

Oridexa

CHRISTIAN SCHRADER

c.schrader@interexa.de

16. Februar 2009

interexa AG

Inhaltsverzeichnis

1	Übersicht	3
1.1	Umgesetzte Anforderungen	4
2	Komponenten	7
2.1	Enterprise Java Beans	7
2.1.1	Session Beans	7
2.1.1.1	LossSessionBean	8
2.1.1.2	LossCollectionSessionBean	8
2.1.2	Entity Classes	9
2.1.2.1	Loss	10
2.1.2.2	LossCollection	11
2.1.2.3	AuditTrail	12
2.2	JAAS	12
2.2.1	Customizing der Berechtigungen	13
2.3	Java Server Faces	13
2.3.1	UI Components	14
2.3.2	Backing Beans	14
2.3.3	Navigationsregeln	15
2.4	Exception Handling	16
2.5	Internationalisierung	17
3	Installation	21
3.1	Installation der Basissoftware	21
3.1.1	Installation von Glassfish	21
3.2	Installation von ORIDEXA	22
3.3	Einrichten der IDE	23
	Literaturverzeichnis	28

1 Übersicht

ORIDEXA ist als JEE-Anwendung konzipiert, die einen Webclient und einige EJB-Komponenten hat. Der Webclient wurde mit Java Server Faces (JSF) implementiert, die EJB-Komponenten mit EJB 3.0 SessionBeans und EntityClasses. Ziel der Umsetzung war es, eine einfache aber funktionale Demonstration der Java-Komponenten zu geben. Die Architektur ist dabei nahe an der Anwendung Duke's Bank Application aus dem SUN Java EE 5 Tutorial angelehnt (siehe dazu (JENDROCK 2006), Kapitel 37). Abbildung 1.1 auf der nächsten Seite gibt eine high-level Übersicht über die Architektur.

Die Benutzerverwaltung beruht auf JAAS, diese beziehen die Benutzer- und Rolleninformationen direkt aus dem Application Server. Es handelt sich dabei um ein rein deklaratives Berechtigungskonzept.

Funktional wurden zentrale Grundanforderungen des interexa ORC RE umgesetzt. Im System können Schadensfälle (Losses) erfasst werden. Daneben können Sammelschäden (LossCollections) definiert werden, denen die Schadensfälle zugeordnet werden können. Bei jeder Änderung an einem Schadensfall wird ein Vermerk im Fachobjekt gespeichert (Audit Trail).

Die folgenden Anwendungsfälle sind im System verfügbar:

- ▷ Erfassen, Bearbeiten und Löschen eines Schadensfalls
- ▷ Ansicht einer einfachen Schadensfall-Historie
- ▷ Erfassen, Bearbeiten und Löschen eines Sammelschadens
- ▷ Zuordnung von Sammelschäden und Schadensfällen
- ▷ Abschließen von Schadensfällen
- ▷ Listenansicht der Schadensfälle
- ▷ Listenansicht der Sammelschäden
- ▷ Eingabe einer dynamischen Suchanfrage nach Schadensfällen

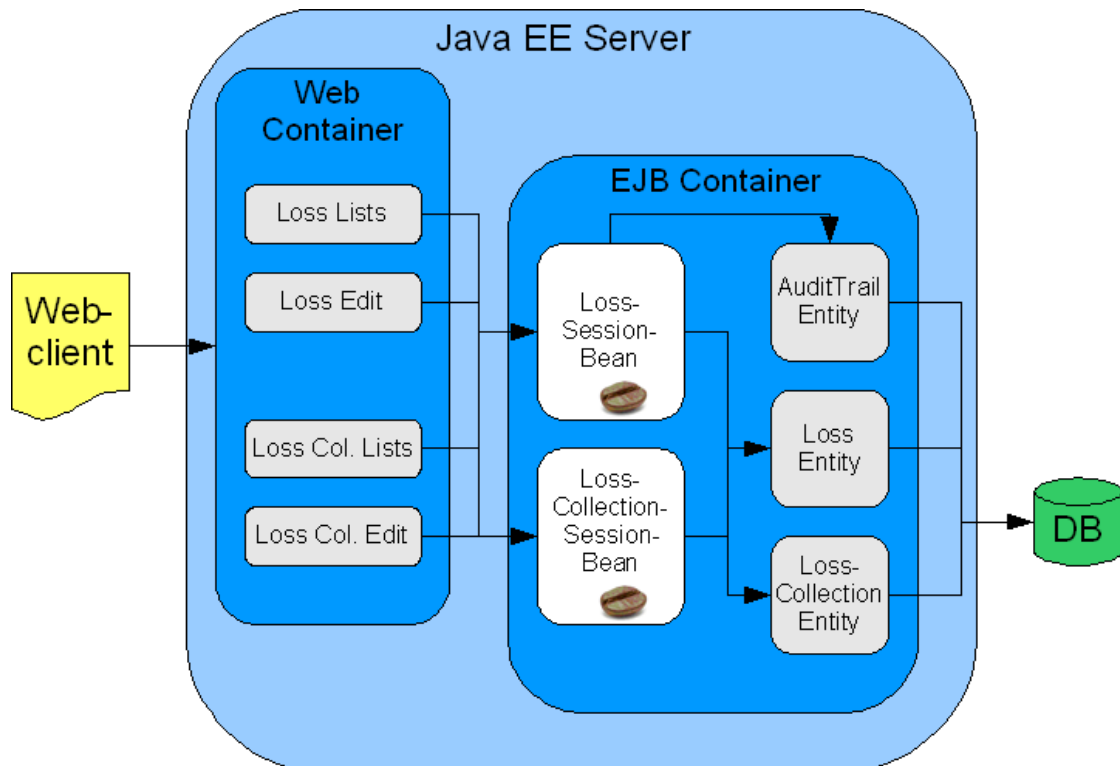


Abbildung 1.1: High-Level Übersicht zur Architektur

Das Löschen von Datensätzen ist dabei nur Mitgliedern der Rolle *Manager* zur Verfügung gestellt, ebenso das Abschließen von Schadensfällen. Alle anderen Funktionen auch den Mitgliedern der Rolle *User*. Weitere Informationen zum Berechtigungskonzept sind dem entsprechenden Kapitel zu entnehmen.

