Hochschule München

Seminararbeit

im Fach

Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

Entity-Control-Boundary als Komponentenmodell für Geschäftskomponenten

Name:	Dhami, Gurpreet
Studiengruppe:	IB4A (Wirtschaftsinformatik)
Matrikelnummer:	06334612
Anschrift:	Brienner Straße 39, 80333 München
Name des betreuenden Dozenten:	Michael Theis

Plagiatserklärung

Ich erkläre hiermit, dass ich meine Seminararbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe.

München, den 04.06.2014 _____

Inhaltsverzeichnis

Abbildungsverzeichnis

Einleitung

Im Rahmen der Anwendungsentwicklung werden viele Funktionen immer auf die ähnliche oder gleiche Art und Weise implementiert. Daraus ergeben sich im Laufe der Zeit viele Entwurfsmuster. Einige dieser Entwurfsmuster können als Schablonen für den Entwurf neuer Software verwendet werden, da sie die Entwicklung und Wartung der Software vereinfachen. Sie helfen auch dabei die Zuordnung fachlicher Verantwortlichkeiten der einzelnen technischen Softwaremodule übersichtlich zu gestalten. Außerdem helfen sie dabei, die Kommunikation der bei der Entwicklung beteiligten Personen zu verbessern. Einige dieser zusammenhängenden Muster werden auch zu Komponentenmodellen zusammengefasst. Eines dieser Modelle nennt sich Entity-Control-Boundary-Komponentenmodell (kurz ECB).

Ziel dieser Arbeit ist es den Einsatz dieses Komponentenmodells bei der Entwicklung von Java Enterprise Edition (*kurz* Java EE-) Anwendungen zu untersuchen. Hinter der Java Enterprise Edition steckt die Idee die Entwicklung großer Anwendungen soweit wie möglich zu vereinfachen, indem viele wiederkehrende Aufgaben und Funktionen, die fast in jeder neuen Anwendung benötigt werden, zu Spezifikationen zusammenzufassen. Diese Spezifikationen können von Softwareherstellern in Form eines Applikationsservers implementiert werden. Ein Applikationsserver stellt diese Implementierungen als Dienste zur Verfügung. Der Entwickler einer neuen Java EE-Anwendung kann auf die bereits vorhandenen Dienste des Applikationsservers aus seiner Anwendung aus zugreifen und sich hauptsächlich auf die Entwicklung der fachlichen Logik seiner Anwendung konzentrieren. Da die Entwicklung solcher Anwendungen trotz dieser arbeitsteiligen Programmierung nicht trivial ist, wird in der Praxis versucht die Komplexität durch den Einsatz von Mehrschichtenarchitekturen und Komponentenmodellen zu reduzieren.

Mit Hilfe des ECB kann eine bestimmte Schicht von der, häufig verwendeten, sog. Drei-Schichten- Architektur weiter strukturiert werden. Daher wird neben dem ECB auch die Drei-Schichten-Architektur im Rahmen dieser Arbeit ausführlich vorgestellt. Anschließend wird die Umsetzung des ECB bei der Implementierung einer Java-EE-Anwendung erläutert. Des weiteren werden auch die Ansätze von 'Design by Contract' eingeführt. Abschließend wird das ECB-Modell bezüglich seiner Stärken und Schwächen bewertet.

Zu Beispiel hatte bei der Erläuterung der Kombination aus Remote Interface für entfernt Clients und Implementierungsklasse für lokale Clients ein kleines Codebeispiel nicht geschadet.

Die Code Beispiele sind teilweise in sich nicht stimmig: So wird zum Beispiel beim Dual-View das Local-Interface vom Remote-Interface abgeleitet, das Session Bean implementiert dann aber doch beide Interfaces. In diesem Fall würde die Implementierung des Local-Interfaces langen.

Anmerkung:

Beim Zitieren wurde nicht immer darauf geachtet, auf die Original-Herkunft zu verweisen. Wenn zum Beispiel aus meinen Folien das

Zitat über das Fassaden-Pattern aus dem GOF-Buch erwähnt wird, dann sollte auch das GOF-Buch als Quelle aufgeführt werden

1 Was verbirgt sich hinter diesem Komponentenmodell?

Bei dem Entity-Control-Boundary Komponentenmodell handelt sich um eine Zusammensetzung von mehreren Entwurfsmustern für die Organisation des Codes innerhalb der Geschäftslogikschicht der Drei-Schichten-Architektur.

Die Drei-Schichten-Architektur, wie in der Abbildung 1 ersichtlich, besteht aus der Präsentationsschicht, Geschäftslogikschicht und der Integrationsschicht.

Abbildung 1: Drei-Schichten-Architektur Modell

Quelle: Vgl. http://tschutschu.de/resources/SS2014_01_Architektur.pdf, S. 5

Die Präsentationsschicht des Drei-Schichten-Modells enthält den Code, der für die Interaktion der Software mit der Umwelt vorgesehen ist. Hierbei handelt es sich in der Regel um eine Komponente, die eine graphische Oberfläche anbietet. Darin kann mit Hilfe verschiedener Technologien die Darstellung des Applikation implementiert werden. Sie greift auf die Geschäftslogikschicht zu, um dem Benutzer die Funktionalität der Anwendung zur Verfügung zu stellen. Ein umgekehrter Zugriff ist allerdings nicht vorgesehen. Auch ein direkter Zugriff von der Präsentationsschicht auf die Integrationsschicht oder umgekehrt ist nicht gestattet.[1]

Die Geschäftslogikschicht bildet den Kern einer Anwendung. Sie ist wiederum von der Integrationsschicht abhängig. Die Integrationsschicht integriert andere Anwendungen und Datenbanken und stellt diese der Geschäftslogikschicht zur Verfügung. Die Geschäftslogikschicht ruft beispielsweise Methoden auf der Integrationsschicht auf, um beispielsweise die Entitäten zu erhalten, auf die später näher eingegangen wird. In der Geschäftslogikschicht kommt das Entity-Control-Boundary Komponentenmodell zum Einsatz.

