könnte. Aus diesem Grund kann, wie oben erwähnt, ein `DataHandler`-Objekt auch ein beliebiges Objekt als Datenquelle benutzen. In diesem Fall wird der MIME-Typ der Daten bei der Instanziierung des `DataHandler`-Objekts im Konstruktor angegeben. Es kann aber sein, daß nicht alle Beans mit diesem Objekt arbeiten können und statt dessen auf ein `InputStream`-Objekt angewiesen sind. Für diese Komponenten muß dann zusätzlich ein `DataSource`-Objekt zur Verfügung stehen. Die Aufgabe eines `DataContentHandler`-Objekts ist es nun, die notwendigen Konvertierungen von einem Objekt zu einem Bytestream und umgekehrt durchzuführen.

4.1.1.4 Das CommandMap-Interface

Das `DataHandler`-Objekt kennt den MIME-Typ der mit ihm verbundenen Datenquelle. Es kann das `CommandMap`-Objekt nach den für diesen MIME-Typ verfügbaren „Befehle“ abfragen, wobei diese Befehle nichts anderes als ausführbare Komponenten und letztendlich JavaBeans sind. Das `CommandMap`-Objekt stellt also eine Component Registry dar. Eine mögliche Implementierung dieses Interfaces kann die Type Registry der Plattform abfragen oder eine serverbasierte Lösung sein. Eine weitere Möglichkeit ist die Auslieferung der Beans zusammen mit einem .mailcap-File nach RFC 1524 [Borenstein, 1993], das ebenfalls die Typinformationen enthält.

4.1.1.5 Die CommandInfo-Klasse

Ein `CommandInfo`-Objekt stellt die Verbindung zwischen dem Befehl, mit dem eine Komponente aufgerufen wird, z.B. „Öffnen“ und der dazugehörigen Klasse her. Es gestattet der Applikation, die entsprechende Bean zu instanziieren oder den ihr zugeordneten Befehl abzufragen.

4.1.1.6 Das CommandObject-Interface

Eine JavaBean implementiert das `CommandObject`-Interface, um mit den Diensten des JAF zusammenarbeiten zu können. Es ermöglicht einer Bean direkten Zugriff auf die Methoden des `DataHandler`-Objekts, außerdem erfährt die Bean dann, mit welchem Befehl sie aufgerufen worden ist.

4.1.2 Anwendungsbereich

Als Anwendungsbereich könnte man sich z.B. eine Applikation vorstellen, die, ähnlich wie der Windows Explorer oder der Dateimanager „dtfile“ unter UNIX, Dateien anzeigt und kontextsensitiv Befehle für verschiedene Dateien zur Verfügung stellt.

Wenn die Applikation dann den Inhalt eines Verzeichnisses anzeigen soll, erzeugt sie für jede dort aufgeführte Datei ein Objekt, das das `DataSource`-Interface implementiert. Zu jedem dieser Objekte wird eine Instanz der Klasse `DataHandler` erzeugt. Diese Klasse hat Zugriff auf das dazugehörige `CommandMap`-Objekt, das, so die Spezifikation des JAF, in „some ‘common’ place“ gespeichert ist. Die Applikation kann nun das `DataHandler` Objekt nach dem Namen der Datenquelle und dem dazugehörigen Icon abfragen.

Wählt nun der Benutzer der Applikation eine Datei aus, so erfragt die Applikation beim zugehörigen `DataHandler`-Objekt die Liste der für diese Datei zur Verfügung stehenden Befehle. Das `DataHandler`-Objekt selbst konsulitiert dafür das `CommandMap`-Objekt. Dieses liefert ein Array von `CommandInfo`-Objekten zurück, das an die Applikation durchgereicht wird.

Die Applikation kann nun die zur Verfügung stehenden Befehle anzeigen. Wählt der Benutzter einen dieser Befehle aus, ruft die Applikation in dem `CommandInfo`-Objekt die Methode `getCommandObject(DataHandler dh, ClassLoader cl)` auf, die die entsprechende Klasse instanziiert. Alternativ kann sie in dem entsprechenden

`DataHandler`-Objekt die Methode `getBean(CommandInfo binfo)`, die dann neben der Instanziierung auch die Eigenschaften der Bean festlegen kann, sofern sie das `CommandMap`-Interface implementiert, aufrufen.

4.1.3 Resümee

Das JAF kann das, wozu es entwickelt wurde, leisten, und es ist auch notwendig, daß eine Lösung für diese Lücke im Konzept von JavaBeans gefunden wurde.

Ein Problem dürfte jedoch die relativ undurchsichtige Struktur darstellen, da doch eine recht große Anzahl von Interfaces und Klassen für den beabsichtigten Zweck spezifiziert worden ist. Diese Interfaces müssen allesamt implementiert werden, was ebenfalls einen nicht zu unterschätzenden Aufwand darstellt.

Etwas schwammig stellt sich auch die Lösung für die Frage dar, wo die zentrale Component Registry denn nun gespeichert, bzw. wie sie aufgebaut wird. Daß Sun verschiedene Möglichkeiten offenläßt, ist zwar auf der einen Seite recht lobenswert, da so das Problem an die lokalen Gegebenheiten angepaßt gelöst werden kann, auf der anderen Seite jedoch kann es dazu führen, daß gewisse Inkompatibilitäten in der Art, wie sich z.B. Komponenten in der Component Registry registrieren, zur Folge haben, daß keine Component Registry entstehen kann, die alle in dem System zur Verfügung stehenden Komponenten kennt. Außerdem muß die Frage gestellt werden, inwiefern die Nutzung der Type Registry der lokalen Plattform dem Gedanken der Plattformunabhängigkeit zuwider läuft.

Wer die Funktionalität benötigt, die im JAF spezifiziert worden ist, hat nun zumindest eine Vorgabe, wie die Implementierung auszusehen hat, damit seine Klassen auch bei der Entwicklung anderer Programme wiederverwendet werden können.

4.2 InfoBus

Schon kurze Zeit nach der Vorstellung der JavaBeans bemerkten die Entwickler von Sun eine Schwachstelle in ihrem Komponentenkonzept. Man hatte keinen Mechanismus vorgesehen, der es Beans verschiedener Hersteller auf einfache Art und Weise erlaubte, Daten auszutauschen. Solche Cross-Bean-Aufrufe basierten auf benutzerdefinierten Schnittstellen oder Basisklassen. Die Beans verwendeten den „Introspection“-Mechanismus, um zur Laufzeit bestimmte Schnittstellen zu finden. Um dies zu ändern, entwickelte man ein

Softwarekonzept, das es erlauben sollte, die Kommunikation zwischen Beans auf einfache Art und Weise zu etablieren: den InfoBus [Colan, 1998].

Der InfoBus, der zum Zeitpunkt der Erstellung dieser Arbeit in der Version 1.1 vorliegt und nach Erscheinen des JDK 1.2 um einige Features erweitert in Version 1.2 veröffentlicht werden wird, stellt eine Kommunikationsinfrastruktur, insbesondere für Java Beans, dar, die aber auch von beliebigen Nicht-Bean-Objekten genutzt werden kann. Dabei registrieren sich Java-Objekte, die auf die Kommunikation mit dem bzw. über den InfoBus vorbereitet sein müssen, als Datenproduzenten (Producer), Datenkonsumenten (Consumer) oder Controller beim InfoBus. Producer sind Objekte, die bestimmte Daten anbieten; Consumer sind Objekte, die über die, über den InfoBus zur Verfügung stehende Daten, informiert werden wollen. Controller wiederum sind beispielsweise in der Lage, den Datenfluß auf dem Bus zu kontrollieren oder Prioritäten für verschiedene Datentypen zu verteilen. Objekte können dabei auch gleichzeitig als Producer und als Consumer auftreten.

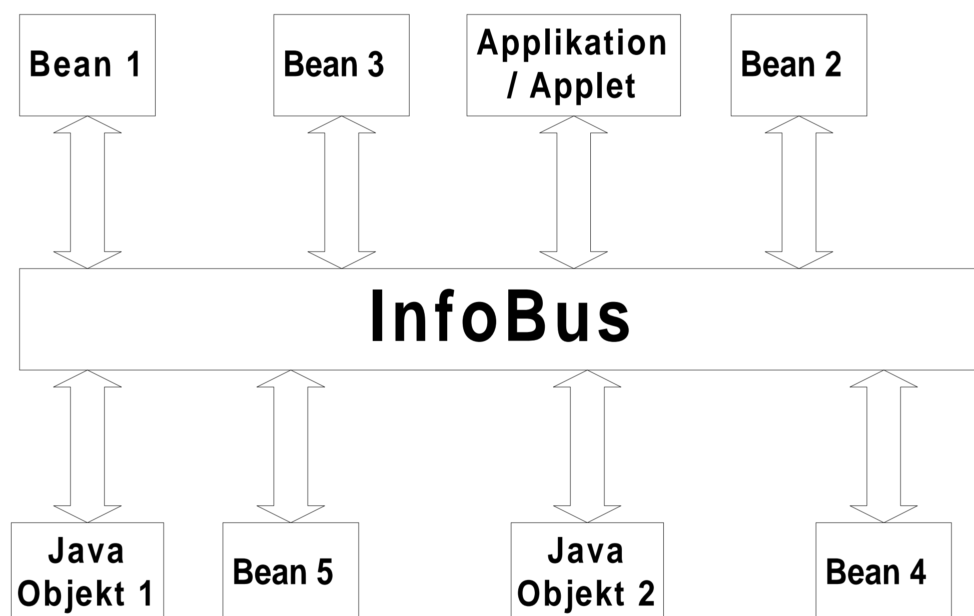

Abbildung 9 Die Verwendung des InfoBuses

4.2.1 Das InfoBus-Protokoll für den Datenaustausch

Schritt 1: Mitgliedschaft – Einrichten der Verbindung zum InfoBus

Jedes Java-Objekt kann eine Verbindung zum InfoBus einrichten. Dies geschieht, indem das Objekt das `InfoBusMember`-Interface implementiert, eine Instanz des InfoBuses ermittelt und sich bei diesem als Mitglied anmeldet.

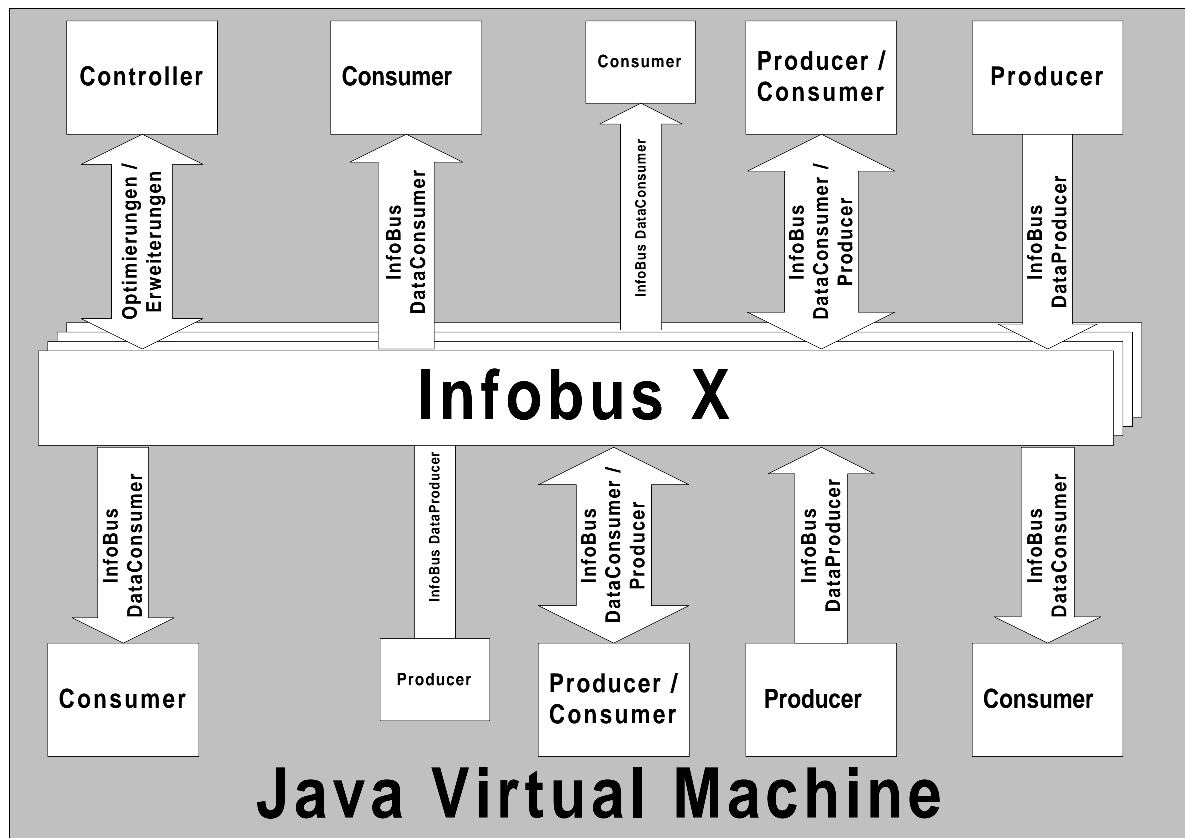

Abbildung 10 Die Mitglieder des InfoBuses

Schritt 2: Überwachen der InfoBus-Events

Sobald ein Objekt Mitglied eines InfoBuses ist, empfängt es Bus-Benachrichtigungen, indem es ein spezielles Interface implementiert und dieses beim InfoBus registriert. Es gibt zwei Event-Listener-Interfaces, die für die beiden grundlegenden Typen von InfoBus-Anwendungen ausgelegt sind. Ein Datenkonsument („Data Consumer“) empfängt Mitteilungen über die Verfügbarkeit von Daten, indem er einen Consumer-Listener, d.h. eine Implementierung des `InfoBusDataConsumer`-Interfaces, beim InfoBus registriert. In gleicher Weise empfängt ein „Data Producer“ Anfragen bezüglich bestimmter Daten, indem er einen Producer-Listener beim Bus registriert.