1.1 Umgesetzte Anforderungen

Zu den obigen UseCases wurden für die Entwicklung folgende Anforderungen spezifiziert:

1. Erfassen / Ändern von Schadensfällen
 - a) Felder siehe Kapitel 2.1.2.1 auf Seite 10.
 - b) Die Felder sollen auf mehreren Reitern dargestellt werden.
 - c) Das Feld *Kurzbeschreibung* soll ein Pflichtfeld sein.
 - d) Das Feld *netValue* berechnet sich aus *lossValue* abzüglich *insuranceValue*.
 - e) Bei allen Änderungen und beim Erfassen soll ein Zeitstempel gespeichert werden.

- f) Änderungen an den Feldern *shortDescription*, *longDescription*, *lossValue* und *netValue* sollen nachvollziehbar sein, d.h. die alten Werte müssen in einem AuditTrail gespeichert werden.
 - g) Die Auswahl des zugehörigen Sammelschadens ist aus dem Schadensfall heraus über eine Auswahlliste möglich.
 - h) Das Speichern eines Schadensfalls soll nur dann möglich sein, wenn dieser nicht abgeschlossen ist.
2. Ansicht eines einzelnen Schadensfalls
- a) Alle Felder sollen auf einer Seite angezeigt werden.
 - b) Der Audit Trail wird unterhalb der Daten angezeigt.
 - c) Es ist ein direkter Wechsel auf die Bearbeiten-Seite des Datensatzes möglich.
3. Erfassen / Ändern von Sammelschäden
- a) Felder siehe Kapitel 2.1.2.2 auf Seite 11.
 - b) Die Felder sollen auf einem Reiter dargestellt werden.
4. Ansicht eines einzelnen Sammelschadens
- a) Alle Felder sollen auf einer Seite angezeigt werden.
 - b) Es ist ein direkter Wechsel auf die Bearbeiten-Seite des Datensatzes möglich.
5. Listenansichten (Schadensfälle und Sammelschäden)
- a) Aus den Listenansichten heraus müssen Buttons auf *Ansicht*, *Bearbeiten* und *Löschen* existieren.
 - b) Die Listenansichten zeigen 10 Einträge und einen Pager zum Wechseln der Seiten.
 - c) Die Listen können nach einer Spalte aufsteigend oder absteigend sortiert werden.
6. Listenansichten der Schadensfälle
- a) Es existiert eine Listenansicht der Schadensfälle unter dem Menüpunkt *Auswertungen* mit dem Namen *Schadensfall Übersicht*.
 - b) Über eine Auswahlliste kann die Liste eingeschränkt werden auf *Offene Schadensfälle* und *Abgeschlossene Schadensfälle*.
 - c) Über ein Häkchen kann von einem *Manager* ein Schadensfall abgeschlossen oder geöffnet werden. (*Freigabe*)

7. Löschen von Datensätzen

- a) Nur Mitglieder der Rolle *Manager* dürfen Datensätze löschen.

8. Abfrageeditor nach Schadensfällen

- a) Der Abfrageeditor erlaubt das Absetzen einer Anfrage direkt in der EJB Query Language.

9. Ansicht der Hauptseite

- a) Auf der Hauptseite sind die letzten 5 angelegten, offenen Schadensfälle zu sehen.
- b) Die Listenansicht unter 9. a) enthält Buttons auf *Ansicht* und *Freigabe*.
- c) Unter den Schadensfällen befinden sich die letzten 5 angelegten Sammelschäden.

2 Komponenten

2.1 Enterprise Java Beans

Die eigentliche Geschäftslogik ist in den EJBs abgebildet. Für jeden genannten UseCase existiert eine Funktion in einer Stateless Session Bean, die diesen abbildet. Die Entity Classes werden in der Persistenzschicht genutzt und als Datenobjekt an die Oberfläche gereicht.

Im vorliegenden Projekt werden nicht alle JEE-Komponenten genutzt. So werden beispielsweise Message Driven Beans, Timer etc. nicht verwendet. Für einen Überblick über weitere Komponenten aus JEE siehe (JENDROCK 2006, HAUGLAND et al. 2004, LANGER und REIBERG 2006).

2.1.1 Session Beans

In den Session Beans ist die eigentliche Geschäftslogik gekapselt. Jeder Anwendungsfall wird dabei in einer public-Methode einer jeweiligen Session Bean abgebildet. Die Anwendung ORIDEXA enthält zwei Session Beans: LossSessionBean und LossCollectionSessionBean.

Obwohl die Namensgebung eine Zugehörigkeit zur jeweiligen Entity Class suggeriert, ist dies nicht ausschließlich der Fall. Session Beans können mehrere Entity Classes nutzen. Die Designentscheidung für eine Session Bean liegt also nicht in der zugehörigen Entity, sondern in der logischen und fachlichen Zusammengehörigkeit der von der Bean bereitgestellten Funktionen. Da es sich beim vorliegenden Projekt in erster Linie um eine simple CRUD¹-Anwendung handelt, ist die Nähe zu den jeweiligen Entity Classes gegeben.

Da die EJBs im gleichen Server wie der Webclient liegen, sind alle Methoden nur in der lokalen Schnittstelle offen gelegt. Session Beans können sowohl statefull als auch stateless sein. Im ersteren Fall wird die Bean über die Session einem Client zugeordnet, der Programmierer kann sich also auf den Status der Bean verlassen. Bei den einfachen Use Cases im vorliegenden Fall sind stateless Session Beans ausreichend. Eine statefull Session Bean wäre zum Beispiel dann nötig, wenn über mehrere Funktionsaufrufe hinweg an einem

¹Create, Retrieve, Update, Delete

Objekt gearbeitet werden müsste, beispielsweise in einem komplexeren Workflow oder in einem Dialog (Wizzard).

2.1.1.1 LossSessionBean

Diese stateless Session Bean enthält die Funktionen, die zur Handhabung der Schadensfälle benötigt werden. Die Methoden sind in Tabelle 2.1 aufgeführt.

Methoden	Beschreibung
<code>void createLoss(Loss loss);</code>	Speichert einen Schadensfall in der DB.
<code>void removeLoss(long id);</code>	Entfernt einen Schadensfall. Darf nur von einem Manager aufgerufen werden.
<code>void updateLoss(Loss loss);</code>	Führt eine Aktualisierung eines Schadensfalls in der DB durch.
<code>List<Loss> getAllLosses();</code>	Gibt eine Liste aller Schadensfälle aus der DB zurück.
<code>List<Loss> getOpenLosses();</code>	Gibt eine Liste aller noch nicht abgeschlossenen Schadensfälle aus der DB zurück.
<code>List<Loss> getClosedLosses();</code>	Gibt eine Liste aller abgeschlossenen Schadensfälle aus der DB zurück.
<code>List<Loss> getLossesByQuery(String query);</code>	Gibt eine Liste aller Schadensfälle zurück, die mit der übergebenen Suchabfrage gefunden wurden.
<code>Loss getLossById(long id);</code>	Gibt einen Schadensfall mit der übergebenen ID zurück.
<code>void closeOrOpenLoss(long id);</code>	Führt den einfachen öffnen / abschließen Workflow auf dem Objekt aus. Darf nur von einem Manager aufgerufen werden.