Die Geschäftslogikschicht kann somit die Anfragen aus der Präsentationsschicht oder auch direkt von anderen Anwendungen direkt entgegennehmen, sie verarbeiten und mit Hilfe der

1 Vgl. http://de.wikipedia.org/wiki/Schichtenarchitektur

Integrationsschicht Daten aus einer Datenbank verändern oder sie darin abspeichern oder an andere Anwendungen weiterreichen.[1]

Daraus ergeben sich drei Hauptaufgabenbereiche für die Geschäftslogikschicht. Daher wird sie in die drei Komponenten Boundary, Control und Entity aufgeteilt. Eine Umsetzung der Drei-Schichten-Architektur u.a. mit einem detaillierten Komponentenmodell zeigt die Abbildung 2. Hierbei ist die Präsentationsschicht als UI Component abgebildet, die Business Component bildet die Geschäftslogikschicht. Die Integrationsschicht ist hierbei als Resource Access Component dargestellt.[1]

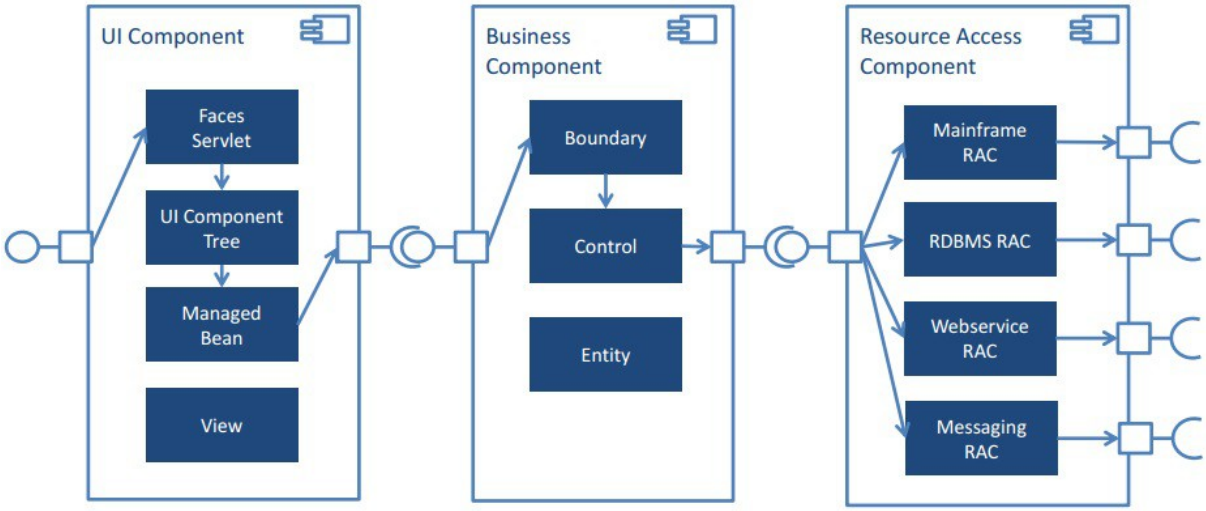

Abbildung 2: Detaillierte Drei-Schichten-Architektur
Quelle: Vgl. http://tschutschu.de/resources/SS2014_01_Architektur.pdf, S. 12

Mit Hilfe des ECB-Komponentenmodells können sowohl einzelne Use-Cases aber auch ganze Geschäftsprozesse einheitlich im System abgebildet werden. Eine bestimmte Packagestruktur und einige Namenskonventionen sind dabei von großer Hilfe, um eine gewisse Ordnung zu gewährleisten.

Wie in der Abbildung 3 dargestellt, wird ein Use-Case analysiert und auf die einzelnen Komponenten des ECB aufgeteilt. In dieser Abbildung sind auch die UML Stereotypen zu sehen, mit denen die einzelnen Komponenten in einem UML-Diagramm dargestellt werden. In der Abbildung 4 ist ein Beispiel eines umgesetzten ECB zu sehen. Darin existieren zwei Boundary-Komponenten, die eine Control-komponente verwenden. Die Control-Komponente wiederum verwendet die drei Entity-Komponenten um ein Use-Case abzubilden. Eine Entity-Komponente kann auch Referenzen auf andere Entity-Komponenten haben, wie hier es der Fall ist.

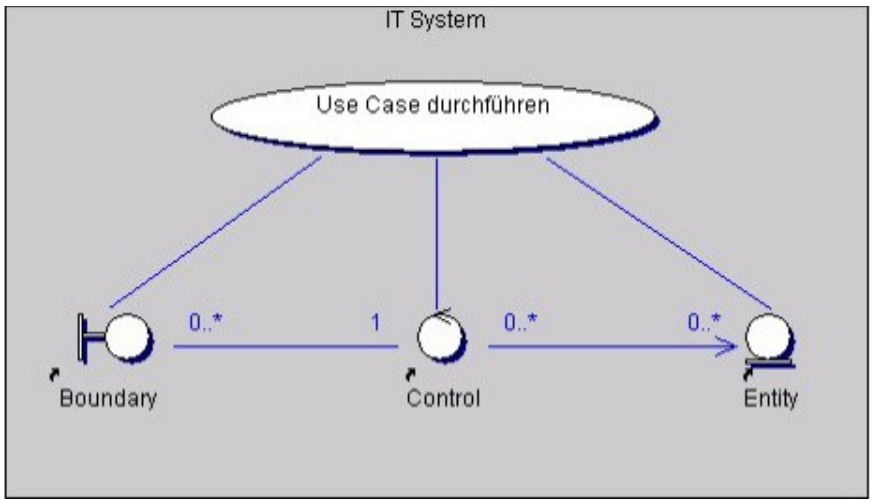

Abbildung 3: Abbildung eines Use-Cases mit ECB

Quelle: http://agilewiki.ipponsoft.de/doku.php?id=entity-control-boundary

Abbildung 4: Umsetzung der Geschäftslogik mit ECB

Quelle: http://epf.eclipse.org/wikis/openup/core.tech.common.extend_supp/guidances/guidelines/entity_control_boundary_pattern_C4047897.html

In diesem Fall sind alle Komponenten einzelne Java-Klassen. In der Regel werden Packages eingesetzt, um die einzelnen Komponenten voneinander abzukapseln und zu ordnen, damit eine möglichst hohe Kohäsion und eine lose Kopplung erreicht wird.