Schritt 3: Rendezvous-Konzept für den Datenaustausch

In der InfoBus-Architektur geben „Data Producer“ die Verfügbarkeit eines neuen Datums über den Bus bekannt, sobald das neue Datum verfügbar ist, d.h. wenn beispielsweise das Lesen einer URL beendet ist, eine Berechnung abgeschlossen ist oder ähnliches.

Konsumenten fordern Daten immer dann an, wenn sie diese benötigen. Das Rendezvous findet über den Namen des Datums statt. Es ist Aufgabe des Anwendungs- bzw. Komponentendesigners, den Daten, die über den Bus austauschbar sind, Namen zuzuweisen.

Folglich müssen alle „Data Producer“ und „Data Consumer“ einen Mechanismus zur Verfügung stellen, der es Anwendungsdesignern erlaubt, den Namen eines Datums für das Rendezvous festzulegen. Beispielsweise muß eine Tabellenkalkulationskomponente die Möglichkeit bieten, daß ein bestimmter Bereich des Arbeitsblattes benannt wird und dieser Bereich im Rahmen der „Data Producer“-Rolle der Komponente über seinen Namen exportiert werden kann. Entsprechend muß eine Geschäftsgrafikkomponente eine Möglichkeit bieten, daß ihr die Namen der Daten, die sie anzeigen soll, mitgeteilt werden.

Schritt 4: Navigation auf strukturierten Daten

Verschiedene „Data Producer“ verwenden häufig sehr unterschiedliche interne Repräsentationen für Daten, die eigentlich eine identische Struktur besitzen. Beispielsweise verwendet sowohl eine Tabellenkalkulation als auch eine Datenbank Tabellen. Beide Anwendungen speichern diese aber sehr unterschiedlich. Die Tabellenkalkulation könnte die Daten als Array von Formeln speichern, während die Datenbank dieselbe Information als Ergebnis eines Datenbank-Joins repräsentieren könnte.

Ein Datenkonsument sollte nicht gezwungen sein, die interne Repräsentation der Daten auf Seite des „Data Producers“ zu kennen. Eine Geschäftsgrafikkomponente sollte in der Lage sein eine Grafik aus den Daten einer Tabelle, sowohl aus einer Tabellenkalkulation, als auch aus einer Datenbank zu erstellen, sofern sich die in der Tabelle enthaltenen Daten sinnvoll als Grafik darstellen lassen. Praktisch bedeutet dies, daß diese Art des Information-Sharings Einigkeit über die Kodierung der Daten zwischen Konsument und Produzent voraussetzt. Daher wurden eine Reihe von Interfaces für verschiedene Standardprotokolle definiert, die verwendet werden sollen, um Daten zu erzeugen, auf die allgemein zugegriffen werden kann.

Schritt 5: Abfragen einer Codierung des „Wertes“ eines Datums

Ein Datum kann als String oder als Java-Objekt abgefragt werden. Java-Objekte sind häufig Wrapper für primitive Datentypen, wie „Double“ oder andere Kern-Klassen wie „Collection“.

Sinn dieser Implementierung ist es, ein nur möglichst unspezialisiertes Verständnis von Datenformaten auf Seite des Konsumenten vorauszusetzen.

Schritt 6: Optional: Ändern von Daten

Ein Konsument kann versuchen, den Wert eines Datums zu ändern. Der Produzent ist für die Implementierung einer Zugriffspolitik zuständig, um festzulegen, ob Daten überhaupt geändert werden dürfen und wer diese Daten ändern darf. Im JDK 1.2 könnte der Produzent darüber hinaus, im Rahmen des dort erweiterten Sicherheitskonzepts, die Zugriffsberechtigung eines Konsumenten überprüfen.

4.2.2 Die wichtigsten InfoBus-Schnittstellen

4.2.2.1 Die InfoBus-Klasse

Die `InfoBus`-Klasse enthält die eigentliche Implementierung des InfoBuses [Colan, 1998, S. 11 ff.]. Die Klasse darf ausschließlich in der unveränderten Originalversion benutzt werden, und es dürfen auch keine Subklassen von dieser Klasse verwendet werden. So soll die Kompatibilität zwischen Klassen, die den InfoBus nutzen, garantiert werden. Der InfoBus enthält u.a. Funktionen, die das Erzeugen von `InfoBus`-Instanzen ermöglichen. Diese werden jedoch i.d.R. nie direkt vom Entwickler aufgerufen, sondern unter Verwendung der `InfoBusMemberSupport`-Klasse (vgl. Kapitel 4.2.2.3) genutzt.

4.2.2.2 Das Interface InfoBusMember

Dieses Interface definiert die Schnittstelle, die ein InfoBus-Mitglied implementieren muß [Colan, 1998, S. 13 f.]. Darunter fällt u.a. die Methode zum Speichern des InfoBuses, der einem `InfoBusMember` zugeordnet ist und Methoden zum Registrieren bzw. Deregistrieren verschiedener Event-Listener.

4.2.2.3 Die Klasse InfoBusMemberSupport

Die `InfoBusMemberSupport`-Klasse implementiert einige Methoden, die es dem Entwickler eines InfoBus-Members erleichtern, die Methoden des `InfoBusMember`-Interfaces zu implementieren [Colan, 1998, S. 14 f.]. Der Entwickler muß nur noch Wrapper für die `InfoBusMember`-Interface-Methoden schreiben, die die korrespondierenden

Methoden der `InfoBusMemberSupport`-Klasse aufrufen. Dieser Umweg bei der Implementierung, auch Delegation genannt, ist aufgrund des Fehlens der Mehrfachvererbung in Java notwendig.

4.2.2.4 Das Interface InfoBusDataProducer

Dieses Interface muß von einem „Data Producer“ implementiert werden und spezifiziert die Methode `dataItemRequested`, die aufgerufen wird, wenn ein Datum von einem „Data Consumer“ nachgefragt wird [Colan, 1998, S. 24].

4.2.2.5 Das Interface InfoBusDataConsumer

Dieses Interface muß von einem „Data Consumer“ implementiert werden und spezifiziert die Methoden `dataItemAvailable` und `dataItemRevoked`, die aufgerufen werden, wenn ein Produzent ein neues Datum „anbietet“ bzw. bekannt gibt, daß er ein Datum nicht mehr zur Verfügung stellt [Colan, 1998, S. 24].

4.2.2.6 Das Interface DataItem

Dieses Interface spezifiziert die grundlegenden Methoden, die ein Datum, ein sogenanntes „Data Item“, implementieren muß [Colan, 1998, S. 25 ff.]. Zusätzlich zu diesem primitiven Interface existieren fünf erweiterte Interfaces, die spezielle Datenstrukturen unterstützen. Dies sind:

- `ImmediateAccess` – spezifiziert das Interface für den Zugriff auf ein relativ einfaches Datum, daß entweder als String oder als Objekt abgefragt werden kann.
- `ArrayAccess` – spezifiziert das Interface für den Zugriff auf Daten, die in Array-Form vorliegen.
- `RowsetAccess` – spezifiziert das Interface für den Zugriff auf Daten, die in Form von Rowsets vorliegen, d.h. in der Art, wie sie nach der Recherche über ein relationales Datenbankmanagementsystem vorliegen. Das Interface enthält Methoden zur Abfrage von Metadaten, zum Lesen von Zeilen, zum Abfragen von Spaltenwerte, zum Ändern von Zeilen, zum Überprüfen und zum Weiterleiten von Änderungen an die Datenbank und um ein `DbAccess`-Objekt, zum direkten Zugriff auf die Datenbank abzufragen.
- `ScrollableRowsetAccess` – implementiert ein Interface das den Zugriff auf Daten ermöglicht, die als eine Menge von Zeilen repräsentiert werden, auf denen vorwärts und

rückwärts gescrollt werden kann. Das Interface enthält Methoden um einen neuen Cursor zu erzeugen, um die Anzahl der lokal gepufferten Zeilen zu setzen bzw. abzufragen, um zur ersten, letzten, nächsten bzw. spezifizierten Zeile zu springen und um die Anzahl der Zeilen abzufragen.

- `DbAccess` – „Data Items", die das `DbAccess`-Interface implementieren, repräsentieren eine Datenbank. Das Interface enthält Methoden, um festzustellen, welche Argumente benötigt werden, um Zugriff auf die Datenbank zu erhalten, um die Verbindung mit der Datenbank auf- bzw. abzubauen, um Datenabfrage- und Update-Anfragen an die Datenbank zu senden, um Transaktionen zu kontrollieren und um Änderungen zu überprüfen, die in der Datenbank durchgeführt wurden.

4.2.3 Die InfoBus-Events

Die Kommunikation über den InfoBus erfolgt Java-üblich über Events [Colan, 1998, S. 16 ff.]. Über diese Events wird bekanntgegeben, daß ein bestimmtes Datum vorliegt und abgefragt werden kann (`InfoBusItemAvailableEvent`), daß ein bestimmtes Datum nicht länger zur Verfügung steht (`InfoBusItemRevokedEvent`) oder daß ein Konsument ein bestimmtes Datum sucht (`InfoBusItemRequestedEvent`). Diese Events werden durch Aufruf der Methoden `fireItemAvailable`, `fireItemRevoked` und `findDataItem` der InfoBus-Klasse gefeuert.

Angebotene und nachgefragte Daten werden über einen Namen spezifiziert. Zusätzlich zu diesem Namen kann man sogenannte `DataFlavors` angeben, die, ähnlich den MIME-Typen, den Typ des angebotenen Datums möglichst genau spezifizieren. Das Konzept der `DataFlavors` entstammt dem Kernsprachumfang Javas. Ein Datum kann mit mehreren `DataFlavors` belegt werden, wenn entweder der „Data Producer" in der Lage ist, das Datum in mehreren Formaten anzubieten, oder der „Data Consumer" ein Datum in mehreren Formaten verarbeiten kann.

4.2.4 Resümee

Der InfoBus stellt ein sehr interessantes Konzept dar, um die Kommunikation zwischen Java-Objekten im allgemeinen und die Kommunikation von Beans verschiedenster Hersteller im speziellen zu vereinfachen. Der ursprünglich auf eine Virtual Machine beschränkte InfoBus

ist über eine Erweiterung IBMs inzwischen auch in der Lage, sich über mehrere Virtual Machines zu erstrecken.

Der „Nachteil“ der Beans, daß man sie einzeln miteinander koppeln muß, damit sie zusammenarbeiten können und man dabei die Schnittstellen zwischen den einzelnen Beans adaptieren muß, tritt nun eher in den Hintergrund. Die Beans können direkt über einen InfoBus miteinander kommunizieren. Voraussetzung dafür ist jedoch eine möglichst große Anzahl von vordefinierten Schnittstellen für den Zugriff auf verschiedene Datenstrukturen und –typen, die von einer möglichst großen Anzahl von Entwicklern genutzt werden können.