Tabelle 2.1: Methoden der LossSessionBean

In den Funktionen *createLoss* und *updateLoss* werden an das jeweilige Objekt die AuditTrail-Objekte angehängt. Diese speichern unter anderem den Zeitpunkt und den Änderer bzw. Erfasser des Datensatzes. Auf anderen Wege können AuditTrail-Objekte nicht erzeugt werden.

2.1.1.2 LossCollectionSessionBean

Diese stateless Session Bean enthält die Funktionen, die zur Handhabung der Sammel-schäden benötigt werden. Die Methoden sind in Tabelle 2.2 auf der nächsten Seite aufgeführt.

Methoden	Beschreibung
<code>void createLossCollection (LossCollection lossCollection);</code>	Speichert einen Sammelschaden in der DB.
<code>void removeLossCollection (long id);</code>	Entfernt einen Sammelschaden. Darf nur von einem Manager aufgerufen werden.
<code>void updateLossCollection (LossCollection lossCollection);</code>	Führt eine Aktualisierung eines Sammelschadens in der DB durch.
<code>List<LossCollection> getAllLossCollections();</code>	Gibt eine Liste aller Sammelschäden aus der DB zurück.
<code>List<LossCollection> getLossCollections- ByQuery(String query);</code>	Gibt eine Liste aller Sammelschäden zurück, die mit der übergebenen Suchabfrage gefunden wurden.
<code>LossCollection getLossCollectionById (long id);</code>	Gibt einen Sammelschaden mit der übergebenen ID zurück.
<code>List<Loss> getLossesForCollection (LossCollection lossCollection);</code>	Gibt eine Liste aller Schadensfälle zurück, die zum übergebenen Sammelschaden gehören.

Tabelle 2.2: Methoden der LossCollectionSessionBean

Die Funktion *getAllLossesForLossCollection* verwendet Loss-Objekte. Diese werden über eine Query zugehörig zur aktuellen LossCollection ausgewählt.

2.1.2 Entity Classes

Entity Classes bilden die Datenbasis für die Session Beans. Eine Session Bean kann dabei mehrere Entity Classes nutzen.

Streng genommen handelt es sich bei Entity Classes nicht um Enterprise Java Beans, wie es noch bei Entity Beans (EJB Version 2.1) der Fall war. Es handelt sich um POJOs, die als solche auch an den Webcontainer durchgereicht werden können. Damit entfällt die Notwendigkeit für reine Data Transfer Objects (DTO), die noch bei EJB 2.1 nötig waren.

Die Entity Classes verwenden zur Generierung ihrer IDs jeweils eine eigene Sequence. Sollte das System auf eine Datenbank deployt werden, die keine Sequences unterstützt,

so müssten die Annotations derart geändert werden, dass die IDs über eine eigene Tabelle generiert werden.

2.1.2.1 Loss

Die Klasse Loss bildet das Fachobjekt Schadensfall ab. Die verwendete Sequence ist *Loss_seq*.

Feldname	Beschreibung
Long id	Eindeutige ID des Objektes.
boolean almostLoss	Gibt an, ob es sich um einen Beinaheschaden handelt.
boolean estimated	Gibt an, ob es sich um einen Schätzwert handelt.
String shortDescription	Kurze, einzeilige Beschreibung des Schadensfalls.
String longDescription	Ausführliche Beschreibung des Schadensfalls.
float lossValue	Schadenshöhe (vor Abzügen) in Euro.
float insuranceValue	Minderungen des Schadensfalls.
float netValue	Netto Schadenshöhe nach Abzug der Minderungen.
Date dateOfDiscovery	Datum, an dem der Schadensfall entdeckt wurde.
Date dateOfOccurence	Datum, an dem der Schadensfall aufgetreten ist.
LossCollection lossCollection	Zugehöriger Sammelschaden.

Tabelle 2.3: Felder des Schadensfalls

Für die Anforderung, dass der Nettoschaden aus Schadenshöhe minus Versicherungshöhe berechnet werden soll, werden in der Entity Class Callback-Methoden verwendet. Dafür wurde die Methode calculateNetValue mit den Annotations @PrePersist und @PreUpdate markiert. Eine Methode kann, wie in diesem Fall, mehrere Annotations haben, es darf jedoch für jede Annotation nur eine Methode geben.

Callback-Methoden können RuntimeExceptions auslösen, die die laufende Transaktion abbrechen und einen Rollback verursachen. Callback-Methoden dürfen keine Entity-Manager oder Query-Methoden aufrufen. Tabelle 2.4 auf der nächsten Seite zeigt die möglichen Callback-Methoden von Entity Classes.

Da es sich bei der Berechnung des Nettoschadens um eine Business-Regel handelt, müsste diese eigentlich in der Logik-Schicht, also in der Session Bean durchgeführt werden. Der Weg über die Callback-Methode wurde in erster Linie gewählt, um deren Verwendung zu demonstrieren. Da von der in diesem Fall notwendigen Logik allerdings keine Beziehungen zwischen Entities betroffen sind, lässt sich auch die Aufnahme der Funktion

Typ	Beschreibung
@PrePersist	Wird ausgeführt, bevor der Entity Manager das tatsächliche Speichern ausführt.
@PreRemove	Wird ausgeführt, bevor der Entity Manager das tatsächliche Löschen ausführt.
@PostPersist	Wird ausgeführt, nachdem der Entity Manager das tatsächliche Speichern ausführt hat, also nach dem DB INSERT-Statement.
@PostRemove	Wird ausgeführt, nachdem der Entity Manager das tatsächliche Löschen ausführt hat.
@PreUpdate	Vor dem DB UPDATE-Statement.
@PostUpdate	Nach dem DB UPDATE-Statement.
@PostLoad	Nach dem Laden oder dem Refresh der Entity aus dem Persistence Context.

Tabelle 2.4: Callback-Methoden für Entity Classes

in die Entity Class direkt rechtfertigen. Zu einer Diskussion darüber welche Geschäftslogik in Entity Classes ihren Platz finden darf, siehe (ALUR et al. 2003), S. 50.

2.1.2.2 LossCollection

Die Klasse LossCollection bildet das Fachobjekt Sammelschaden ab. Die verwendete Sequence ist *LossCol_seq*.