Unter hoher Kohäsion versteht man, dass ähnliche, zusammengehörende Aspekte einer Software möglichst von einer Komponente abgedeckt werden und dass jede Komponente auf möglichst wenige Verantwortlichkeiten spezialisiert ist. Bei der losen Kopplung werden die Komponenten modular gegen Interfaces implementiert und sie sind dadurch leichter austauschbar, ohne den Code der verwendenden Komponente verändern zu müssen.[2,3]

2 Vgl. http://de.wikipedia.org/wiki/Koh%C3%A4sion_(Informatik)

3 Vgl. http://de.wikipedia.org/wiki/Kopplung_(Softwareentwicklung)

Wie bereits am Anfang erwähnt setzt sich das ECB aus mehreren Entwurfsmustern bzw. Patterns zusammen. Die Boundary wird durch den Service Facade Pattern abgebildet. Das Control besteht u.a aus einem Service Pattern und vielen weiteren je nach Anwendung. Die Entity wird durch ein Persistent Domain Object abgebildet. Weitere hilfreiche Patterns, die im Zusammenhang mit dem ECB Anwendung finden, sind in der Abbildung 5 aufgeführt.

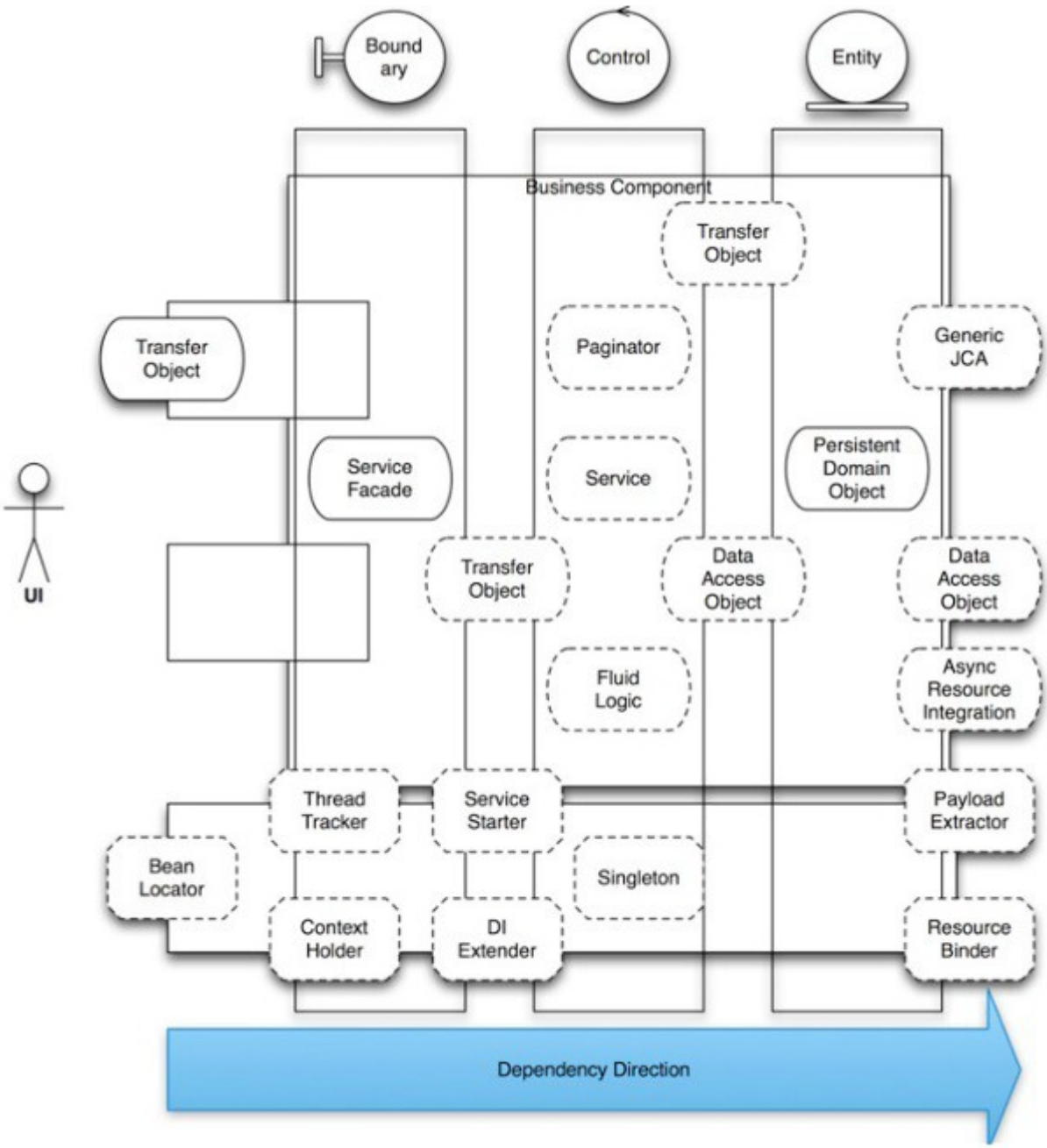

Abbildung 5: Weitere Entwurfsmuster im ECB

Quelle: http://download.java.net/general/podcasts/real_world_java_ee_patterns.pdf , S.26

2 Welche Rollen spielen die einzelnen Beteiligten E,C,B?

Bevor man die einzelnen Rollen der beteiligten Komponenten genauer untersucht, ist es sinnvoll davor ein analoges Beispiel aus der realen Welt anzuschauen, da die Modelle in der Regel eine Abbildung der Realität darstellen. In der Abbildung 6 sind die klassischen Rollen in einem fiktiven Unternehmen dargestellt, in dem zwei Personen eine Arbeit erledigen.