4.3 Das Enterprise JavaBeans-Komponentenkonzept

Enterprise JavaBeans (EJB) ist eine Architektur für komponentenbasierte, verteilte Umgebungen. Enterprise Beans (EB) stellen Komponenten verteilter, transaktionsorientierter Anwendungen dar. Die folgenden Abschnitte erläutern das Konzept und die Architektur der Enterprise JavaBeans-Spezifikation. Die Ausführungen basieren auf [Matena et al., 1998] und [Thomas, 1997].

4.3.1 Grundkonzept und Ziele der Enterprise JavaBeans

Die Enterprise JavaBeans-Spezifikation stellt ein Komponentenmodell und ein Modell einer Komponentenausführungsumgebung dar. Derzeit liegt die Spezifikation in der Version 1.0 vor. Durch diese Architektur soll es einem Enterprise Bean-Entwickler ermöglicht werden, sich voll auf die eigentliche Logik der Applikation zu konzentrieren. Hierzu liegen Verträge in Form von Java Interfaces vor, die besagen, welche Konventionen der Entwickler zu befolgen hat, um eine reibungsfreie Zusammenarbeit mit der Ausführungsumgebung zu ermöglichen.

Die Ausführungsumgebung soll das Transaktionsmanagement, das Sicherheitsmanagement, das Persistenzmanagement, die Verteilung der Komponenten und den Zugriff auf knappe Ressourcen übernehmen.

Auch für Enterprise JavaBeans gilt das Write-Once-Run-Anywhere-Prinzip. Dies wird jedoch noch dadurch erweitert, daß eine Enterprise Bean ebenfalls unabhängig von z.B. dem zugrundeliegenden DBMS oder TMS auf jeder Maschine, die einen EJB-Server, der einen oder mehrere EJB-Container zur Verfügung stellt, ohne Anpassungen und mit gleicher Funktionalität ausführbar ist.

Die Enterprise JavaBeans erlauben eine Verteilung der Kompetenzen auf verschiedene Rollen:

- **Der Enterprise Bean-Provider**

  Ein Enterprise Bean-Provider ist typischerweise ein Domain-Experte. Er entwickelt spezifische, wiederverwendbare Komponenten, die einen Geschäftsprozeß oder eine Entität beschreiben. Er kann sich gänzlich auf die Geschäftslogik konzentrieren und muß sich dabei nicht um die Low-Level-Programmierung, wie z.B. Transaktionen, Sicherheit, Verteilung, etc. kümmern. Einem Client wird die Funktionaliät über Interfaces mitgeteilt.

- **Der Application-Assembler**

  Der Application-Assembler ist ebenfalls ein Domain-Experte. Er kombiniert jedoch vorhandene Enterprise Beans zu einer Anwendung. Dabei sieht er die Enterprise Bean nur aus der Sicht eines Clients. Die Funktionalität der Enterprise Bean sieht er über die bereitgestellten Interfaces.

- **Der EJB-Server-Provider**

  Der EJB-Server-Provider ist ein Experte in der Implementierung von Transaktionsmanagement, verteilten Objekten und weiteren Low-Level-System-Diensten. Ein typischer EJB-Server-Provider ist beispielsweise ein Vertreiber von Middleware oder DBMS.

- **Der EJB-Container-Provider**

  Der EJB-Container-Provider ist ein Experte in der System-Programmierung. Der Container ist für das Life-Cycle-Management einer Enterprise Bean verantwortlich. Er kontrolliert ebenfalls den Zugriff von Clients und stellt das Bindeglied zwischen EJB-Server und Enterprise Bean dar.

4.3.2 Grobarchitektur

Die Grobarchitektur wird durch die verschiedenen Rollen bereits umrissen. Die wesentlichen Bestandteile dieses Konzepts sind die Enterprise Beans, der EJB-Container, der EJB-Server und der Client. Während die Interoperabilität zwischen einem Container und einer Enterprise Bean in der Spezifikation 1.0 durch klar definierte Verträge geregelt wird, ist die Zusammenarbeit zwischen dem EJB-Server und einem EJB-Container nicht standardisiert.

Ein EJB-Server-Provider wird deshalb üblicherweise seine Low-Level-Interfaces anderen Parteien bekanntgeben. Der Client kann eine Java Applikation, ein Applet oder ein beliebiges CORBA-Objekt sein.

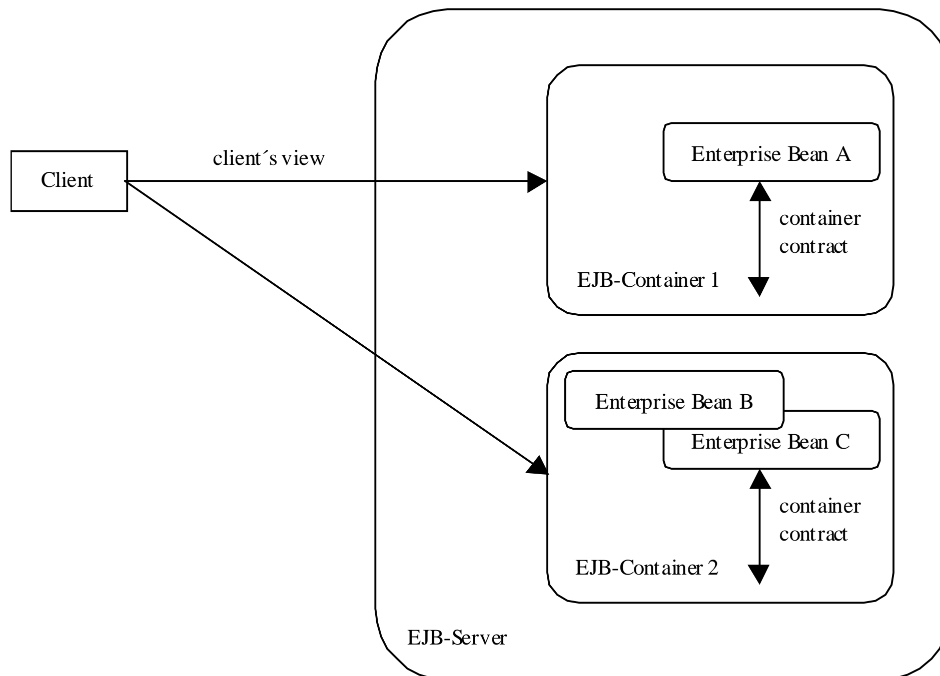

Abbildung 11 EJB-Grobarchitektur

Der Client erhält über den Container indirekten Zugriff auf die Enterprise Bean. Der Container kümmert sich, bevor er die gewünschte Methode der Enterprise Bean ausführt, z.B. um den Transaktionskontext, die Instanziierung einer Enterprise Bean oder das Securitymanagement. Transaktionen und weitere Low-Level-System-Dienste werden an den EJB-Server delegiert. Die Kommunikation zwischen Client und Server findet per RMI bei Java Clients oder per GIOP/IIOP bei CORBA Clients statt. Sun erwartet, daß viele EJB-Server auf dem CORBA-Standard aufbauen werden [Garg, 1998, S. 4].

4.3.3 Der EJB-Container aus Sicht des Providers

Der EJB-Container ist der „Lebensraum" der Enterprise Beans, in dem mehrere Enterprise Beans aus verschiedenen EJB-Klassen nebeneinander existieren können. Wie bereits erwähnt, greift der Client nicht direkt auf die Enterprise Beans zu, sondern über Objekte, die die `EJBObject`- und `EJBHome`-Interfaces implementieren. Mittels JNDI [JNDI, 1998] muß der Container dann das `EJBHome`-Objekt für den Client auffindbar machen.

Der Hersteller des EJB-Containers stellt Standardklassen zur Verfügung, während der Hersteller der Enterprise Beans die Interfaces `EJBObject` und `EJBHome` erweitert. Das erweiterte `EJBObject`-Interface wird dann als Remote Interface bezeichnet (vgl. Kap.4.3.4.1, 4.3.4.2).

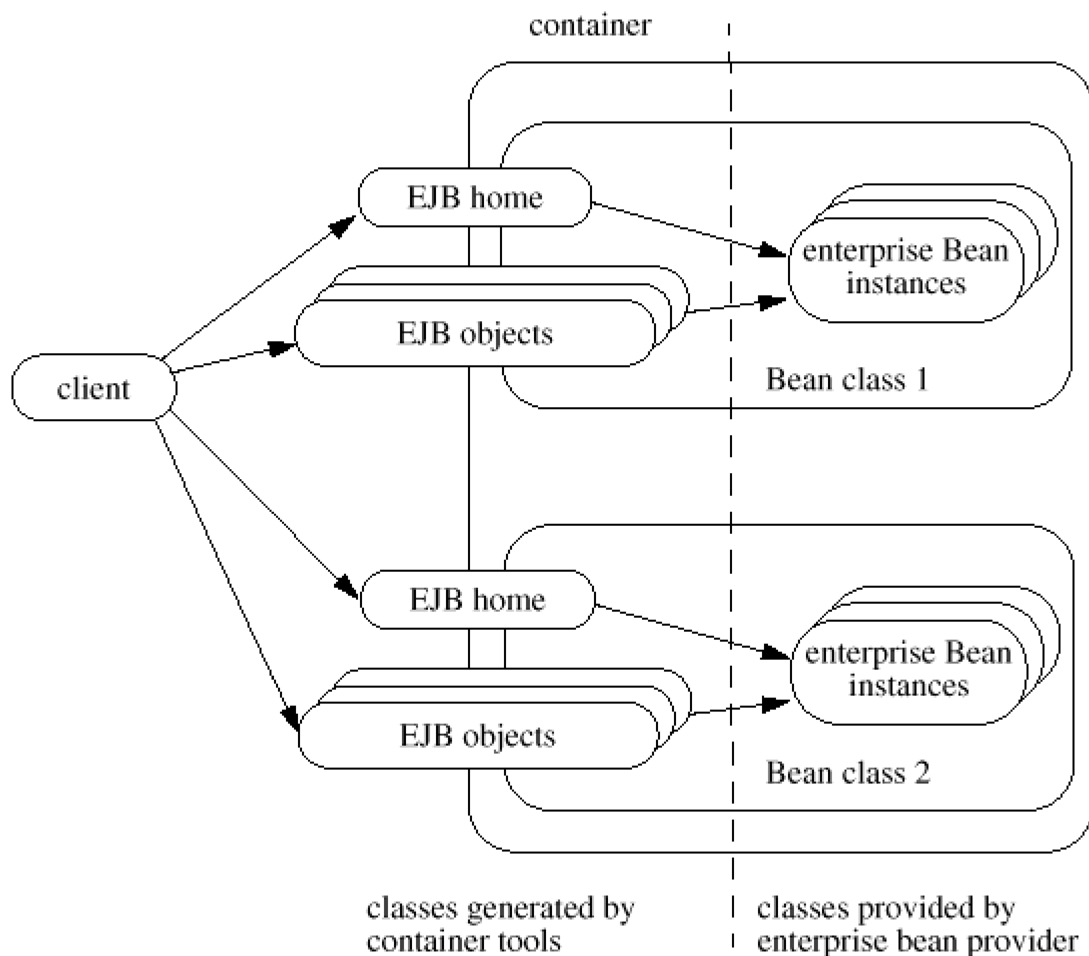

Abbildung 12 EJB Runtime Execution Model

Die Tools, die der Hersteller des Containers mitliefert, müssen dann folgende Aufgaben erfüllen können:

- Erzeugen einer Remote Bean-Klasse aus dem Remote Interface der Enterprise Bean und der Standardklasse. Das Resultat ist eine Wrapper Klasse für die Enterprise Bean, die dann die Sicht des Clients auf die Enterprise Bean darstellt.
- Erzeugen der Stubs und Skeletons für die Kommunikation der Remote Bean-Klasse.
- Erzeugen einer Bean Klasse, die an die herstellerspezifischen Bedürfnisse des Containers angepaßt ist. Dazu wird mit dem Container eine Standard Bean Klasse mitgeliefert, deren

Methoden im Rahmen der Interception, durch Aufrufe des Clients an die Enterprise Bean ausgeführt werden; unter Interception versteht man dabei die Unterbrechung des eigentlichen Methodenaufrufs zur Erledigung von Serviceleistungen durch den Container. Dies kann durch Vererbung, Delegation oder Codegenerierung geschehen. Während die Enterprise Bean-Klasse die Business Logik liefert, kommen aus der Standard Bean-Klasse noch einige spezifische Dienste des Containerherstellers hinzu.