Feldname	Beschreibung
Long id	Eindeutige ID des Objektes.
String shortDescription	Kurze, einzeilige Beschreibung des Sammelschadens.
String longDescription	Ausführliche Beschreibung des Sammelschadens.

Tabelle 2.5: Felder des Sammelschadens

Zwischen LossCollection und Loss besteht eine unidirektionale ManyToOne-Beziehung. Zu einem Sammelschaden können also mehrere Schadensfälle gehören, die Referenz ist nur vom Schadensfall aus als Property zur Verfügung gestellt. Beziehungen zwischen Klassen werden als Annotation im Quelltext kenntlich gemacht, eine zusätzliche Beschreibung der Beziehung über den Deployment-Descriptor, wie in EJB 2.1, ist nicht mehr nötig.

2.1.2.3 AuditTrail

Diese Klasse wird verwendet, um Informationen zur Änderung an des Fachobjekts Schadensfall zu speichern. Die verwendete Sequence ist *AuditTrail_seq*. Für jedes Speichern wird ein AuditTrail angelegt, jedoch nur für einige Felder wird eine Information im Feld *changeInformation* gespeichert.

Feldname	Beschreibung
Long id	Eindeutige ID des Objektes.
String editorName	Name des Benutzers, der den Datensatz geändert hat.
Date editTime	Datum und Uhrzeit der Änderung.
Loss belongsTo	Zugehöriges Loss-Objekt, für das der AuditTrail angelegt wurde.
String changeInformation	Der Inhalt einiger Felder vor der Änderung.

Tabelle 2.6: Felder des AuditTrails

Zwischen der Klasse AuditTrail und Loss besteht eine bidirektionale ManyToOne-Beziehung. Dadurch können die AuditTrails zusammen mit dem zugehörigen Loss-Objekt kaskadierend gelöscht werden. Weitere Informationen zur Verwendung von Entity Classes finden sich in (JENDROCK 2006) und (KEITH und SCHINCARIOL 2006).

Zusammenfassend ergibt sich bei den Entity Classes das Beziehungsgefüge aus Abbildung 2.1 auf Seite 19.

2.2 JAAS

JAAS, die *Java Authentication and Authorization Services*, erlauben die Implementierung der Sicherheitsfunktionen in Web- und EJB-Anwendungen. Im vorliegenden Projekt wurde ein rein deklaratives Sicherheitskonzept umgesetzt. In einem deklarativen Sicherheitskonzept werden bestimmte Funktionen an Rollen geknüpft, es wird sozusagen nur festgelegt, dass eine bestimmte Funktion von einem Träger einer bestimmten Rolle aufgerufen werden darf. Ein programmatisches Sicherheitskonzept würde benötigt, wenn innerhalb einer Funktion nach gewissen Rechten unterschieden werden müsste, beispielsweise in Abhängigkeit der Parameter. So würde eine Bedingung, dass ein Benutzer nur Schadensfälle bis zu einer bestimmten Schadenshöhe freigeben darf, ein programmatisches Sicherheitskonzept voraussetzen.

Im System sind die folgenden Rollen verfügbar:

- ▷ Admin
- ▷ User
- ▷ Manager

Für das ORIDEXA wird eine Formular-basierte Authentifizierung genutzt. Bei der Formularseite handelt es sich um eine reine JSP-Seite (kein JSF). Benutzer müssen im Applikationsserver angelegt werden, es wird der File-Realm benutzt. Im Standard wird ein Mapping der Rollen auf die Gruppen `adminGroup`, `userGroup` und `managerGroup` durchgeführt.

Die benötigten Rollen werden in den SessionBeans überprüft. So ist der Aufruf der Methode `removeLoss` mit der Annotation `@RolesAllowed("Manager")` versehen, so dass nur Träger dieser Rolle die Methode aufrufen können. Gleiches gilt für `removeLossCollection` in der `LossCollectionSessionBean`.

Zur Verdeutlichung der Rollenüberprüfung wurde in der Listenansicht der Sammelschäden auch der Löschen-Button mit einer Rollenprüfung nach Manager versehen. Daher ist für Benutzer ohne die entsprechende Rolle der Link deaktiviert. Ein Deaktivieren nach Rolle ist bei den JSF-Komponenten der Standardimplementierung nicht möglich, daher wurde für den Link die entsprechende Tomahawk-Komponente verwendet.

2.2.1 Customizing der Berechtigungen

Der Vorteil eines deklarativen Berechtigungskonzeptes liegt darin, dass sich die Berechtigungen ohne Eingriff in den Quellcode ändern lassen. So können beispielsweise die Rollen, die eine bestimmte Methode einer SessionBean aufrufen dürfen, über die Datei `META-INF/ejb-jar.xml` angepasst werden. Spezifikationen dieser Datei überschreiben die Einstellungen, die über Annotations im Quelltext abgelegt wurden. Weitere Informationen zum Ändern der Berechtigungen über einen Deployment Descriptor finden sich in (JENDROCK 2006), S. 958 ff..

2.3 Java Server Faces

Für die Erstellung der Oberfläche kamen weitgehend Komponenten des Java Server Faces Frameworks zum Einsatz. JSF bietet neben den Oberflächenkomponenten auch Controller-Funktionalitäten. Dabei handelt es sich um Managed Beans (Backing Beans) und Navigationsregeln.

Für die Erstellung von JSF-Anwendungen stehen mittlerweile zahlreiche RAD-Anwendungen zur Verfügung, siehe dazu auch Abbildung 2.2 auf Seite 20. Das ORIDEXA wurde allerdings nicht mit einem solchen Oberflächendesigner erstellt, da diese Tools die Auswahl der verfügbaren UI Components oft einschränken.

Weitere Informationen zu Java Server Faces liefern (GEARY und HORSTMAN 2004) und (JENDROCK 2006), S. 275 ff..

2.3.1 UI Components

Für die Benutzeroberfläche kommen einige Komponenten aus der SUN-Implementierung, aber auch Komponenten des Apache Tomahawk-Projektes zum Einsatz. So ist beispielsweise der Tabellenpager über eine Tomahawk-Komponente realisiert, ebenso das Menü.

Anmerkung: Auf den Einsatz der Standard-Komponenten hätte verzichtet werden können, da diese vollständig in den Tomahawk-Komponenten abgebildet wurden. Durch den Einsatz beider Bibliotheken wurde allerdings gezeigt, dass für bestimmte Anwendungsfälle jederzeit Komponenten aus anderen Frameworks verwendet werden können.