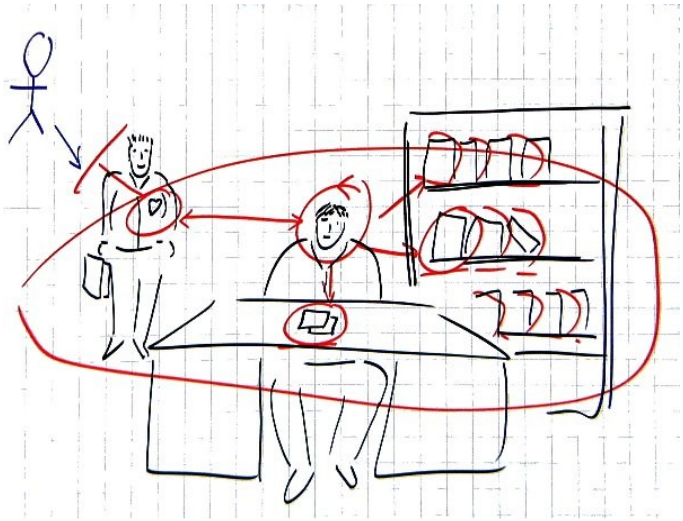

Abbildung 6: Rollen der Beteiligten

Quelle: http://agilewiki.ipponsoft.de/doku.php?id=entity-control-boundary

Es sind drei Akteure abgebildet und einige Objekte. In der linken obereren Hälfte sieht man einen Akteur außerhalb des Systems, bei dem es sich beispielsweise um einen Kunden handeln könnte. Darunter ist ein weiterer Akteur an der Systemgrenze abgebildet, der den Außendienstmitarbeiter darstellt. Er bildet hierbei die Boundary des Unternehmens, da er von außen die Aufträge herein holt, die zu bearbeiten sind. Er übernimmt auch die Auslieferung des Produkts bzw. Leistung an den Kunden. Der Kunde muss nicht wissen, wie dieses Unternehmen intern arbeitet. In der Mitte sitzt ein Sachbearbeiter an einem Schreibtisch, worauf ein Dokument liegt. Der Sachbearbeiter bearbeitet die Aufträge bzw. Dokumente nach einer bestimmten Logik und ist daher als Control gekennzeichnet. Im Hintergrund ist noch ein Regal mit weiteren Dokumenten zu sehen. Diese stellen die einzelnen Entitäten dar. Wenn man nun von mehreren Außendienstmitarbeitern und Sachbearbeitern ausgeht, die auf verschiedene Produkt- oder Leistungsarten spezialisiert sind, bekommt man eine gute Analogie zum ECB-Komponentenmodell. Die einzelnen Außendienstmitarbeiter müssen in diesem Beispiel sicherstellen, dass die Bestellung des Kunden vom Auftragseingang bis zur

Fertigstellung fehlerfrei abgearbeitet wird. Erstens müssen sie sicherstellen, dass die Kunden diese Bestellung auch tätigen dürfen, weil sie sich beispielsweise dem Unternehmen gegenüber einwandfrei authentifizieren, auch geschäftsfähig sind und eine einwandfreie Bonität haben. Zu jeder Entgegennahme eines Auftrags startet der Außendienstmitarbeiter eine neue Bearbeitung bzw. Transaktion. Je nach Wunsch des Kunden beauftragt er die entsprechenden Sachbearbeiter für die bestellten Produkte bzw. Leistungen. Wenn ein Sachbearbeiter eine bestimmte Leistung eines anderen Sachbearbeiters für die Leistungserstellung benötigt, kann er diesen direkt danach fragen. Eine Bestellung durch einen Sachbearbeiter an den Außendienstmitarbeiter wäre in diesem Fall ein Overhead, den man vermeiden sollte. Beide Sachbearbeiter können Aufträge vom Außendienstmitarbeiter entgegennehmen, verarbeiten, ins Regal stellen, diese aus dem Regal holen und auch an den Außendienstmitarbeiter ausliefern, damit dieser in unserem Fall auch den Versand der erstellten Produkte, Dokumente, Leistungen usw. an den Kunden einleiten kann. So ähnlich läuft es auch innerhalb der Geschäftslogikschicht ab. Der genaue technische Ablauf wird im Folgenden erläutert.

2.1 Boundary

Boundary steht im Englischen für eine Grenze bzw. Bereichsgrenze. Bei der Boundary kommt das Entwurfsmuster Facade zum Einsatz, der von den Autoren Gamma et.al., besser bekannt unter dem Namen „Gang of Four“ bzw. GoF in ihrem Buch Design Patterns wie folgt beschrieben wurde: „Facade: Provide a unified interface to a set of interfaces in a system. Facade defines a higher-level interface that makes the subsystem easier to use.“[4] Das bedeutet, dass die Boundary selbst Schnittstellen anbietet, mit der auf die Funktionalität des Gesamtsystems ihrer Schicht einfacher zugegriffen werden kann.

Sie wird durch alle Boundary-komponenten der fachlichen Komponenten repräsentiert. Sie bildet eine einfache und saubere Abgrenzung zwischen der Präsentationsschicht und der Geschäftslogikschicht. Sie stellt Schnittstellen bzw. öffentliche Methoden zur Verfügung, über die auf die Funktionalität der Businessschicht zugegriffen werden kann, ohne die innere Struktur der Businessschicht zu kennen. Ihr Zweck besteht unter anderem darin die Komplexität der Businessschicht vor dem Aufrufer zu verbergen und zu abstrahieren.

4 Vgl. http://tschutschu.de/resources/SS2014_06_EJB.pdf, S. 20

Sie ist auch erforderlich, da sie die Präsentationsschicht von der Geschäftslogikschicht bezüglich der Austauschbarkeit dieser beiden Komponenten sehr hilfreich ist. Außerdem hilft sie dabei die Performance der Anwendung zu verbessern, da nicht die feingranularen Methoden der Geschäftslogik über Remote-Interfaces aufgerufen werden müssen, sondern nur wenige grob-granulare Methoden. Daher sollte sie möglichst die grob-granulare Geschäftslogik abbilden und auch sicherstellen, „dass die gesamte Geschäftslogik immer konsistent ausgeführt wird“ und „dass nur Benutzer mit den erforderlichen Zugriffsrechten Zugang [zur Geschäftslogik] erhalten“[5].