- Generierung der Implementierung des `EJBHome`-Interfaces mit Hilfe der mitgelieferten Standardklasse.
- Erzeugen des Stubs und des Skeletons für die Home Klasse
- Erzeugen einer Klasse, die das Interface `EJBMetaData` implementiert. Diese Klasse ermöglicht das Abfragen der Metadaten einer Enterprise Bean, d.h. das Erfragen des `EJBHome`-Interfaces, des `Class`-Objekts des Home Interfaces, der Primary Key Klasse, sowie des Remote Interfaces und weiterhin, ob es sich um eine SessionBean handelt oder nicht. Dies ist hauptsächlich für die Unterstützung von Buildertools gedacht.

Im folgenden soll nun näher auf die Implementierung des `EJBHome` sowie des `EJBObject` Interfaces eingegangen werden. Danach wird das Transaktionsmanagement beschrieben, und es werden weitere Aspekte kurz erläutert.

4.3.3.1 Implementierung des EJBHome-Interfaces

Die Tools, die der Container Provider mit dem Container ausliefern muß, implementieren also unter anderem das `EJBHome`-Interface mit folgenden Methoden:

- die `find(...)`-Methoden, zum Auffinden einer Entity Enterprise Bean (vgl. Kap. 4.3.4.5). Da Entity Enterprise Beans persistente Objekte sind, werden diese Methode benötigt, um die Entity-Objekte, z.B. anhand ihres Primärschlüssel, wiederzufinden.
- die `create(...)`-Methoden, die es dem Client erlauben, EJB-Objekte zu instanziieren. Zwar erfolgt die Implementierung durch die Tools des Container-Providers, dieser schreibt jedoch nur einen Wrapper, der lediglich die entsprechende Methode der Enterprise Bean-Klasse aufruft. So jedoch besteht die Möglichkeit, daß der Container vor dem tatsächlichen Erzeugen des Enterprise Bean-Objekts noch zur Verwaltung notwendige Aufgaben erledigen kann. Eventuell kann der Container auf eine Instanziierung verzichten, nämlich dann, wenn es sich um eine zustandslose Session Bean oder eine Entity Bean handelt. Im

ersten Fall können der `create(...)`-Methode natürlich keine Parameter übergeben werden, da die Session Bean ja zustandslos sein soll. Wenn der Client nun `create()` aufruft, entscheidet der Container, ob er eine weitere Instanz dieser Session Bean erzeugen soll oder die vorhandenen nutzen kann (Loadbalancing). Im zweiten Fall kann der Container überprüfen, ob sich die gewünschte Entity Bean in einem Pool unbenutzter Beans befindet, der vom Container verwaltet wird. Dann muß der Container die Enterprise Bean lediglich vom „Pooled-Zustand“ in den „Ready-Zustand“ versetzen.

- die Methode `getEJBMetaData()`, die ein Objekt vom Typ `EJBMetaData` zurückliefert.
- die Methode `remove(Handle handle)`, die die Enterprise Bean, auf die das Handle verweist, zerstört.
- die Methode `remove(Object primaryKey)`, die die Entity Enterprise Bean mit dem angegeben Primärschlüssel zerstört.

4.3.3.2 Implementierung des EJBObject-Interfaces

Die folgende Methoden des `EJBObject`- bzw. Remote Interfaces müssen durch die Tools des Container-Providers implementiert werden:

- die Methoden, die die Business Logik der Enterprise Bean darstellen. Dafür werden wiederum Wrapper generiert, die einfach die entsprechenden Methoden der Enteprise Bean-Klasse aufrufen. Jedoch können an dieser Stelle vom Container Operationen, wie etwa zum Start einer Transaktion eingeschoben werden.
- die Methode `getEJBHome()`, die das korrespondierende Home Interface-Objekt zurückgibt.
- die Methode `getHandle()`, die ein Handle für das `EJBObject` zurückgibt. Mit dem Handle können `EJBObject`-Objekte später wieder referenziert werden, wenn das Handle serialisiert und auf einen persistenten Speicher geschrieben wurde. Voraussetzung dafür ist im Falle einer Session Bean jedoch, daß der Server in der Zwischenzeit dieses Objekt nicht aufgrund eines Timeouts zerstört und auch keinen Crash erlitten hat. Entity Beans hingegen müssen einen Crash des Servers unbeschadet überstehen.

- die Methode `getPrimaryKey()`, die den Primärschlüssel des `EJBObject`-Objekts zurückgibt.
- die Methode `isIdentical(EJBObject obj)`, die das übergebene `EJBObject`-Objekt mit dem aktuellen auf Identität vergleicht.
- die Methode `remove()`, die das `EJBObject`-Objekt und damit auch die Enterprise Bean zerstört.

4.3.3.3 Transaktionsmanagement

Hier ist grundsätzlich zwischen Session Enterprise Beans und Entity Enterprise Beans zu unterscheiden. Während Session Beans transaktional sein können, unterstützen Entity Beans grundsätzlich Transaktionen, da sie Daten, z.B. in einer Datenbank, unmittelbar repräsentieren. Session Beans hingegen können zwar auf Daten zugreifen, repräsentieren sie aber nicht.

Soll eine Session Bean an Transaktionen teilnehmen können, so wird sie vom Container in die Protokolle bezüglich der Transaktionssteuerung eingebunden. Sie muß jedoch selbständig auf die entsprechenden Mitteilungen, wie zum Beispiel ein „Prepare to commit“, reagieren. Im Deployment Descriptor einer Session Bean wird festgelegt, ob ihre Business Methoden in einer Transaktion ausgeführt werden sollen. Wenn dies der Fall ist, muß die Session Bean das `SessionSynchronization`-Interface implementieren. Dann kann sie nämlich vom Container über die Abläufe der Transaktion informiert werden, indem der Container die Methoden des Interfaces bei der Session Bean aufruft:

- `afterBegin()` wird aufgerufen, nachdem die Transaktion gestartet worden ist. Nun kann die Enterprise Bean die notwendigen Operationen auf der Datenbank durchführen.
- `beforeCompletion()` wird aufgerufen, wenn der Client seine Arbeit in der aktuellen Transaktion beendet hat, jedoch vor dem Commit. An dieser Stelle muß die Enterprise Bean spätestens die von ihr noch nicht geschriebenen Daten in die Datenbank speichern, sie kann aber auch an den Container mittels Aufruf der Methode `setRollbackOnly()` aus dem `SessionContext`-Interface signalisieren, daß die Transaktion abgebrochen werden soll.

- `afterCompletion(boolean committed)` wird aufgerufen, nachdem die Transaktion beendet worden ist. Wenn der Rückgabewert `true` ist, bedeutet dies ein Commit der Transaktion, im anderen Fall ein Rollback. Der Zustand der Session Bean muß im Fall eines Rollbacks eventuell zurückgesetzt werden. Diese Funktionalität übernimmt aber **nicht** der Container.

Entity Beans bieten, wie bereits oben angedeutet, eine Objektsicht auf persistente Daten. Diese können sowohl in einer Datenbank als auch in einer existierenden Enterprise Anwendung, z.B. einem Legacy System, das auf einem Großrechner läuft, gespeichert sein. Ob die Persistenz von der Bean selbst oder dem Container realisiert wird, ist dem Entwickler der Bean überlassen. Wenn der Entwickler sich entscheidet, die Persistenz im Rahmen einer Transaktion selbst sicherzustellen, dann bedient er sich direkter Aufrufe auf die Datenquelle, wie z.B. JDBC für den Zugriff auf relationale Datenbanksysteme. In diesem Fall übernimmt die Entity Bean genauso wie eine Session Bean die Transaktionssteuerung selbst. Dies hat den Vorteil, daß die Bean quasi in jedem Container lauffähig ist, unabhängig von der Unterstützung von Transaktionen durch den Container. Allerdings ist dann die Bean auf die darunterliegende Datenquelle spezialisiert und kann möglicherweise nicht für andere Datenquellen benutzt werden.

Wird die Verwaltung der Persistenz dem Container überlassen, muß im Deployment Descriptor das Property `containerManagedFields` mit den persistent zu haltenden Feldern definiert sein. Dann kann nämlich ein Tool, das der Container-Provider zur Verfügung stellen muß, das Mapping der Felder zu einer persistenten Datenquelle umsetzten. Das Problem dabei ist, daß das Tool abhängig von der Datenquelle sein muß und einen relativ großen Entwicklungsaufwand erfordert. Da die Beschreibung des Protokolls zwischen Container und Entity Bean an dieser Stelle zu weit führen würde sei auf [Matena et al., 1998, S.75ff] verwiesen.

4.3.3.4 Weitere Aspekte

Die Implementierung des EJB-Containers muß ferner verhindern, daß eine Instanz einer Enterprise Bean von mehreren Clients benutzt wird, oder sie muß entsprechende Regeln, die einen reibungslosen Ablauf garantieren, umsetzen. Letzteres ist jedoch nur bei den Entity Beans möglich und nicht zu empfehlen, wenn die beiden Anfragen an eine Entity Bean aus

dem gleichen Transaktionskontext stammen. Gegen einen konkurrierenden Zugriff auf Entity Beans von verschiedenen Transaktionen ist jedoch nichts einzuwenden. Der Container muß entsprechende Synchronisationskonzepte (entsprechend der Concurrency-Control bei DBMS) zur Verfügung stellen.

Da der verfügbare Platz im Hauptspeicher beschränkt ist, kann der Container Enterprise Beans auf einen sekundären Speicher, gewöhnlich eine Festplatte, auslagern und von dort auch bei Bedarf wieder laden. Dieser Vorgang wird Passivation bzw. Activation genannt und geschieht durch Serialisierung der entsprechenden Beans. Bevor eine Bean ausgelagert wird, wird in ihr die Methode `ejbPassivate` aufgerufen. Wenn die Methode abgearbeitet ist, muß sich das Objekt in einem Zustand befinden, in dem es serialisiert werden kann. Außerdem müssen alle Objektreferenzen der Bean mit abgespeichert werden. Bei der Activation muß zunächst das Enterprise Bean Objekt wieder deserialisiert und dann die Objektreferenzen wiederhergestellt werden, bevor dann `ejbActivate` aufgerufen wird. Zustandslose Session Beans werden nicht ausgelagert, sondern zerstört. Werden sie wieder benötigt, können sie einfach wieder instanziiert werden.

4.3.4 Die Enterprise Beans aus Sicht des Providers

Die Enterprise Beans stellen die eigentlichen Serverkomponenten im Enterprise JavaBeans-Konzept dar. Sie sind spezialisierte JavaBeans, die nicht zur Darstellung am Bildschirm gedacht sind, sich ansonsten jedoch genauso wie normale JavaBeans verhalten. Sie lassen sich mit anderen JavaBeans zusammenschalten um so neue Anwendungen zu erzeugen. Manipulation und Anpassung der Funktionalität von EBs erfolgt über deren Eigenschafts-Tabelle (Property Table) und über spezielle Methoden, die sogenannten Customization Methods.

Die EBs implementieren die eigentliche Business Logik im EJB-Komponentenmodell. Zu einem für Geschäftsanwendungen geeigneten Komponentenmodell gehört ebenfalls die Unterstützung von Transaktionen. Hierbei sieht das EJB-Komponentenmodell von vornherein vor, daß EBs keine explizite Transaktionsverwaltung vorzunehmen haben, d.h. im Programmcode der EBs müssen keine Aufrufe zur Transaktionsverwaltung enthalten sein. Die Propagierung des Transaktionskontextes, in dem ein Aufruf einer Methode einer EB steht, erfolgt also implizit. Die Transaktionsverwaltung selbst wird durch den umgebenden Container vorgenommen.

Ein Client greift niemals direkt auf ein EB-Objekt zu, sondern verwendet dazu das, ihm vom Container zugewiesene, `EJBObject`, d.h. ein Objekt, das das `EJBObject`-Interface implementiert.

Zur Implementierung einer EB ist es notwendig bestimmte Klassen bzw. Interfaces zur Verfügung zu stellen oder zu implementieren. Die wichtigsten dieser Interfaces und Klassen werden im folgenden vorgestellt.

4.3.4.1 Das Home Interface

Das Home Interface einer Enterprise Bean erweitert das `EJBHome`-Interface. Jede EB muß ein solches Home Interface zur Verfügung stellen. Das Home Interface definiert Methoden, die es einem Client erlauben, ein `EJBObject` (vgl. Kap. 4.3.4.2) zu erzeugen (`create`), zu löschen (`remove`) und, im Falle des Zugriffs auf eine Entity Bean (vgl. Kap. 4.3.4.5), ein oder mehrere Entity Bean-Objekte respektive ein oder mehrere Entities einer Datenbank über `find`-Methoden zu suchen. Außerdem kann der Client über dieses Interface bestimmte Meta-Daten über die EB abrufen. Bezüglich Implementierungsdetails vgl. hierzu Kapitel 4.3.3.1.