An einigen Stellen wurden die Tomahawk-Komponenten verwendet, obwohl es entsprechende Komponenten in der Referenzimplementierung gibt. Die Tomahawk-Komponenten wurden bevorzugt, wenn bestimmte Features benötigt wurden, die die Referenzimplementierung nicht zur Verfügung stellte. So wurden beispielsweise die Löschen-Links der Sammelchäden mit Tomahawk-Komponenten gerendert, da diese nur für bestimmte Rollen enabled werden können.

Neben den hier genutzten Komponentenbibliotheken gibt es noch zahlreiche weitere, die sich oft ohne große Probleme miteinander kombinieren lassen. Zudem ist es leicht möglich, eigene Komponenten zu erstellen, die gänzlich neue Möglichkeiten zur Verfügung stellen oder die andere Komponenten zu einer Komponente zusammenfassen.

2.3.2 Backing Beans

Die Backing Beans bilden einen Teil des Controllers der Anwendung. Jede Seite kann mehrere Backing Beans haben, eine Backing Bean kann selbst von mehreren Seiten verwendet werden.

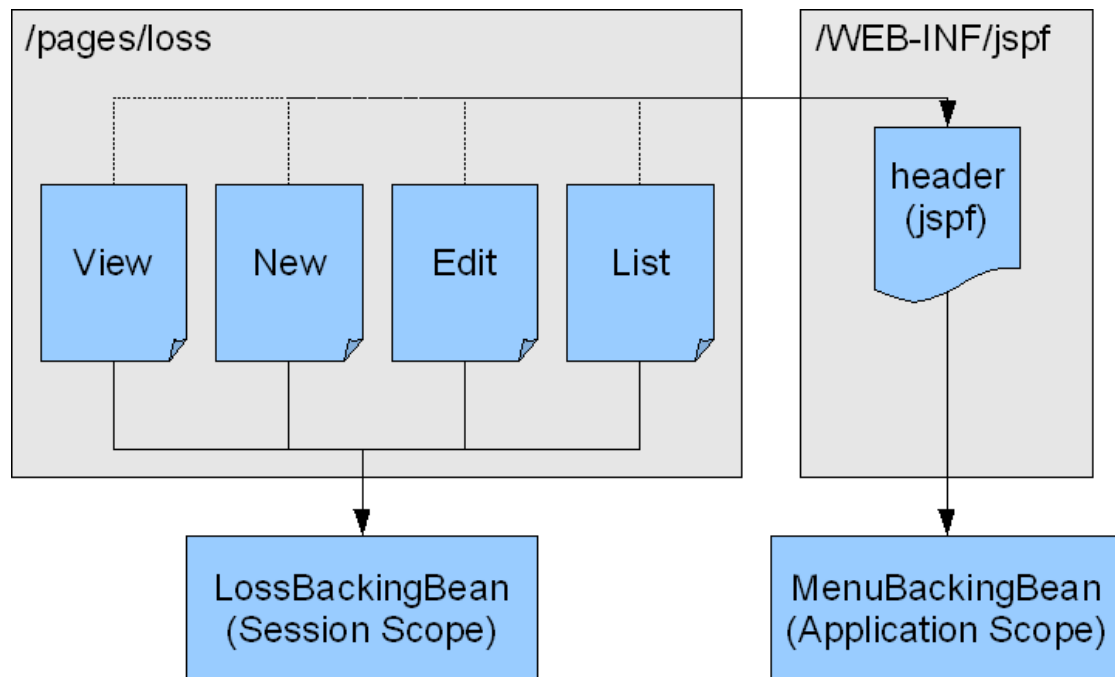


Abbildung 2.3: Nutzung der Backing Beans

Backing Beans können einen von drei Scopes haben: *Request*, *Session* oder *Application*. Ein Großteil der BackingBeans in ORIDEXA hat als Scope die Session, da damit eine Datenhaltung über mehrere Seiten hinweg (View, List, Edit, New) möglich ist.

Die MenuBean enthält die Informationen für das Anwendungsmenü. Die Bean liegt im Application Scope, da sie so nur ein mal instantiiert werden muss. Trotzdem wird das Menü bei jedem Aufruf neu erstellt. Dadurch könnte eine Unterscheidung nach Benutzern stattfinden, welche Elemente hinzugefügt werden sollen. So ließe sich die Fähigkeit des aktuellen iwat-Menüs umsetzen, dass das Menü je nach Rechten des Benutzers anders aussehen kann.

2.3.3 Navigationsregeln

Die Navigationsregeln sind bei JSF in der zentralen Konfigurationsdatei *faces-config.xml* abgelegt. Je nach Ausgangsseite und Navigationsbefehl wird eine Zielseite definiert. Da viele Navigationselemente im Menü hinterlegt sind, wird für diese Navigationsbefehle keine Ausgangsseite angegeben (keine *from-view-id*).

```
<navigation-rule>
  <navigation-case>
```

```

    <from-outcome>loss_edit</from-outcome>
    <to-view-id>/pages/loss/Edit.jsp</to-view-id>
  </navigation-case>
</navigation-rule>

```

Die Navigationsbefehle selbst werden entweder direkt angegeben, beispielsweise im Menü beim Aufruf der Übersichtsseite, oder aus Action-Methoden in den BackingBeans generiert.

Der Aufruf der Suchergebnisseite ist nur aus der Detailsuche heraus möglich. In diesem Fall wird auch die Ausgangsseite in der faces-config angegeben.

```

<navigation-rule>
  <from-view-id>
    /pages/reporting/DetailSearch.jsp
  </from-view-id>
  <navigation-case>
    <from-outcome>
      searchresult_loss
    </from-outcome>
    <to-view-id>
      /pages/reporting/searchresultLoss.jsp
    </to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>
      searchresult_losscollection
    </from-outcome>
    <to-view-id>
      /pages/reporting/searchresultLossCollection.jsp
    </to-view-id>
  </navigation-case>
</navigation-rule>

```

2.4 Exception Handling

Zur Handhabung von Exceptions existieren einige Vorgehensmodelle. Eine häufig ange-troffene ist die, dass die Exceptions in jeder Anwendungsschicht aus der darunter lie-

genden Schicht gefangen werden und in eine Exception höherer Ebene verpackt werden. Diese neuen Exceptions werden dann weiter geworfen. Die Begründung für dieses Verfahren ist, dass eine Schicht nur Kenntnis über die Exceptions der ihr benachbarten Schichten haben muss. So ist jede Schicht *self contained*, und gibt beispielsweise an einen Client keine Informationen über tiefer liegende Schichten weiter.