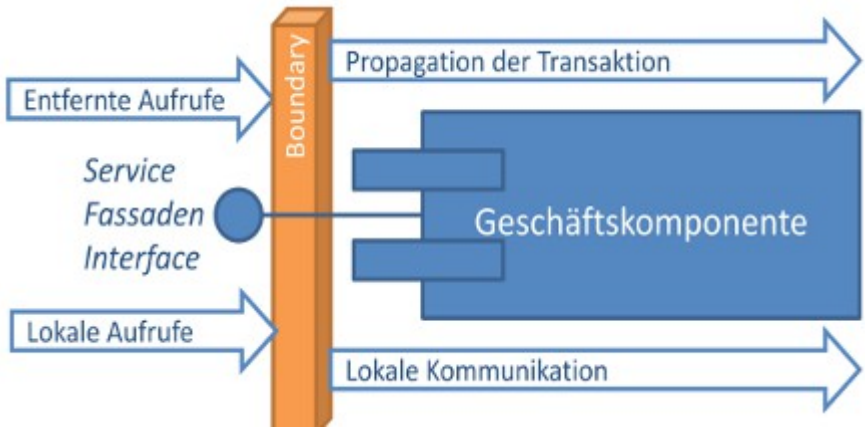

Abbildung 7: Boundary und Geschäftskomponente

Quelle: http://tschutschu.de/resources/SS2014_06_EJB.pdf

Zu der grob-granularen Geschäftslogik gehört u.a. die Validierung der Eingabeparameter und Aufruf der benötigten Controls in einer notwendigen Reihenfolge mit entsprechenden Parametern. Die eigentliche fein-granulare Geschäftslogik sollte in den verschiedenen Control-Komponenten realisiert werden. Es kann jedoch manchmal auch sinnvoll erscheinen die Geschäftslogik bereits in der Boundary-komponente zu implementieren anstatt neue Control-Klassen zu erstellen. Beispielsweise bei einer Implementierung, die lediglich Entitäten erzeugt, verändert oder löscht. Ein eigenes Control dafür wäre ein sog. Anti-Pattern, was wenig sinnvoll wäre[6]

5 http://tschutschu.de/resources/SS2014_06_EJB.pdf, S. 21

6 Vgl. Bien, 2012. S.59

2.2 Control

Die Control-Klassen des ECB-Modells beinhalten die Kernfunktionalität der Businessschicht. Ein Control bildet in der Regel eine wiederverwendbare Komponente, die aus mehreren Klassen bestehen kann. Er bietet die fein-granulare Geschäftslogik an. Diese kann von der Facade bzw. Boundary für die Abbildung eines bestimmten Use-Cases verwendet kann. Ein Control wiederum verwendet die darunterliegende Schicht, um beispielsweise Daten von einer Datenbank oder einem Service zu beziehen oder an diese zu versenden. Ein Control geht häufig aus einem Refactoring einer Boundary hervor. Falls die Boundary zu komplex und unübersichtlich wird, dann kann die kohäsive Funktionalität in eigene Controls ausgelagert werden. Ein Control wird als ein POJO (Plain Old Java Object) implementiert. Er unterliegt automatisch dem Management durch Contexts and Dependency Injection (kurz CDI). Da die Boundary für das Erstellen einer neuen Transaktion zuständig ist, muss der Control sich nicht um die Transaktion kümmern. Die Boundary koordiniert die verschiedenen Controls in einem Transaktionskontext.[7]

2.3 Entity

Entities sind einer der Hauptbestandteile einer Anwendung. Sie bilden die Objekte aus der Realität in der Anwendung ab. Die Entitykomponente spielt vor allem bei der domänen-getriebenen Entwicklung, auch Domain-Driven-Development genannt, eine große Rolle. Anforderungen an die Geschäftslogik werden mit Hilfe der Entities beschrieben und implementiert. Dabei wird sehr stark objektorientiert programmiert und nur ein geringer Teil der Logik in Controls prozedural implementiert.[8]

Bei der serviceorientierten Programmierung bei sog. serviceorientierten Architekturen nehmen die Entities jedoch eine eher passive Rolle von einfachen Datenträgern ohne spezielle Geschäftslogik ein. Bei nicht-trivialer Geschäftslogik ist diese Variante nicht besonders empfehlenswert, da sie die Vorteile der Objektorientierung vollkommen außer acht lässt und mehr komplexeren Code in den Controls erfordert. Beispielsweise kommt es dabei zu langen If-Blöcken, um die Typprüfungen zu machen.[9]

7 Vgl. Bien, 2012. S.83-85

8 Vgl. Bien, 2012. S. 92 ff

9 Vgl. Bien, 2012. S.91

3 Wie lässt sich dieses Komponentenmodell mit Java EE-Mitteln umsetzen?

Um Geschäftslogikschicht mit Hilfe des ECB-Komponentenmodell abzubilden muss eine bestimmte Packagestruktur beachtet werden. Das oberste Package wird gemäß einer Konvention mit dem inversen Domainnamen der Organisation dargestellt gefolgt vom Projektnamen und dem Namen der Schicht 'business'. Letztere kann aber auch anders benannt werden.

Für jede neue fachliche Komponente sollte ein Unterpackage erstellt werden, welches wiederum die drei Unterpackages für die Entity, Control und Boundary enthält. Im Boundary-Package sind dann alle Boundary-Klassen und Interfaces enthalten ohne irgendwelche Namenszusätze wie z.B. Boundary, Service, Impl oder ähnliches. Genauso sollten sich die einzelnen Klassen und Interfaces der Controls in einem Package namens Control befinden und die Entities im Package Entity.[10] In der Abbildung 8 ist beispielsweise die Packagestruktur einer fachlichen Komponente 'registrations' in Netbeans zu sehen. Die jeweiligen implementierten Klassen sind in der Abbildung 9 innerhalb der Unterpackages boundary, control und entity zu sehen.