4.3.4.2 Das Remote Interface

Ein Client greift ausschließlich über das Remote Interface einer Enterprise Bean respektive über ein Objekt, das dieses implementiert, auf die EB zu und niemals direkt. Das Remote-Interface einer EB erweitert das `EJBObject`-Interface. Über das Remote Interface ist es einem Client möglich, die Methoden, die die Business Logik einer EB implementieren, aufzurufen. Dazu kommen die Methoden aus dem `EJBObject`-Interface. Diese Methoden ermöglichen es dem Client den Container des `EJBObjects` oder ein Handle für das `EJBObject` zu ermitteln, auf Gleichheit zwischen zwei `EJBObjects` zu testen, ein `EJBObject` zu löschen und, im Falle eines Zugriffs auf ein Entity Bean-Objekt (vgl. Kap. 4.3.4.5), den Primärschlüssel des Entity-Objekts abzufragen. Für Details bezüglich der Implementierung siehe Kapitel 4.3.3.2.

4.3.4.3 Enterprise Bean-Klasse

Jede Enterprise Bean muß das `EnterpriseBean`-Interface implementieren. Dies geschieht i.d.R. durch Implementierung eines der beiden Interfaces `EntityBean` oder `SessionBean` (vgl. Kap. 4.3.4.5), die beide das `EnterpriseBean`-Interface erweitern.

Zu jeder der create-Methoden des Home Interfaces der EB muß die EB-Klasse eine Methode ejbCreate mit der gleichen Signatur implementieren. Beim Aufruf einer der create-Methoden des Home Interfaces leitet dann das Objekt, das der Container als Implementierung des Home Interfaces bereitstellt, den Aufruf an die entsprechende ejbCreate-Methode der EB-Klasse weiter. Des weiteren enthält die EB-Klasse die Implementierungen der Methoden, die für den Typ der Enterprise Bean (Session oder Entity) notwendig sind. Außerdem implementiert die EB-Klasse alle Methoden, die im Remote Interface der EB definiert sind, d.h. die Methoden, die die Business Logik der Bean implementieren. Dabei müssen selbstverständlich alle Methodennamen und –signaturen übereinstimmen. Im Falle einer Entity-Bean implementiert diese Klasse außerdem die find-Methoden, die dem Auffinden einer oder mehrerer Entities dienen. Diese werden in der EB-Klasse, analog zu den create-Methoden, ejbFind benannt.

4.3.4.4 Deployment Descriptor

Jede EB wird mit einem Deployment Descriptor ausgeliefert. Der Deployment Descriptor ist, je nach Typ der EB (Session oder Entity, vgl. Kap. 4.3.4.5), eine serialisierte Instanz der Klasse deployment.EntityDescriptor bzw. der Klasse deployment.-SessionDescriptor. Der Deployment Descriptor dient zur Ermittlung von vielfältigen Eigenschaften der EJB.

4.3.4.5 Session Beans vs. Entity Beans

Es existieren zwei Arten von Enterprise Beans. Sogenannte Session Beans und sogenannte Entity Beans.

Session Bean-Objekte, d.h. Objekte die das SessionBean-Interface implementieren, werden i.d.R. auf Anforderung eines Clients erzeugt und existieren in den meisten Fällen nur für die Dauer einer einzelnen Client/Server-Sitzung. Session Bean-Objekte führen Aufgaben im Auftrag des Clients durch, wie z.B. den Zugriff auf eine Datenbank oder die Durchführung einer Berechnung. Session Bean-Objekte können Transaktionen unterstützen, sie nehmen aber nach dem Absturz des Servers nicht am Recovery des Servers teil, sondern gehen verloren. Session Bean-Objekte können zustandslos sein oder aber einen internen Zustand über Methodenaufrufe und sogar über Transaktionen hinweg speichern. Zustandslose Session Bean-Objekte können selbstverständlich von mehreren Clients genutzt werden, da es keine

Abhängigkeiten zwischen Methodenaufrufen gibt. Muß ein Session Bean-Objekt aus dem Speicher ausgelagert werden, dann verwaltet der EJB-Container den Kommunikationsstatus des Session Bean-Objekts. Persistente Daten bzw. seinen eventuell intern vorhandenen Zustand muß ein Session Bean-Objekt selbst verwalten. Der EJB-Container ruft die Methoden des `SessionBean`-Interfaces auf, um die Session Bean-Objekte über Ereignisse in ihrem Life-Cycle zu informieren. Diese Ereignisse können u.a. sein, daß das Session Bean-Objekt in den Sekundärspeicher ausgelagert oder aus diesem eingelagert wird oder daß das Session Bean-Objekt gelöscht werden wird.

Entity Bean-Objekte, d.h. Objekte die das Interface `EntityBean` implementieren, repräsentieren persistente Daten, die in einer Datenbank gespeichert sind. Jedes Entity Bean-Objekt wird, im Gegensatz zu Session Bean-Objekten, über einen Primärschlüssel identifiziert und kann über diesen auch jederzeit wiedergefunden werden. Entity Bean-Objekte können, entweder durch Aufruf der „`create`“-Methode einer Object-Factory oder indem Daten direkt in eine Datenbank geschrieben werden, erzeugt werden. Eine Object-Factory ist ein Objekt, das dazu dient, Instanzen eines bestimmten Objektes zu erzeugen. In der Regel hält ein Container eine bestimmte Anzahl von Entity Bean-Objekten eines Typs in einem Objekt-Pool vor und ordnet eines dieser „untätigen“ Objekte, bei Anforderung durch einen Client, diesem zu. Entity Bean-Objekte unterstützen Transaktionen und sind in das Recovery nach einem Servercrash eingebunden. Entity Bean-Objekte besitzen einen internen Zustand und sind außerdem implizit persistent, d.h. sie können, müssen sich jedoch nicht um die Persistenz ihrer Daten bzw. ihres internen Zustandes kümmern. Wird Persistenz nicht von einem Entity Bean-Objekt selbst verwaltet, so wird diese Aufgabe an den EJB-Container delegiert. Im Unterschied zu Session Bean-Objekten können Entity Bean-Objekte i.d.R. von mehreren Clients genutzt werden. Die Methoden des `EntityBean`-Interfaces werden vom EJB-Container aufgerufen, um das Entity Bean-Objekt von bestimmten Ereignissen im Rahmen seines Life-Cycles zu informieren. Diese Methoden umfassen Ereignisse, die sich auf die Zuordnung von Clients zu Entity Bean-Objekten, das Speichern bzw. Laden des Zustandes des Entity Bean-Objektes in bzw. aus der Datenbank oder das Auflösen der Verbindung zwischen einem Client und dem Entity Bean-Objekt beziehen.

Die Unterstützung von Entity Bean-Objekten durch den Container ist in der Version 1.0 der EJB-Spezifikation optional, es ist jedoch vorgesehen, in der Version 2.0 die Unterstützung von Entity-Objekte verpflichtend vorzuschreiben.

4.3.5 Resümee

Die Ziele von JavaBeans, eine Standardkomponentenarchitektur zur Erzeugung und Ausführung von Business Objekten, die Abstraktion von Low-Level APIs, wie denen zur Transaktionssteuerung, sowie die Voraussetzungen zu schaffen, damit Tools und Komponenten verschiedenster Hersteller miteinander reibungslos zusammenarbeiten, sind hoch gesteckt.

Dennoch ist es Sun Microsystems gelungen, diese Ziele weitestgehend zu erreichen. Enterprise JavaBeans ist ein Konzept, das gut durchdacht ist und mit Sicherheit den Wünschen vieler Entwickler entgegenkommt. Die Möglichkeit, Gruppen mit unterschiedlichen und weitgehend voneinander getrennten Kompetenzen zu bilden, ist ein wichtiger Aspekt in diesem Konzept. Dadurch können sich Anbieter einzelner Bausteine aus der EJB-Architektur auf jene konzentrieren, bei denen ihre Stärken liegen. Eine Problematik, die bei der Verwendung von großen, selbständig arbeitenden Komponenten auftreten kann, ist in der EJB-Architektur jedoch noch nicht berücksichtigt, die Synchronisation der Komponenten untereinander, d.h. die Synchronisation von parallel ablaufenden Tätigkeiten in verschiedenen Enterprise Beans. Diese Problematik ist im Moment ausschließlich von den Enterprise Bean-Providern und den Application-Assemblern, z.B. durch entsprechende Event-Protokolle zu lösen. Es wäre wünschenswert, wenn Sun an dieser Stelle mit neuen Konzepten die EJB-Architektur erweitern würde. Man muß jedoch erwähnen, daß diese Problematik in Komponentenkonzepten grundlegend ist und auch in der Forschung bis heute nicht gelöst wurde.

Eine Reihe namhafter Firmen, darunter IBM, Oracle und Sybase, um nur einige zu nennen, haben ihre Unterstützung zugesagt. Tatsächliche Implementierungen lassen aber zur Zeit noch auf sich warten. Das mag zum einen daran liegen, daß die EJB-Spezifikation noch sehr neu ist, zum anderen ist die Entwicklung gerade von Servern und Containern mit Sicherheit nicht trivial, da dort die meiste Low-Level-Funktionalität des Konzepts verborgen ist, denn schließlich wird insbesondere die einfache Entwicklung der Enterprise Beans selbst forciert.

Dennoch muß man auch die allgemeinen Probleme der Sprache Java im Blick behalten, denn EJB ist ja lediglich eine Erweiterung. Deshalb wird auch der Erfolg von EJB maßgeblich von dem weiteren Erfolg von Java abhängen. Trotzdem kann man sagen, daß EJB ein erfolgversprechendes Konzept ist und es wünschenswert wäre, wenn die Theorie nun in die Praxis umgesetzt würde.

5 Java und JavaBeans in einer heterogenen Umwelt

In der heutigen Welt mit ihren heterogenen, verteilten Umgebungen und ihren vorhandenen Legacy-Systemen will Java nicht den vollständigen Ersatz dieser Systeme erreichen. Jedoch kann Java zum einen als ein Verbindungsglied und zum anderen als ein kleinster gemeinsamer Nenner verwendet werden, um Altsysteme um neue Funktionalität und speziell um eine Anbindung ans WWW zu erweitern.

Java sieht daher seit jeher Möglichkeiten vor, in verteilten, heterogenen Systemen zu existieren und sie zu nutzen. Die heterogenen Systeme sind für Java unproblematisch, sofern es auf jedem zu nutzenden System eine Java-Implementierung gibt. Diese bildet als Zwischenschicht zwischen dem Basissystem und der Java-basierten Anwendung eine Möglichkeit, um von der verwendeten Hardware zu abstrahieren und so die Plattformunabhängigkeit zu gewährleisten. Existiert keine Java-Umgebung bzw. ist eine Anbindung an diese nicht möglich, so kann man auf diesen Plattformen plattformspezifischen Code verwenden und auf diesen von Java aus zugreifen.