Dieses Verfahren hat jedoch trotzdem einige Nachteile. Die größten Probleme beim oben genannten Vorgehen, sind, dass erstens eine Ebene eine Exception fängt, die nicht weiß, wie diese zu handhaben ist. Dies sollte nie passieren. Schlimmer ist aber, dass eine höhere Ebene eventuell weiß, wie mit der Exception hätte umgegangen werden müssen. Diese höhere Ebene muss nun allerdings die originale Exception umständlich auspacken. Durch diese Menge an Ein- und Auspacken der Exceptions und an try-catch-Blöcken enthält ein Quellcode oft mehr Fehlerbehandlung als eigentliche Geschäftslogik. Dadurch wird der Code schwerer zu lesen. Auf gar keinen Fall sollte eine Ebene eine Exception fangen und eine andere Exception weiter werfen, ohne dass die originale Exception darin eingepackt wird.

Zur Diskussion um Best Practices beim Exception Handling siehe auch (KALLIO 2008).

Da im Vorliegenden Projekt alle Schichten der Architektur bekannt sind, wird auf ein gesondertes Einpacken der Exceptions auf mittleren Schichten verzichtet. Statt dessen fängt der Client die originale Exception ab, da dieser damit umgehen kann. Die einzige Ausnahme bildet hier der QueryEditor, der eine eigene Exception aus der LossSession-Bean erhält.

2.5 Internationalisierung

Unter Internationalisierung versteht man den Prozess, eine Anwendung so vorzubereiten, dass sie in unterschiedlichen Sprachen verwendet werden kann. Unter Lokalisierung versteht man das Übersetzen der Message-Bundles und anderer Dateien in die jeweils benötigte Zielsprache. Internationalisierung wird oft mit i18n abgekürzt, da beim Englischen Wort internationalization zwischen dem i und dem n 18 Buchstaben ausgelassen werden.

Die Internationalisierung von ORIDEXA findet auf zwei Ebenen statt: Auf der einen Seite werden die JSF-Seiten durch ein Message-Bundle lokalisiert. Auf der anderen Seite wird auch auf Ebene der BackingBeans die Datei messages.properties genutzt, um die Texte der jeweiligen Sprache zu finden.

Auf den JSF-Seiten kann mittels

```
<f:loadBundle basename="org.oidexa.resources.messages" var="
```

```
bundle"/>
```

ein Message-Bundle geladen werden, auf das in der Seite über den Namen `bundle` zugegriffen werden kann. In den Backing Beans wird in der Util-Methode `getLocalizedString` ein `ResourceBundle` geladen und verwendet. Existiert der gewünschte Schlüssel nicht in der message-Datei, so wird die entsprechende Exception abgefangen und der gesuchte Schlüssel mit dem Präfix und Suffix `???` zurückgegeben.

Für weitere Hinweise zur Internationalisierung von JSF-Anwendungen siehe (MANN 2005).