Abbildung 8: Beispiel für die Packagestruktur der Geschäftslogikschicht

Quelle: https://github.com/AdamBien/javaee-bce-pom.git

Abbildung 9: Implementierte Klassen im ECB

Quelle: https://github.com/AdamBien/javaee-bce-pom.git,

10 Vgl. Bien, 2012. S.59, 85

Abgesehen von in einigen Ausnahmen dürfen die Komponenten innerhalb der Geschäftslogikschicht untereinander kommunizieren. Ein direkter Zugriff von einer Boundary auf eine andere Boundary sollte vermieden werden. Hingegen ist der Zugriff von einer Control-Komponente auf eine andere Control-Komponente ohne den Umweg über dessen Boundary durchaus richtig, da dies ein 'over-engineering' vermeidet. Dabei wird beispielsweise eine Control-Klasse der fachlichen Komponente X in eine Control-Klasse der fachlichen Komponente Y importiert.[11]

3.1 Implementierung der Boundary

Die Implementierung der Boundary erfolgt in der Regel als ein Stateless Session Bean. Dazu wird die Annotation @Stateless an die Boundary-Klasse angehängt. Sie kann ein Interface enthalten, wenn sie von außerhalb der Java Virtual Machine (*kurz* JVM) zugänglich sein soll. Sie kann jedoch auch ohne Interfaces implementiert werden.Die Konfiguration ohne Interface wird als No-Interface-View bezeichnet.[12]

Um einen lokalen oder internen Zugriff handelt es sich, wenn der Aufruf innerhalb der gleichen JVM von der höheren Schicht der Anwendung erfolgt, beispielsweise von der Präsentationsschicht auf die Geschäftslogikschicht der Anwendung innerhalb der gleichen Instanz des Applikationsservers. Um einen remote oder externen Zugriff handelt es sich, wenn der Aufruf aus einer anderen JVM beispielsweise über die RMI bzw. 'Remote Method Invocation' Schnittstelle erfolgt oder innerhalb des gleichen Applikationsservers von einer Anwendung auf eine andere Anwendung erfolgt. Es kann sich dabei beispielsweise um einen Swing-Client oder auch um einen zweiten Applikationsserver handeln, in dem die Präsentationsschicht läuft.

Für den lokalen Zugriff kann ebenfalls ein Local-Interface implementiert werden. Entweder wird ein zweites Interface neben dem Remote-Interface, wie in Abbildung 10 zu sehen, oder als eine Vererbung von Remote- Interface, wie in Abbildung 11 zu sehen, hinzugefügt. Diese Konfiguration wird als Dual-Interface-View bezeichnet.

11 Vgl. Bien, 2012. S. 62

12 Vgl. Bien, 2012. S. 62

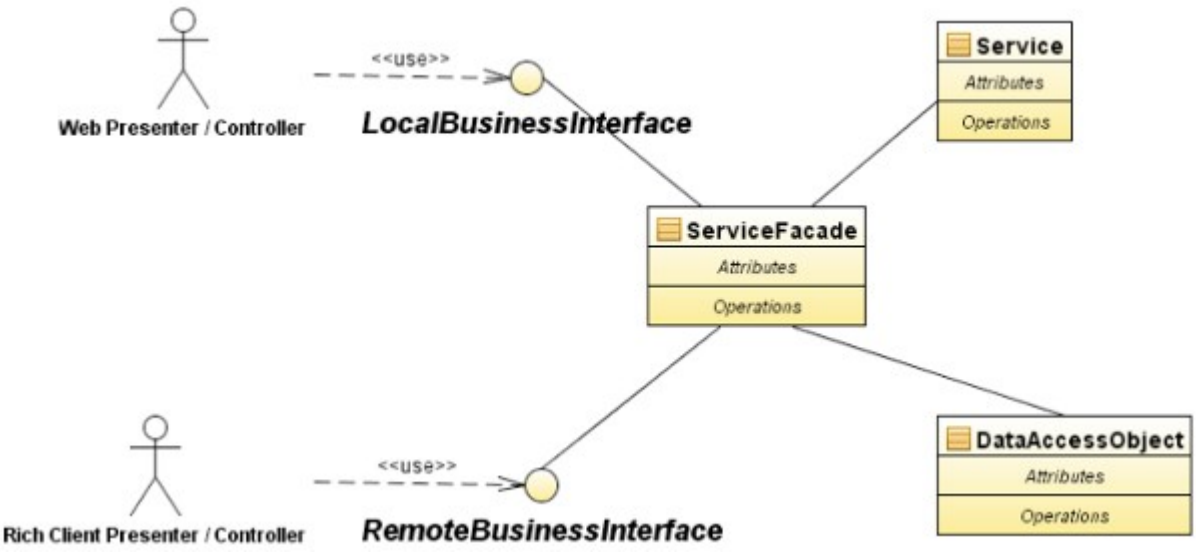

Abbildung 10: Local und Remote Interfaces

Quelle: http://download.java.net/general/podcasts/real_world_java_ee_patterns.pdf , S.31