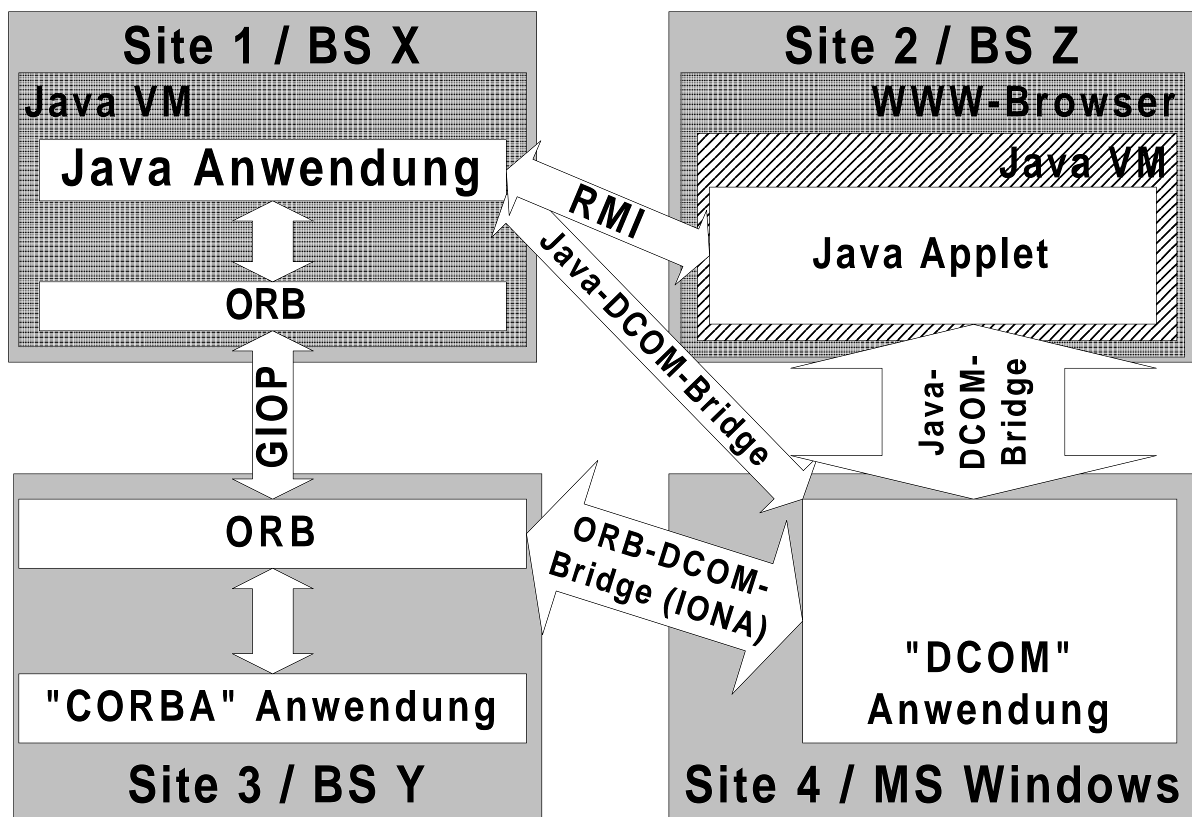

Abbildung 13 Java in einer heterogenen Welt

Den Anforderungen einer verteilten Umgebung genügt Java zum heutigen Zeitpunkt auf drei Arten. Die erste Art betrifft die Kommunikation in Java-basierten, verteilten Umgebungen. Hier bietet Java sein eigenes Kommunikationskonzept: RMI – Remote Method Invocation. RMI entspricht der Java-Version des Remote Procedure Calls, der heutzutage in nahezu allen Betriebssystemen zu finden ist. Er ermöglicht die Kommunikation zwischen verteilten Java-„Prozessen".

Der Heterogenitätsaspekt bringt es jedoch mit sich, daß ein Entwickler nicht nur innerhalb der Java-Welt agieren kann. Daher gibt es zwei weitere Kommunikationsarten, die der Kommunikation mit der „Nicht-Java-Welt" dienen. Die erste verwendet den CORBA-Standard der Object Management Group (OMG), um mit Systemen zu interagieren, die die CORBA-Architektur unterstützen. Diese Kommunikationsart wird von Haus aus von Java respektive Sun unterstützt und präferiert. Die zweite Art verwendet DCOM (Distributed Component Object Model), den Microsoft-„Standard" für verteilte Kommunikation. Die Kommunikation mit Systemen die DCOM unterstützen kann zum einen direkt, aber auch über den Umweg über CORBA erfolgen.

Um die Kommunikation mit CORBA zu ermöglichen, ist in Zusammenarbeit von Sun mit der OMG ein Mapping der CORBA Interface Definition Language (IDL) auf Java definiert worden. Die IDL dient der sprachunabhängigen Definition von Typen, Records und Methoden, die innerhalb einer CORBA-Umgebung verwendet werden sollen. Die IDL wird mittels eines sprachabhängigen IDL-Compilers in Programmcode der entsprechenden Zielsprache, wie z.B. C++, C, Smalltalk oder Java, umgewandelt. Der erzeugte Code dient zum Zugriff auf CORBA-Objekte, die entsprechend der IDL-Definition implementiert wurden, unabhängig von der Plattform, auf der das Objekt instanziiert wurde und der Sprache, in der es implementiert wurde.

Nach der Definition des IDL-Java-Language-Mappings entwickelte Sun die entsprechenden Klassen für Java, um eine Anbindung an ein CORBA-System zu ermöglichen. Dies bezieht auch die Implementierung eines Object Request Brokers (ORB), des Kommunikationszentrums einer CORBA-Implementierung mit ein. Somit wird es mit dem Java Development Kit ab der Version 1.2 möglich sein, eine Anbindung von Java an CORBA vorzunehmen. Neben der Referenzimplementierung eines ORB für Java, wie sie Sun bietet, wird es zusätzlich ORB-Implementierungen von Drittherstellern geben. Eine solche Implementierung existiert heute bereits mit OrbixWeb von Iona Technology, einer Java-

Portierung des sehr erfolgreichen Orbix-ORBs des gleichen Herstellers. Mit weiteren ORB-Implementierungen wird in Zukunft zu rechnen sein, sofern der wirtschaftliche Erfolg Javas anhält.

Zusätzlich zur Implementierung der grundlegenden CORBA-Funktionalitäten hat Sun auch begonnen, weitere CORBA-Dienste nach Java zu portieren. So ist der Java Transaction Service eine Portierung des Object Transaction Services, des standardisierten Transaktionsdienstes für CORBA.

Ein besonderer Vorteil der Anbindung von Java und CORBA stellt die Tatsache dar, daß es sowohl in Java, als auch in CORBA ausgeklügelte Sicherheitsmechanismen gibt, so daß hier, bei entsprechender Implementierung, ein relativ umfassendes Schutzkonzept für den Zugriff auf Hardware, Betriebssystem und CORBA-Objekte besteht.

Die Anbindung von Java an das verteilte Komponentenmodell Microsofts, DCOM, ist ein Ziel, daß primär nicht von den Entwicklern Javas anvisiert, von diesen jedoch als möglich beurteilt wird. Diese zurückhaltende Beurteilung dürfte weniger auf prinzipielle technische Gründe zurückzuführen sein, als auf wirtschaftliche und „firmenpolitische“ Erwägungen.

DCOM ist primär ein Modell für verteilte Komponenten innerhalb der Windows-Welt. Zwar gibt es inzwischen Portierungen auf diverse Unix-Systeme, die durch die Zusammenarbeit der Software AG und Microsoft entstanden sind, jedoch ist zum heutigen Zeitpunkt noch nicht abzusehen, inwieweit sich DCOM in der Unix-Welt etablieren können wird.

Implementierungen einer Bridge zwischen Java und DCOM sind somit nicht von den Sun-Entwicklern zu erwarten. Es gibt jedoch bereits erste Implementierungen, die sich noch in einem frühen Entwicklungsstadium befinden, die in Zukunft entsprechende Produkte erwarten lassen.

Wer schon heute auf den Zugriff von Java auf DCOM-Server angewiesen ist, kann dies über einen „Umweg“ erreichen, indem er Iona Technologys ActiveX-Bridge von CORBA nach DCOM verwendet. ActiveX-Komponenten sind spezielle (D)COM-Objekte. Diese Komponenten stellen quasi eine Erweiterung der Standardfunktionalität von (D)COM-Objekten dar, die von Microsoft u.a. für den Einsatz in verteilten Windows-Systemen entwickelt wurde. Will man also zum momentanen Zeitpunkt von Java auf DCOM zugreifen, so muß man den Umweg über CORBA wählen. Die Einbettung von JavaBeans in OLE-,

COM- und ActiveX-Komponenten kann schon heute über die ‚JavaBeans Bridge for ActiveX' von Sun erreicht werden.

Ein Problem, das sich allerdings aus der Anbindung von Java an DCOM ergibt, ist ein Problem, daß allen auf DCOM basierenden Objekte innewohnt. Auf den Rechnern, auf denen ein DCOM-Objekt ausgeführt wird, hat es vollen Zugriff. Es existiert kein weiterführendes Sicherheitskonzept, daß beispielsweise in genormter Art und Weise den Zugriff auf solche Objekte regeln würde.

Welche Brücke aus der „Sandkastenwelt" Javas in die reale Welt von hardwareabhängigen Programmen sich durchsetzen wird, kann zum heutigen Zeitpunkt nicht beurteilt werden. Aufgrund der Tatsache, daß sich Microsoft dem Einsatz von Java mit aller Macht zu entziehen sucht und keine vollständige Java-Implementierung in absehbarer Zeit von Microsoft zu erwarten sein wird, ist es jedoch wahrscheinlich, daß sich die Anbindung von Java an die Welt der bestehenden Systeme über CORBA etablieren wird. Da dieser Weg nicht unter der Kontrolle von Microsoft steht und somit eine Verwendung desselben von Microsoft kaum verhindert werden kann, dürfe dies eine relativ zukunftssichere Variante zur Kopplung von Java- und Legacy-Systemen darstellen.

6 JavaBeans und Enterprise Beans als Komponenten

Für JavaBeans und Enterprise Beans stellt sich die Frage, wie sie in die Komponentendefinition des zweiten Beitrags dieser Vortragsreihe einzugliedern sind. Dieser Beitrag bietet eine Art Checkliste anhand welcher diese Frage überprüft werden kann. Da Enterprise Beans eine Erweiterung und Verfeinerung der JavaBeans darstellen, gelten die unten aufgeführten Punkte somit auch für sie.

- Komponenten werden zusammengesetzt, um ein Informationssystem zu bilden.

  Gerade hierfür sind JavaBeans mit ihrer Zusammenarbeit mit Entwicklungswerkzeugen konzipiert.

- Sie sind austauschbar, konfigurierbar und erweiterbar.

  Die Austauschbarkeit ist gegeben und wird durch das InfoBus Konzept weiter verstärkt. Die Konfigurierbarkeit wird durch Property Sheets und Customizer erreicht, die Erweiterbarkeit ist durch das Vererbungskonzept gegeben.

- Die Bindung zwischen den Komponenten sollte minimal sein.
  Die Bindung zwischen Beans erfolgt lediglich über Methodenaufrufe des öffentliche Interfaces und das Eventhandling. Das InfoBus Konzept verkleinert diese Bindung nochmals, da fremde Beans nicht bekannt sein müssen.
- Komponenten bilden eine Abstraktion der realen Welt.
  Jede Klasse einer objektorientierten Sprache bildet eine Abstraktion der realen Welt, womit auch JavaBeans als Erweiterungen der Java Klassen diese abbilden.
- Komponenten haben eine signifikante Komplexität.
  Die Komplexität der JavaBeans ist durch ihre Funktionalität gegeben, die einem bestimmten Anwendungsfall entspricht.
- Komponenten sind in sich abgeschlossen.
  JavaBeans können nur über ihr öffentliches Interface angesprochen werden. Eine Bean enthält jegliche für sie wichtige Funktionalität.

Darüber hinaus lassen sich JavaBeans in die Komponentenklasse der Anwendungselemente einstufen, da JavaBeans im Bytecode vorliegen und somit auf einer JVM ausführbar sind. Enterprise Beans lassen sich in eine Unterklasse der Anwendungselemente, die Business Objects, einordnen. Nach dem Selbstverständnis von Sun Microsystems sind sie gerade für solche Objekte konzipiert. Es ist jedoch möglich auch Objekte, die keine betriebswirtschaftlichen Methoden enthalten, mit ihrer Hilfe zu implementieren.

7 Resümee

7.1 Die Zukunft von Java

Java verfolgt primär drei Ziele:

- Plattformunabhängigkeit
- WWW-Unterstützung
- Sicherheit

Durch die Plattformunabhängigkeit soll erreicht werden, daß Code nur einmal geschrieben werden muß und dann auf vielen Plattformen ablauffähig ist, was die Entwicklungskosten von Software erheblich senken würde und einer Softwarekomponente eine viel größere Basis zur

Verfügung stellen würde, auf der sie genutzt und somit auch verkauft werden könnte. Außerdem ist das Ziel der Plattformunabhängigkeit sehr stark mit dem zweiten Ziel, der WWW-Unterstützung verbunden. Wenn Anwendungen über das WWW auf einen Rechner geladen und dann auf diesem Rechner ausgeführt werden sollen, ist es nicht, bzw. kaum möglich, diese Anwendung in unzähligen plattformspezifischen Varianten zur Verfügung zu stellen. Somit bleibt nur die Möglichkeit, eine gemeinsame Ausführungsplattform zu definieren, auf der dann ein und dieselbe Anwendung ablaufen kann. Diese Plattform muß selbstverständlich systemspezifisch sein, jedoch für die Anwendung eine systemunabhängige Ausführungsumgebung zur Verfügung stellen. Die Tatsache, daß „fremde“ Applikationen über das Internet auf den eigenen Rechner übertragen und dort ausgeführt werden, führt direkt zum dritten Ziel, der Sicherheit. Es darf einer solchen Anwendung nicht möglich sein, auf dem Rechner, auf dem sie ausgeführt wird Schaden anzurichten. Genau dies bezweckt Java mit seinen Sicherheitskonzepten.