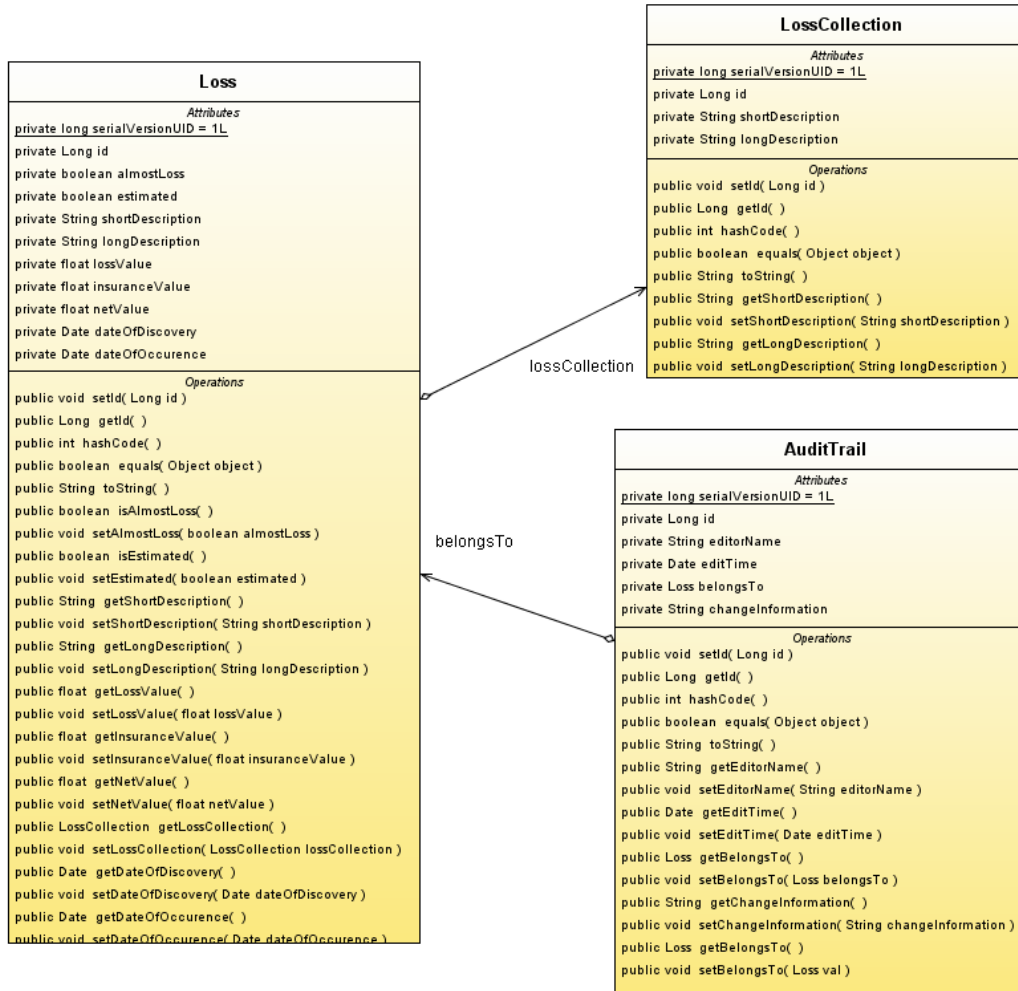


Abbildung 2.1: Entity Classes des ORIDEXA

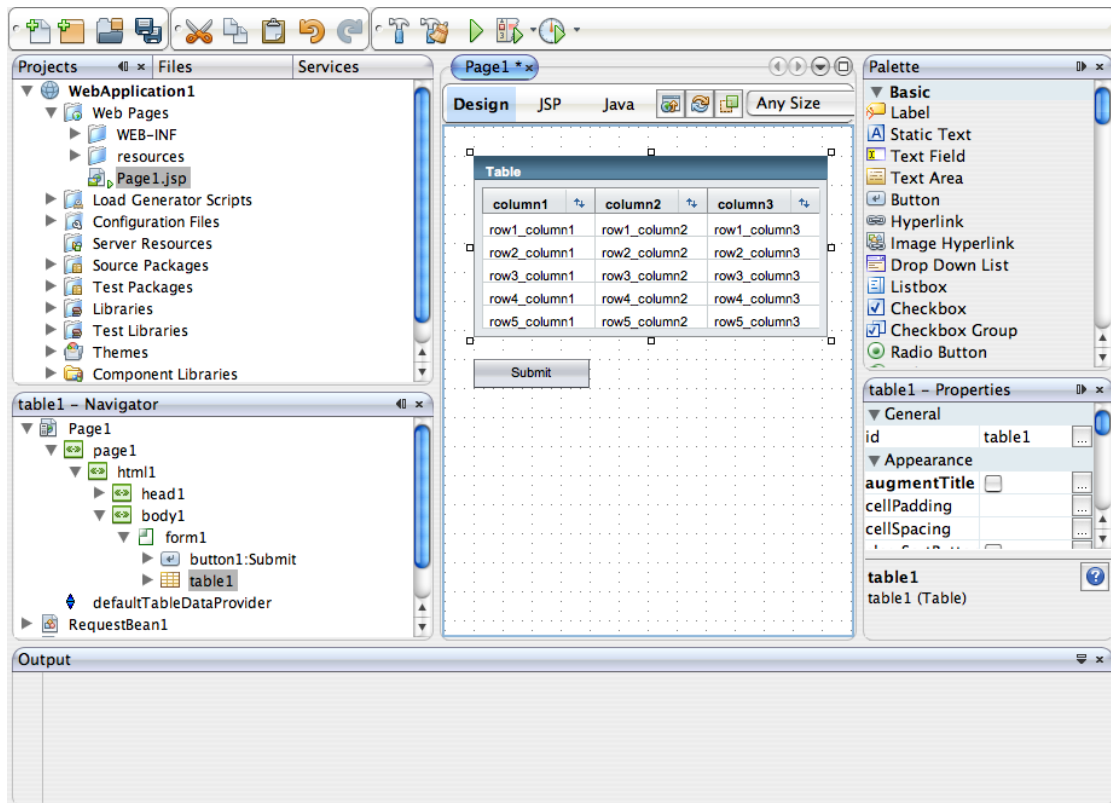


Abbildung 2.2: RAD-Anwendung für JSF

3 Installation

3.1 Installation der Basissoftware

Für den Betrieb von ORIDEXA sind folgende Bestandteile zu installieren:

- ▷ Glassfish Application Server v2ur2, alternativ SUN Application Server 9.1.
- ▷ Datenbank Apache Derby, alternativ eine andere Datenbank mit JDBC-Unterstützung.

Der Glassfish Application Server befindet sich für die Betriebssysteme Linux, Windows und Mac OS X auf der CD. Sollten bei der hier beschriebenen Installation Probleme auftreten, so sind die Installationshinweise der Application Server zu beachten.

3.1.1 Installation von Glassfish

Voraussetzung für die Installation von Glassfish ist ein JDK 5 oder ein JDK 6. Ein reines Java Runtime Environment ist nicht ausreichend.

Das jar-File für die entsprechende Systemumgebung muss an die Stelle kopiert werden, an der Glassfish installiert werden soll. Das Entpacken der Anwendung erfolgt mit

```
% java -Xmx256m -jar filename.jar
```

Danach muss in das Verzeichnis glassfish gewechselt werden. Unter Unix oder Linux muss das execute-Bit für *Ant* gesetzt werden, danach kann die Konfiguration aufgerufen werden:

```
% chmod -R +x lib/ant/bin  
% lib/ant/bin/ant -f setup.xml
```

bzw. unter Windows:

```
% lib\ant\bin\ant -f setup.xml
```

Weitere Informationen zur Installation sind unter (GLASSFISH 2008) zu finden.

3.2 Installation von Oridexa

ORIDEXA wird als EAR-Archiv geliefert. Dieses Archiv muss im oben genannten Application Server installiert werden.

Das Deployment der Anwendung erfolgt über die Adminkonsole. Diese befindet sich normalerweise unter *http://Servername:4848*. Vor dem Deployment müssen noch folgende Schritte durchgeführt werden:

- ▷ Starten der Java DB Datenbank
 - ▷ Windows: im Startmenü *Start Java DB*
 - ▷ Linux: mit dem Kommandozeilentool *JAVADB_HOME/bin/startNetworkServer*
- ▷ Starten des Application Server
 - ▷ Windows: im Startmenü *Start Default Server*
 - ▷ Linux: *GLASSFISH_HOME/bin/asadmin start-domain*
- ▷ Einloggen in die Adminkonsole (*http://Servername:4848*)
- ▷ Anlegen eines Connection Pools
 - ▷ Name: bspw. *ConnectionPool*
 - ▷ Type: *javax.sql.DataSource*
 - ▷ Database Vendor: *JavaDB*
 - ▷ Property *ConnectionAttributes*: *;create=true* (inklusive Semikolon!)
 - ▷ Property *ServerName*: *localhost*
 - ▷ *DatabaseName*: Pfad zur DB, bspw. */opt/SDK/javadb/dbs/oridexa* – Das Verzeichnis *oridexa* darf in diesem Fall noch nicht existieren, es wird automatisch angelegt.
 - ▷ Property *User*:
 - ▷ Property *Password*:
- ▷ Anlegen einer „JDBC Resource“
 - ▷ *JNDI-Name*: *jdbc/oridexa*
 - ▷ *Pool Name*: oben gewählter Name, bspw. *OridexaConnectionPool*

Unter Applications -> Enterprise Applications -> Deploy kann die EAR-Datei direkt hochgeladen werden. Siehe Abbildung 3.1 auf der nächsten Seite.



Abbildung 3.1: Deployment eines EAR in Glassfish

Die Einrichtung der Benutzer erfolgt ebenfalls über die Adminkonsole. Unter Configuration -> Security -> Real -> file auf Add Users, dort können die Benutzer angelegt werden. Die benötigten Benutzergruppen für das Standard-Mapping sind userGroup, managerGroup und adminGroup. Siehe dazu auch die Abbildung 3.2.