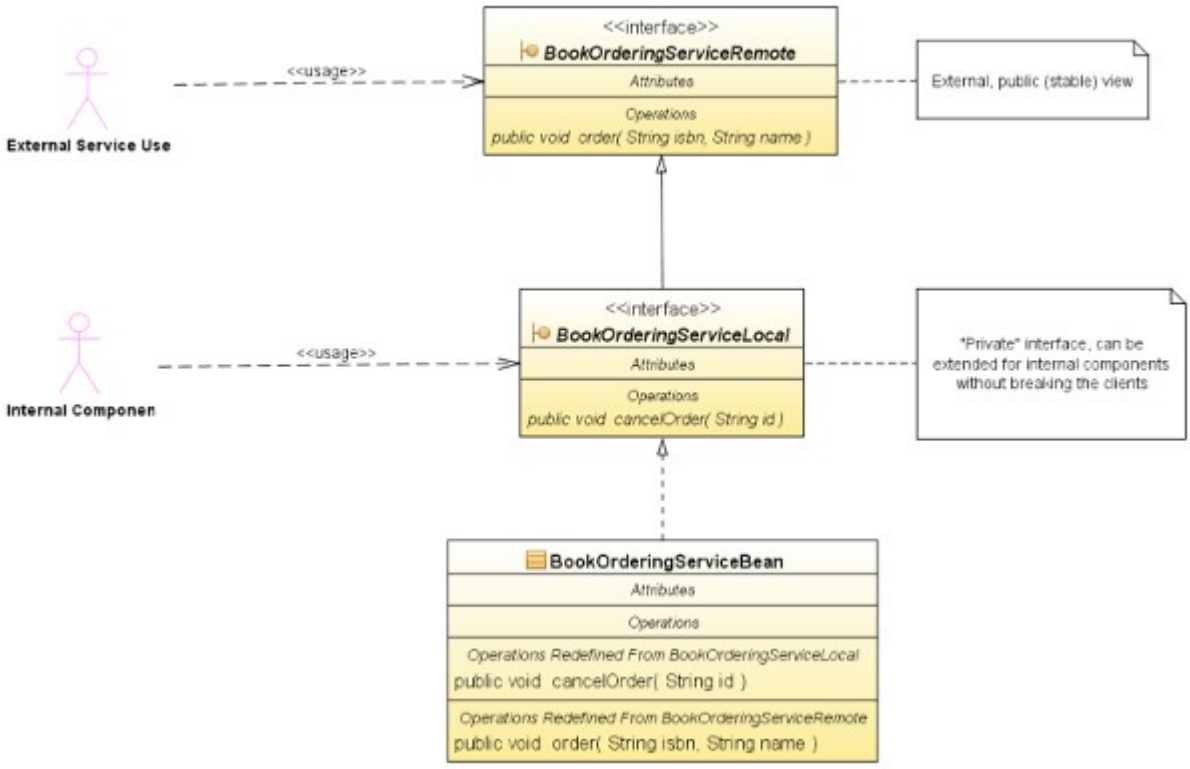

Abbildung 11: Dual-View Boundary

Quelle: http://download.java.net/general/podcasts/real_world_java_ee_patterns.pdf , S.33

Der Einsatz von Remote-Interfaces im lokalen Umfeld sollte vermieden werden, da dies unter Umständen unnötige Ressourcen des Applikationsservers verschwendet, da in diesem Fall nicht die Referenzen der Objekte überreicht werden sondern Kopien der Objekte.[13]

In den Abbildungen 12 und 13 sind die zwei Interfaces zu sehen, die das Dual-View repräsentieren. Diese werden von der Boundary-Klasse Registrations in der Abbildung 14 implementiert. Der Vorteil von Interfaces liegt darin, dass sie ein sog. 'Design by Contract' ermöglichen. Dem Aufrufer wird lediglich eine Interface-Datei zur Verfügung gestellt und auch zugesichert, dass beim Aufruf dessen Methoden die Funktionalität in einem bestimmten Datenformat zur Verfügung gestellt bekommt. Die implementierende Klasse kann beliebig oft bearbeitet oder ausgetauscht werden, solange sie die angegebenen Interfaces richtig implementiert.

```
package com.airhacks.workshops.business.registrations.boundary;

public interface IRegistrationsRemote{

    Registrations find(int registrationId);

    Registrations register(Registrations request);
}
```

Abbildung 12: Remote Interface der Boundary

Quelle:https://github.com/AdamBien/javaee-bce-pom.git

```
package com.airhacks.workshops.business.registrations.boundary;

public interface IRegistrationsLocal extends  IRegistrationsRemote{

    Registrations find(int registrationId, String bla);

    Registrations register(Registrations request, String bla);
}
```

Abbildung 13: Local Interface der Boundary abgeleitet vom Remote Interface

Quelle:https://github.com/AdamBien/javaee-bce-pom.git

Damit der Applikationsserver weiß, welches Interface für die lokalen und welches für die entfernten Aufrufe der implementierten Klasse zuständig ist, werden an die implementierende Klasse Annotations @Remote([interfaceklasse].class) und @Local([interfaceklasse].class) vorangestellt.

13 Bien, 2012. S.62-63

```java
package com.airhacks.workshops.business.registrations.boundary;

import javax.ejb.*;
import javax.inject.Inject;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import com.airhacks.workshops.business.registrations.control.*;

@Stateless
@Local(IRegistrationsLocal.class)
@Remote(IRegistrationsRemote.class)
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public class Registrations implements IRegistrationsLocal, IRegistrationsRemote
{
    @PersistenceContext
    EntityManager em;

    @Inject
    VatCalculator priceCalculator;

    @Override
    public Registrations register(Registrations request) {
        Registrations registration = em.merge(request);
        int netPrice = registration.getNetPrice();
        int totalPrice = priceCalculator.calculateTotal(request.isVatIdAvailable(), netPrice);
        registration.setTotalPrice(totalPrice);
        return registration;
    }
```

Abbildung 14: Implementierung der Interfaces in einer Boundary

Quelle:https://github.com/AdamBien/javaee-bce-pom.git

Die Boundary hat in diesem Modell auch die Aufgabe eine neue Transaktion zu erzeugen und dies wird durch die folgende Annotation an der Boundary-Klasse erreicht.

@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)

Außerdem haben die injizierten Felder keine Zugriffsmodifizierer, sodass sie innerhalb des Packages sichtbar sind.

Damit auch bestimmte entfernte Anwendungen über JAX-RS, JAX-WS, Hessian und IIOP auf die Boundary zugreifen können, sollten diese auch im Boundary Package liegen.

3.2 Implementierung des Controls

Bei den Control-Komponenten handelt es sich in der Regel um Enterprise Java Beans (EJBs) oder um CDI managed Beans. Sie sind als POJOs implementiert. In der Regel ist weder ein Interface noch eine Annotation erforderlich, um sie als solche zu kennzeichnen. Bei der Verwendung eines Controls in einer anderen Klasse muss die Referenzvariable entweder mit @EJB oder @Inject annotiert werden, damit eine Instanz des Controls vom Applikationsserver injiziert werden kann. Es ist sinnvoll dem Control mit der Annotation wie beispielsweise @TransactionAttribute(TransactionAttributeType.MANDATORY) mitzuteilen, ob er beim Aufruf eine Transaktion erfordert, ob er eine bestehende Transaktion verwenden soll oder eine neue Transaktion starten soll. Beim ECB wird in der Regel die Transaktion von der Boundary gestartet und die Controls werden innerhalb dieser ausgeführt, sodass der Transaktionstyp Mandatory hierfür notwendig ist.[14]

```
package com.airhacks.workshops.business.registrations.control;

import java.math.BigDecimal;
import static java.math.BigDecimal.valueOf;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;

@TransactionAttribute(TransactionAttributeType.MANDATORY)
public class VatCalculator {

    public int calculateTotal(boolean vatIdAvailable, int price) {
        BigDecimal net = valueOf(price);
        if (vatIdAvailable) {
            return net.intValue();
        } else {
            return net.add(net.multiply(valueOf(0.19))).intValue();
        }
    }
}
```

Abbildung 15: Implementierung eines Controls

Quelle:https://github.com/AdamBien/javaee-bce-pom.git

Ein Control enthält in der Regel viel prozeduralen Code. Bei der Domänengetriebenen Entwicklung enthält der Control weniger Code, da er nur noch die domänen-übergreifende Funktionalität abdecken muss.[15]

14 Bien, 2012. S.83-85

15 Vgl. Bien, 2012. S.83

3.3 Implementierung des Entity

Bei den Entity-Komponenten handelt es sich um einfache POJOs, die mit @Entity annotiert sind. Entities können mit Hilfe der Java Persistence API (JPA) persistiert werden. Sie werden von Controls mit Hilfe des sog. Entitymanagers im Persistence Context von JPA erzeugt und persistiert, wenn es sich um JPA Entities handelt.