Bedauerlicherweise gibt es nach wie vor eine Reihe von Problemen in Bezug auf den Einsatz von Java in großen Projekten. Das erste Problem liegt in der **Performance**. Java-Anwendungen zeichnen sich bisher primär durch eine sehr geringe Ausführungsgeschwindigkeit aus. Es wird zwar ständig versucht die Performance zu steigern, indem man z.B. Just-in-Time-Compiler einsetzt, die auf der ausführenden Maschine den Java-Bytecode in maschinenabängigen Code übersetzen und diesen dann mit höherer Geschwindigkeit direkt durch den Prozessor des Rechners ausführen lassen. Dies ist eine relativ gute Zwischenlösung, die jedoch ebenfalls nicht unproblematisch ist, da jede Java-Klasse vor der Ausführung compiliert werden muß, was bei größeren Systemen ebenfalls eine nicht unerhebliche Zeit in Anspruch nimmt. Ende 1998 soll von Sun eine neue Version der Java Virtual Machine unter dem Codenamen „HotSpot“ auf den Markt kommen, von der Sun behauptet, daß sie die Ausführungsgeschwindigkeit von Java auf das Niveau von compiliertem C++-Code anheben soll. Es bleibt abzuwarten, ob Sun diese hehren Zielen erreichen kann. Eine weitere Möglichkeit, die Ausführungsgeschwindigkeit von Java zu erhöhen, bestände im Einsatz spezieller Java-Prozessoren. Bisher hat Sun jedoch jede Art von Benchmarking der Prototypen dieser Prozessoren untersagt, was Zweifel begründet, ob die Performance durch diese Prozessoren tatsächlich deutlich ansteigt. Ein Problem bei der Entwicklung performanter Java-Prozessoren könnte sein, daß die einfache Struktur der Java Virtual Machine, eine Stackmaschine, gerade die Optimierung der Ausführung in einem

Prozessor erschwert, denn Konzepte wie Parallelausführung von Anweisungen oder auch Pipelineing sollen sich für Stackmaschinen nur schwer bzw. gar nicht implementieren lassen.

Das nächste Problem liegt in der geringen Stabilität von Java-Anwendungen. Bei bestehenden, großen Java-Anwendungen ist bedauerlicherweise zu beobachten, daß sie, neben der geringen Performance, nicht stabil laufen. Teilweise treten sogar bei aufeinanderfolgenden Aufrufen der Anwendungen verschiedene Fehler auf. Auch lassen sich die Anwendungen nicht problemlos von einer Plattform auf eine andere übernehmen. Als Beispiele seien hierbei Suns WWW-Browser „HotJava 1.1“ und Suns Java-Entwicklungswerkzeug „Java-Workshop 2.0a“ genannt. Beiden Anwendungen kann man leider nur eine geringe Stabilität und Robustheit bescheinigen. Die Ursachen für diese Probleme liegen gerade in der Abstraktionsschicht, die Java über der Hardware aufbaut. Denn diese Schicht, die an vielen Stellen von der darunterliegenden Hardware und dem verwendeten Betriebssystem abhängt, weist auf verschiedenen Plattformen immer wieder Implementierungsunterschiede und –fehler auf. Da diese Fehler aber von Implementierung zu Implementierung unterschiedlich sind, erhält man leider auf fast allen Plattformen bei komplexen Anwendungen ein unterschiedliches Verhalten. Besonders leidet bisher die Bildschirmdarstellung unter diesen Unterschieden, da es scheinbar nicht gelingt, die verschiedenen Benutzeroberflächen so zu nutzen, daß das Erscheinungsbild einer Java-Anwendung auf jedem System identisch ist. Zumindest dieses Problem sollte mit Einführung der Java Foundation Classes, auch Swing genannt, in den Hintergrund treten, da nun die Darstellung der Anzeigeelemente, wie z.B. ein Textfeld oder ein Button, fast vollständig von Java aus realisiert wird und das darunterliegende Betriebssystem nur noch zum Zeichnen von Linien und Punkten genutzt wird.

Aufgrund der beschriebenen Probleme und der Tatsache, daß zwar viele namhafte Unternehmen Java unterstützen, jedoch für die Entwicklung von Großanwendungen bisher nicht erfolgreich eingesetzt haben, muß Java erst noch beweisen, daß es auch zur Implementierung großer Anwendungen geeignet ist.

Das nächste Problem ergibt sich aus der Tatsache, daß eine breite Kluft zwischen der Versionsnummer des Java-Standards und der Implementierung der JVMs in den gängigen WWW-Browsern entstanden ist. Netscapes Communicator unterstützt nach wie vor nur Java 1.0.2, Suns HotJava unterstützt zwar den aktuellen Java-Standard, ist jedoch kein WWW-Browser für ernsthafte Anwendungen und ausgerechnet Microsofts Internet Explorer 4

unterstützt immerhin große Teile des Java 1.1-Standards und das, obwohl Microsoft, aus firmenpolitischen Erwägungen heraus, eine ablehnende Haltung zu Java eingenommen hat. Jedoch plant Microsoft zur Zeit nicht, jemals den vollen Sprachumfang Javas zu unterstützen. Davon abgesehen weist der Internet Explorer, im Rahmen seiner Java-Implementierung, auch noch die scheinbar stabilste Implementierung Javas auf. Dieses traurige Bild ist um so bedauerlicher, als Java 1.1 seit dem 18. Februar 1997 veröffentlicht ist und Java 1.2 im Sommer 1998 erwartet wird. Nur noch pro forma soll angemerkt werden, daß selbst Suns Java-Entwicklungsumgebung „Java-Workshop 2.0a“ Java 1.1, insbesondere das dort eingeführte Event-Konzept, nicht vollständig unterstützt.

Ein weiteres Problem, das bei der Softwareentwicklung unter Java immer wieder auftritt, ist das Fehlen der Mehrfachvererbung im Java Sprachstandard. Zwar ist dies gewollt, da sowohl der Laufzeit-Overhead als auch die Compiler-Komplexität bei Verwendung von Mehrfachvererbung deutlich steigt, jedoch führt dies dazu, daß man häufig ein und denselben Code mehrfach programmieren muß, nur weil immer wieder das gleiche Interface implementiert werden muß. Dies würde bei Verwendung von Mehrfachvererbung nicht der Fall sein. Heute wird dieses Dilemma meist dadurch umgangen, indem man die Methode der Delegation verwendet. D.h. man hat eine „Hilfsklasse“, die die „zu erbenden“ Methoden enthält und schreibt für diese Methoden bei der Implementierung des dazugehörenden Interfaces nur noch Wrapper, die die Methoden der Hilfsklasse aufrufen. Dieses Vorgehen entspricht „Mehrfachvererbung von Hand“. Es ist jedoch inzwischen bewiesen, daß die Konzepte Mehrfachvererbung und Delegation gleichmächtig sind.

7.2 Die Zukunft von JavaBeans

Mit der JavaBeans-Architektur verfolgt Sun das Ziel eine Komponentenarchitektur für Java zu definieren. Diese Architektur basiert auf einer möglichst einfachen API, um so die Erstellung von Komponenten einfach zu gestalten. JavaBeans ist zur Entwicklung von Komponenten kleiner und mittlerer Granularität gedacht und zwar speziell für solche, die für die Bildschirmanzeige zuständig sind. Außerdem sind von vornherein Entwicklungswerkzeuge zur Applikationserstellung in das Konzept einbezogen.

Die Hauptprobleme des JavaBeans-Konzepts liegen zum einen in der Tatsache, daß durch die sehr einfach gehaltene JavaBeans-APIs das Konzept primär aus Designpatterns besteht und somit mit der sklavischen Befolgung dieser Vorgaben steht und fällt. Zum anderen gibt es

noch Probleme mit den Entwicklungswerkzeugen, denn viele dieser Werkzeuge sind, abgesehen von der fehlenden Unterstützung von Java 1.1, nicht ausgereift. Teilweise arbeiten sie nicht zuverlässig und teilweise ist die Bedienung nicht ausgereift. Zum wiederholten Mal sei hier speziell auf Suns eigene Entwicklungumgebung „Java-Workshop“ verwiesen. Selbst dieses, für Sun eigentlich strategische Produkt, bietet weder volle Java 1.1 Unterstützung, noch die zu wünschende Stabilität und Ergonomie.

Es gibt inzwischen eine erkleckliche Menge an freien und kommerziellen JavaBeans, sowohl von den großen Softwareherstellern wie IBM oder Sun, als auch von vielen kleinen Softwarehäusern. Man kann sich leicht durch Inspizieren eines der folgenden Softwareverzeichnisse, die ausschließlich oder teilweise JavaBeans enthalten, davon überzeugen:

- JavaBeans Directory
  (http://jsg.cuesta.com)
- IBM-Verzeichnis für JavaBeans
  (http://www.software.ibm.com/ad/vajava/vatools1.htm)
- Netscape-Verzeichnis für Softwarekomponenten
  (http://developer.netscape.com/software/components/listing/index.html)

Festzustellen ist, daß eine große Anzahl renommierter, bekannter und erfolgreicher Softwarehersteller Javas Komponentenkonzept unterstützt. Es bleibt jedoch abzuwarten, ob tatsächlich alle Firmen, die ihre Unterstützung angekündigt haben, auch professionelle Produkte auf den Markt bringen, oder ob sie, wegen der Probleme, die Java heute noch hat, erst einmal abwarten, ob sich Java etablieren kann oder nicht.

7.3 Enterprise JavaBeans

Die Enterprise JavaBeans-Architektur als Erweiterung des JavaBeans-Konzepts ist als Standardkomponentenarchitektur zur Erzeugung und Ausführung von Business-Objekten konzipiert. Dabei wird der Entwickler der Enterprise Beans, der die eigentliche Business-Logik erstellen soll, von systemnahen Konzepten, wie Persistenz-, Ressourcen- oder Transaktionsmanagement ferngehalten. Er kann und soll sich ausschließlich um die eigentliche Funktionalität „seiner“ Komponenten kümmern müssen.

Über die Definition der Enterprise JavaBeans-Architektur soll sichergestellt werden, daß Tools, systemnahe Komponenten, wie z.B. der EJB-Server, und Anwendungskomponenten reibungslos zusammenarbeiten.

Die Probleme der EJB-Architektur liegen zum einen in der mangelnden Spezifikation einzelner Elemente eines EJB-Systems, ihrer Kompetenzen und Aufgaben, zum anderen darin, daß die Architektur selbst eine hohe Komplexität in Bezug auf die Implementierung des EJB-Servers und des EJB-Containers aufweist. Der Erfolg der Enterprise JavaBeans bleibt abzuwarten, da im Moment zwar wiederum sehr viele Unternehmen ihre Unterstützung zugesagt haben, es sich hierbei jedoch in den meisten Fällen lediglich um Absichtserklärungen handelt. EJB-Produkte sind bislang nicht auf dem Markt verfügbar.

8 Ausblick

Die neuen Versionen, sowohl des Java-Sprachstandards, der diesen Sommer in Version 1.2 erscheinen soll und dann volle CORBA-Unterstützung beinhalten wird, als auch der Enterprise JavaBeans-Spezifikation, deren Erscheinen in der Version 2.0 angedeutet ist und die dann die Schwachstellen der aktuellen Spezifikation beseitigen oder zumindest abschwächen soll, lassen auf weitere Verbesserungen im Bereich Javas, seiner Anbindung an verteilte Komponentenarchitekturen wie CORBA und JavaBeans hoffen.

Inwieweit Java in Zukunft aber für mehr als zur Implementierung portabler Frontends für große Softwaresysteme eingesetzt werden wird, hängt primär davon ab, ob Sun oder andere Hersteller die vorhandenen Probleme mit Java in kurzer Zeit beseitigen kann. Gelingt es Sun die Entwicklung von stabilen, performanten, webintegrierten Anwendungen zu ermöglichen, so könnte Java auch für größere Softwareprojekte, die dann unabhängig von spezifischen Hardwareplattformen lauffähig wären, interessant werden und in Verbindung mit der, per se, gut konzipierten Komponentenarchitektur und ihren Erweiterungen ein kommerzieller Erfolg auf breiter Front werden.

9 Code-Beispiel

Dieses Code-Beispiel zeigt das JavaBean „Multiplicator“ und seine Anbindung in Suns Beanbox (ein einfaches beispielhaftes Buildertool von Sun).