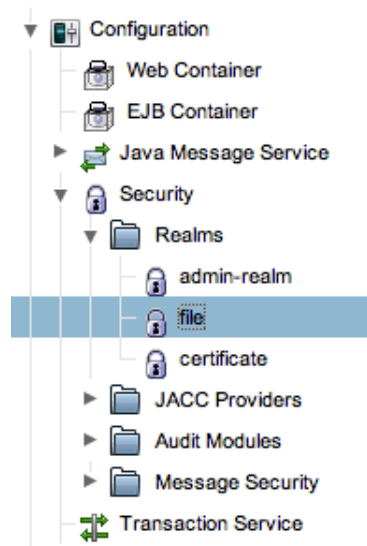


Abbildung 3.2: file-Realm im Glassfish

3.3 Einrichten der IDE

In der Entwicklung von ORIDEXA wurde NetBeans 6.5 verwendet. Zur Einrichtung des Projektes befinden sich die Quellcodes und Projektdateien auf der CD.

Das auf der CD befindliche NetBeans-Paket enthält bereits den Application Server Glassfish, eine gesonderte Installation ist daher nicht notwendig.

Die Projektdateien können direkt in NetBeans 6.5 geöffnet werden. Es handelt sich streng genommen um drei Projekte: Ein war-Projekt, das den Webclient enthält, ein EAR-Projekt, das die Session Beans und Entity Classes enthält und ein Enterprise-Projekt, das die beiden Projekte zusammenfasst.

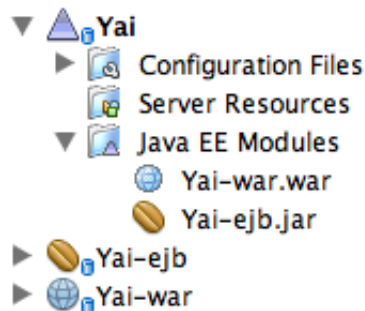


Abbildung 3.3: Oridexa-Projekte in NetBeans 6.1

Anschließend wird wahrscheinlich ein Hinweis erscheinen, dass einige Referenzen nicht aufgelöst werden konnten. Dabei handelt es sich um den Application Server und die verwendeten Bibliotheken. Der Application Server lässt sich einrichten über einen Rechtsklick auf das Projekt, dann Properties -> Run. Unter *Server* sollte der lokal eingerichtete Server (in der Defaulteinstellung Glassfish 2) ausgewählt werden.

Im Webmodul (Oridexa-war) müssen die Tomahawk-Bibliotheken konfiguriert werden. Dies erfolgt über einen Rechtsklick auf das Projekt, dann Properties -> Libraries. Es müssen alle jar-Dateien aus dem lib-Verzeichnis der CD ausgewählt werden.

Vor dem Starten der Anwendung muss eine Derby-Datenbank angelegt werden. Dies erfolgt über das Services-Fenster mit einem Rechtsklick auf den Derby-Treiber. Dort kann unter Create Database eine Datenbank angelegt werden. In der Standardkonfiguration des Projektes erwartet JDBC eine Datenbank mit dem Namen *oridexa* und dem Benutzernamen und Kennwort *oridexa*.

Nach dem Deployment ist eine Einrichtung der Benutzer vorzunehmen, siehe dazu 3.2 auf Seite 22.

Abbildungsverzeichnis

1.1	High-Level Übersicht zur Architektur	4
2.3	Nutzung der Backing Beans	15
2.1	Entity Classes des ORIDEXA	19
2.2	RAD-Anwendung für JSF	20
3.1	Deployment eines EAR in Glassfish	23
3.2	file-Realm im Glassfish	23
3.3	Oridexa-Projekte in NetBeans 6.1	24

Tabellenverzeichnis

2.1	Methoden der LossSessionBean	8
2.2	Methoden der LossCollectionSessionBean	9
2.3	Felder des Schadensfalls	10
2.4	Callback-Methoden für Entity Classes	11
2.5	Felder des Sammelschadens	11
2.6	Felder des AuditTrails	12

Index

- Apache, 14
- Application, 15
- AuditTrail, 12

- Backing Beans, 14
- Basissoftware, 21

- Callback-Methode, 10
- Controller, 14
- deklaratives Sicherheitskonzept, 12

- Enterprise Java Beans, 7
- Entity Class, 9
- Entwicklungsumgebung, 23
- Exception, 16

- faces-config.xml, 15

- Glassfish, 21

- i18n, 17
- Installation, 21
- Internationalisierung, 17

- JAAS, 12
- Java Authentication and Authorization Services, 12
- Java Server Faces, 13

- Lokalisierung, 17
- Loss, 10
- LossCollection, 11
- LossCollectionSessionBean, 8
- LossSessionBean, 8

- Navigationsregeln, 15
- NetBeans 6.5, 23

- programmatisches Sicherheitskonzept, 12

- RAD, 14
- Request, 15

- Sammelschaden, 11
- Schadensfall, 10
- Session, 15
- Session Bean, 7

- Tomahawk, 14

- UI Components, 14

Literaturverzeichnis

- [ALUR et al. 2003] ALUR, DEEPAK, J. CRUPI und D. MALKS (2003). *Core J2EE Patterns*. Prentice Hall, 2 Aufl.
- [GEARY und HORSTMAN 2004] GEARY, DAVID und C. HORSTMAN (2004). *Core Java Server Faces*. Prentice Hall.
- [GLASSFISH 2008] GLASSFISH (2008). *GlassFish Project - V2 UR2 Final Build*.
<https://glassfish.dev.java.net/downloads/v2ur2-b04.html>.
- [HAUGLAND et al. 2004] HAUGLAND, SOLVEIG, M. CADE und A. ORAPALLO (2004). *J2EE 1.4 The Big Picture*. Prentice Hall.
- [JENDROCK 2006] JENDROCK, ERIC (2006). *The Java EE 5 Tutorial*. Addison Wesley, 3 Aufl.
- [KALLIO 2008] KALLIO, SAMI (2008). *Difficult Choices in Handling Exceptions in Enterprise Java Applications*.
<http://blog.decaresystems.ie/index.php/2006/04/07/difficult-choices-in-handling-exceptions-in-enterprise-java-applications/>.
- [KEITH und SCHINCARIOL 2006] KEITH, MIKE und M. SCHINCARIOL (2006). *Pro EJB 3 Java Persistence API*. Apress.
- [LANGER und REIBERG 2006] LANGER, TORSTEN und D. REIBERG (2006). *J2EE und JBoss*. Hanser.
- [MANN 2005] MANN, KITO (2005). *Java Server Faces in Action*. Manning.