Entities, die nicht persistiert werden sollen, werden mit @Transient annotiert.

Je nach dem, ob eine serviceorientierte oder domänen-getriebene Anwendung erforderlich ist werden die Entities 'schlank' oder 'fett' gestaltet. Bei schlanken Entities sind hauptsächlich nur die Attribute vorhanden, die in der Regel public sind. Die Funktionalität der Entity bzw. Methoden sind in den Controls zu finden, die die Entities 'bearbeiten'. Bei den 'fetten' Entities handelt es sich um übliche, objektorientiert geschriebene Java-Klassen, die die Geschäftslogik in sich tragen. Nur die Geschäftslogik, die alle Entities gleichermaßen betrifft wird zentral in Controls ausgelagert, die prozedural eine Entity nach der anderen abarbeiten.[16]

```
package com.airhacks.workshops.business.registrations.entity;

import ...3 lines

@Entity
public class Registration {

    @Id
    @GeneratedValue
    private long id;

    private int numberOfDays;
    private int numberOfAttendees;
    private boolean vatIdAvailable;

    private final static int DAILY_PRICE = 300;

    private int totalPrice;

    public Registration(boolean vatIdAvailable, int numberOfDays, int numberOfAttendees) {
        this.numberOfDays = numberOfDays;
        this.numberOfAttendees = numberOfAttendees;
        this.vatIdAvailable = vatIdAvailable;
    }
```

Abbildung 16: Implementierung einer einfachen Entity

Quelle:https://github.com/AdamBien/javaee-bce-pom.git

16 Vgl. Bien, 2012. S. 91 ff

Bewertung und Fazit

Seit der Java Enterprise Edition 6 müssen deutlich weniger bis fast keine Konfigurationen und Infrastrukturcode geschrieben werden. Dies hilft dem Entwickler sich hauptsächlich auf die Geschäftslogik fokussieren. Doch bewahrt ihn diese Erleichterung nicht vor dem Chaos einer unstrukturierten, schlecht wartbaren Anwendung. Das Entity-Control-Boundary Komponentenmodell hilft ihm dabei die Anwendung besser zu strukturieren, da es die Anwendung der Programmierparadigmen wie beispielsweise 'Separation of concerns', DRY (Don't repeat yourself), enge Bindung und lose Kopplung fördert. Das Prinzip der Single-Responsibility bzw. engen Bindung wird durch dieses Modell sehr unterstützt. So können einzelne Komponenten modularer programmiert werden und unabhängig von anderen getestet werden. Diese sind wiederum nach außen lose gekoppelt, sodass sie leicht erweiterbar und austauschbar sind ohne die Kompatibilität zu älteren Clients zu verlieren. Die Trennung der Domänenobjekte von der Geschäftslogik sorgt auch für mehr Unabhängigkeit von den darunterliegenden Schichten. Auch selbst ist die die Anwendung bei Beachtung dieses Modells unabhängig nach außen zu Domänenmodellen aus anderen Anwendungen, da der Zugriff auf andere Schichten von der Geschäftslogikschicht aus über standardisierte Technologien wie z.B. Java Persistence API (JPA) oder Java Connector API (JCA) erfolgt und nicht direkt auf die Ressourcen. Ein weiterer Vorteil dieses Modells ist auch die Flexibilität der Strukturierung, sodass es kaum zu einem sog. 'over-engineering' kommt. Der Trade-Off für alle diese Vorteile ist jedoch die Einhaltung einer gewissen Struktur, die dieses Modell vorgibt. Dieser kleine Strukturierungs-overhead, der durch die Trennung des Codes in drei Komponenten in eigenen Klassen in mehreren Packages könnte durchaus auch als Nachteil gesehen werden, wenn beispielsweise eine triviale Anwendung sonst einfacher und schneller entwickelt werden könnte. Dennoch sollte beachtet werden, dass eine strukturierte Anwendung allerdings einfacher und leichter gewartet werden kann. Somit ist es insgesamt sinnvoller eine Komponentenmodell wie dieses zu verwenden.

Literaturverzeichnis

Monographie:

Bien, Adam (2012): Real World Java EE Patterns. 2nd Edition. S.l: press.adam-bien.com.

Online:

„Architektur verteilter Anwendungen“. Zugegriffen 09. Mai 2014.
http://tschutschu.de/resources/SS2014_01_Architektur.pdf.

„entity-control-boundary [Agile Wiki]“. Zugegriffen 09. Mai 2014.
http://agilewiki.ipponsoft.de/doku.php?id=entity-control-boundary.

„Geschäftskomponenten mit EJBs“. Zugegriffen 09. Mai 2014.
http://tschutschu.de/resources/SS2014_06_EJB.pdf.

„Java EE BCE POM Gitgub Projekt“. Zugegriffen 14. Mai 2014.
https://github.com/AdamBien/javaee-bce-pom.git.

„Kohäsion (Informatik) – Wikipedia“. Zugegriffen 13. Mai 2014.
http://de.wikipedia.org/wiki/Koh%C3%A4sion_(Informatik).

„Kopplung (Softwareentwicklung) – Wikipedia“. Zugegriffen 13. Mai 2014.
http://de.wikipedia.org/wiki/Kopplung_(Softwareentwicklung).

„Real World Java EE Patterns“. Zugegriffen 14. Mai 2014.
http://download.java.net/general/podcasts/real_world_java_ee_patterns.pdf.

„Schichtenarchitektur – Wikipedia“. Zugegriffen 09. Mai 2014.
http://de.wikipedia.org/wiki/Schichtenarchitektur.