JavaBean `Multiplicator.java`:

```
import java.io.Serializable;

public class Multiplicator implements Serializable {

 private int x,y,result;

 public Multiplicator() {
    super();
    x = 0;
    y = 0;
    result = 0;
 }

 public void setX(int anX) {
  x = anX;
 }


 public int getX() {
  return x;
 }

 public void setY(int anY) {
    y = anY;
 }


 public int getY() {
    return y;
 }

 public int getResult() {
  return result;
 }

      public void multiplicate() {
  result = x * y;
 }

}
```

Diese Bean wird bei Verwendung von Suns Java Development Kit auf folgende Weise in Bytecode übersetzt:

```
javac Multiplicator.java
```

Um die Bean für ein Entwicklungswerkzeug wie die Beanbox zugänglich zu machen, muß sie zusammen mit einem sogenannten Manifest-File in ein Jar-File gepackt werden.

Manifest-File `myManifest.mt`:

```
Name: Multiplicator.class
Java-Bean: True
```

Das Jar-File `multiplicator.jar` wird auf folgende Weise erstellt:

```
jar cfm multiplicator.jar myManifest.mt Multiplicator.class
```

Dieses Jar-File kann nun entweder in das Jar-Verzeichnis des Entwicklungswerkzeugs (in diesem Fall der Beanbox) gestellt oder manuell geladen werden. Wie einzelne Entwicklungswerkzeuge mit dem Laden der Jar-Files umgehen ist nicht spezifiziert und vom jeweiligen Werkzeug abhängig.

Die folgenden Abbildungen zeigen das Property Sheet der Multiplicator Bean und die Verbindung eines Buttons über das `actionPerformed`-Event mit der `multiplicate`-Methode der Bean.

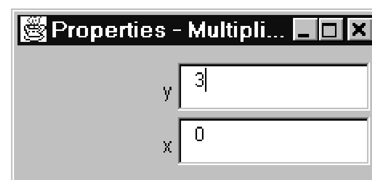

Abbildung 14 Property Sheet der Multiplicator Bean

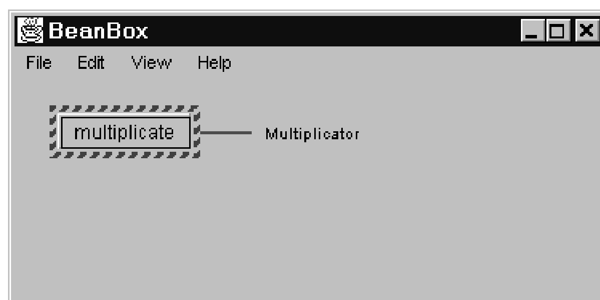

Abbildung 15 Kombination mit einer Button Bean

Literaturverzeichnis

Borenstein et al., 1993	N. Borenstein, N. Freed – MIME (Multipurpose Internet Mail Extension), Network Working Group, RFC 1521, September 1993
Borenstein, 1993	N. Borenstein – A User Agent Configuration Mechanism for Multimedia Mail Format Information, Network Working Group, RFC 1524, September 1993
Calder et al., 1998	Bart Calder, Bill Shannon – JavaBeans™ Activation Framework Specification Version 1.0, Sun Microsystems, Palo Alto, Kalifornien, 16. März 1998
Colan, 1998	Mark Colan (Lotus Development Corp) – InfoBus 1.1 Specification, Sun Microsystems, Palo Alto, Kalifornien, 16. März 1998
DeSoto, 1997	Alden DeSoto – Using the Beans Development Kit 1.0, Sun Microsystems, Palo Alto, Kalifornien, September 1997
Flanagan, 1997	David Flanagan – Java in a Nutshell, 2nd Edition, O'Reilly, Camebridge, 1997
Garg, 1998	Rohit Garg – Enterprise JavaBeans™ to CORBA Mapping Version 1.0, Sun Microsystems, Palo Alto, Kalifornien, 23. März 1998
Hamilton, 1997	Graham Hamilton (Editor) – JavaBeans™ Version 1.01, Sun Microsystems, Palo Alto, Kalifornien, 24. Juli 1997
JNDI, 1998	Sun Microsystems – JNDI: Java™ Naming and Directory Interface, Sun Microsystems, Palo Alto, Kalifornien, 29. Januar 1998
Jubin et al., 1997	Henri Jubin and the Jalapeno Team – JavaBeans by Example, Prentice Hall PTR, New Jersey, 1997
Matena et al., 1998	Vlada Matena, Mark Hapner – Enterprise JavaBeans Version 1.0, Sun Microsystems, Palo Alto, Kalifornien, 21. März 1998
Thomas, 1997	Anne Thomas – Enterprise JavaBeans – Server Component Model for Java, Patricia Seybold Group 1997

Java-Seiten im WWW

Java-Homepage	http://www.javasoft.com/
JavaBeans-Homepage	http://www.javasoft.com/beans/index.html
Enterprise JavaBeans-Homepage	http://java.sun.com/products/ejb/index.html
InfoBus-Homepage	http://java.sun.com/beans/infobus/index.html
Distributed InfoBus	http://www.alphaworks.ibm.com/formula.nsf/system/technologies/4AA174089F154CB18825661200769F64?OpenDocument
JavaBeans Activation Framework-Homepage	http://java.sun.com/beans/glasgow/jaf.html
Java Workshop-Homepage	http://www.sun.com/workshop/java/
Java-Produkte	http://java.sun.com/products/

Stichwortverzeichnis

.

.mailcap-File 23

A

`abstract` 3
Abstract Window Toolkit........ 4
abstrakte Klassen 3
Activation........ 40
ActiveX-Bridge........ 47
Anwendungselemente 49
`Applet` 3, **5**, 10
Application Assembler........ 33
`ArrayAccess` 30
AWT 4

B

BeanInfo Klasse........ **16**
Beispiel 54
Bound Property........ 15
Buildertool 9, 36
Business Objects 49
Bytecode 5
Bytestream 19

C

Class........ 36
Client........ 33
CommandInfo Klasse 24
CommandMap Interface 23
CommandObject Interface........ 24
Commit 38, 39
Component Registry 23
Constrained Property 15
Consumer 26
Controller 26
CORBA........ 34, 46
`create` 42
Customization........ **17**
Customization Methods 40
Customizer........17, **18**

D

DataContentHandler Interface........23
`DataFlavor` 31
DataHandler Klasse........22
`DataItem`........30
`dataItemAvailable`........30
`dataItemRequested`........30
`dataItemRevoked` 30
DataSource Interface........23
Datenkonsumenten........26
Datenproduzenten 26
`DbAccess` 30, 31
DCOM 46, 47
Default Property Editoren........18
Delegation........30, 36
Deployment Descriptor........38, 39, **42**
Deserialisation 21
Deserialisierung 20
Designpatterns 16
Distributed Component Object Model........46

E

EB 32
EJB 32
EJB-Container........33, **34**
EJB-Container Provider........33
`ejbCreate`........42
`ejbFind` 42
EJBHome........34, 41
EJBMetaData........36
EJBObject........34, 41
EJB-Server........33
EJB-Server Provider 33
Enterprise Bean Provider........33
Enterprise Beans 32, 33, 40, **41**, 48

Enterprise JavaBeans **32**, 40
- **Deployment Descriptor** 42
- **Enterprise Beans** 41
- Entity Bean... 43
- **Home Interface** 41
- **Remote Interface** 41
- Session Bean 42

`EnterpriseBean`.. 41
Entity Bean 36, **43**
Entity Enterprise Bean 36
`EntityBean` 41, 43
Entwicklungswerkzeug 16, 17
Ereignis 12
Event 12
Event Listener 12, **13**
Event Listener Interface 12, **13**
Event Object 12, **13**
Event Source 12, **13**
Events
- dataItemAvailable 30
- dataItemRequested 30
- dataItemRevoked... 30
- InfoBus.. 27, 31
- InfoBusItemAvailableEvent........................... 31
- InfoBusItemRequestedEvent.......................... 31
- InfoBusItemRevokedEvent 31

Externalization **20**

F

`find`... 42
`findDataItem`.. 31
`fireItemAvailable`................................... 31
`fireItemRevoked`.. 31

G

Garbage Collection 3
`getCustomizerClass()` 19
`getSource()` 13

H

Handle...37
Home Interface...**41**

I

IDL...46
IDL-Java-Language-Mapping...............................46
`ImmediateAccess` 30
`implements` 3
Indexed Property 14
InfoBus..**25**
`InfoBusDataConsumer`..........................27, 30
`InfoBusDataProducer` 30
InfoBus-Events..31
`InfoBusItemAvailableEvent`..................31
`InfoBusItemRequestedEvent`..................31
`InfoBusItemRevokedEvent` 31
InfoBus-Klasse 29
`InfoBusMember` 26, 29
InfoBusMemberSupport-Klasse.......................29
instanziieren 3
Integration..10
Interception 36
Interface 3
Interface Definition Language 46
Interfaces
- Applet...3
- ArrayAccess...30
- **DataItem** 30
- DbAccess 30, 31
- EJBHome 41
- EJBMetaData 36
- EJBObject 41
- EnterpriseBean...41
- EntityBean...41, 43
- ImmediateAccess 30
- InfoBusDataConsumer...............................27, 30
- **InfoBusDataProducer**...................................30
- InfoBusMember 26, 29

java.applet.Applet ... 6
java.beans.Customizer ... 19
java.beans.PropertyEditor ... 18
RowsetAccess ... 30
ScrollableRowsetAccess ... 30
Serializable ... 20
SessionBean ... 41, 42
SessionSynchronization ... 38
Internationalization ... 8
Introspection ... 15, 25
Introspector ... **16**

J

JAF ... 21
Java Database Connectivity ... 4
Java Development Kit ... 2
Java Foundation Classes ... 4
Java Transaction Service ... 47
Java Virtual Machine ... 5
java.applet.Applet ... 6
`java.beans.Customizer` ... 19
`java.beans.PropertyEditor` ... 18
`java.beans.PropertyEditorSupport` . 18
`java.util.EventListener` ... 13
`java.util.EventObject` ... 13
JavaBeans ... **9**, 40, 45, 48
JavaBeans Activation Framework ... **21**
JavaBeans Bridge for ActiveX ... 48
JDBC ... 4, 39
JDK ... 2
JFC ... 4
JNDI ... 34
JVM ... 5

K

Klassen
BeanInfo ... 16
Introspector ... 16
java.util.EventListener ... 13
java.util.EventObject ... 13
PropertyEditorSupport ... 18
Komponente ... 48
Granularität ... 9
Komponentenausführungsumgebung ... 32
Komponentendefinition ... 48
kontextsensitiv ... 24

L

Legacy-Systeme ... 45, 48
Life-Cycle ... 43
Loadbalancing ... 37

M

Mehrfachvererbung ... 2
Methoden
create ... 42
ejbCreate ... 42
ejbFind ... 42
find ... 42
findDataItem ... 31
fireItemAvailable ... 31
fireItemRevoked ... 31
getCustomizerClass() ... 19
getSource() ... 13
start() ... 6
stop() ... 6
toString() ... 13
Microsoft ... 10, 46
MIME-Typ ... 23

O

Object Management Group ... 46
Object Request Broker ... 46
Object Transaction Service ... 47
Operator Overloading ... 2
ORB ... 46

P

Package ... **4**
Passivation ... 40
Persistenz ... **19**, 39, 43

Plattformunabhängigkeit 5, 10
Portabilität 10
Producer 26
Property **14**
Property Editor 17
  Default 18
Property Editor Manager 18
Property Sheet 17
Property Table 40
Public Interface 12

R

Recovery 42, 43
Reflection API 4, 16
Remote Bean Klasse 35
Remote Interface 35, **41**
Remote Method Invocation 4, 46
Remote Procedure Call 46
Rendezvous-Konzept 27
RMI 4, 34, 46
Rollback 39
`RowsetAccess` 30

S

`ScrollableRowsetAccess` 30
`Serializable` 20
Serialization **19**
Session Bean 36, **42**
`SessionBean` 41, 42
SessionSynchronization 38
Sicherheitskonzept 6
Signatur 3
Simple Property 14
Skeletons 35
start() 6
stop() 6
Stubs 35
Swapping 40
Synchronisationskonzept 40

T

Templates 2
`toString()` 13
Transaktion 38
Transaktionsmanagement **38**
Transaktionssteuerung 38
`transient` 20
Type Registry 23

U

UID 19
Unicode 8

W

Webbrowser 6