



TLGen Handbuch

EJB3 Backend Code-Generator

Version 2.5 (Deutsch)

Autoren:

Titus Livius Rosu (tituslivius.rosu@stardata.eu)

Titus Rosu (titus.rosu@stardata.eu)

Liviu Rosu (liviu.rosu@stardata.eu)

Letzte Änderung: 21. Juni 2011

Inhaltsverzeichnis

ABBILDUNGSVERZEICHNIS	4
TABELLENVERZEICHNIS	4
1. ALLGEMEINE INFORMATION.....	5
1.1 ZIELE	7
1.2 TLGEN'S ARCHITEKTUR-KONZEPT	8
1.2.1 <i>Input für die Code-Generierung</i>	9
1.3 BESCHREIBUNG DER MÖGLICHKEITEN DURCH TLGEN	9
1.3.1 <i>Code Generierung mit einem Domain Model (UML)</i>	10
1.3.2 <i>Code Generierung mit einer schon vorhandenen Datenbank</i>	11
2. TLGEN CODE-GENERIERUNG UND DIE ANALYSE VON DATENBANKEN.....	12
2.1 ANALYSE DES DOMAINMODELLS UND DER DATENBANK	13
2.2 DATEN KLASSEN - INTERFACES	17
2.3 SESSION BEAN	18
2.3.1 <i>Interfaces für eine Session Bean</i>	20
2.3.2 <i>XML Persistente Datei</i>	21
2.3.3 <i>Interface Klassen zu Fachlogik Komponenten</i>	22
2.3.4 <i>Interceptors</i>	22
2.4 CLIENTS, BCI (BUSINESS COMMON INTERFACE)	23
2.4.1 <i>Test Klassen</i>	24
2.4.1.1 <i>Verwendung von Test Daten aus externen Dateien</i>	25
2.5 MANAGER KLASSEN	26
2.5.1 <i>Manager Interfaces</i>	28
2.6 ENTITY BEANS	29
2.6.1 <i>Relationen</i>	33
2.6.1.1 <i>Relation „OneToMany“</i>	33
2.6.1.2 <i>Relation „ManyToOne“</i>	34
2.6.1.3 <i>Relation „OneToOne“</i>	35
2.6.1.4 <i>Relation „ManyToOne“</i>	35
2.7 MESSAGE DRIVEN-BEAN	38
2.8 SERVICES	39
2.8.1 <i>Timer-Service</i>	39
2.8.2 <i>Web-Services</i>	39
2.8.3 <i>Callback Klasse</i>	39
2.9 FREE KLASSE	40
2.10 GENERIERUNG VON DATENBANK UND SQL SKRIPTE	40
2.11 REGEL FÜR DIE GENERIERUNG VON NAMEN	41
2.12 LOG DATEIEN BESCHREIBUNG	42
2.13 MESSAGE DRIVEN BEAN	43
2.14 SERVICES	43
2.14.1 <i>Timer Service</i>	44
2.14.2 <i>Web Services</i>	44
2.14.3 <i>Callback Klasse</i>	44
2.15 FREE KLASSE	45
3. VERGLEICH ZWISCHEN TLGEN UND ANDEREN CODE GENERATOREN	46
4. TLGEN GENERIERUNG	48
4.1 WAS WIRD GENERIERT	48
4.2 WIE WIRD CODE GENERIERT?	48
4.3 ORDNERSTRUKTUR FÜR DEN GENERIERTEN CODE	48
4.4 DEFAULT KONFIGURATIONS DATEI	49
4.4.1 <i>Default Konfigurations-Datei - Struktur</i>	50
4.4.2 <i>Default Konfigurations-Datei - Beschreibung</i>	51
4.4.2.1 <i><Standard> Tag</i>	51
4.4.2.2 <i><Project> Tag</i>	51

4.4.2.3	<Locator> Tag.....	52
4.4.2.4	<Class> Tag	52
4.4.2.5	Entity> Tag	52
4.4.2.6	<Manager> Tag	53
4.4.2.6.1	Fields für die Manager-Beans	54
4.4.2.6.2	Methoden für Manager	54
4.4.2.7	<Session> Tag.....	55
4.4.2.7.1	Methods für Session	55
4.4.2.7.2	<Client>.....	55
4.4.2.7.3	<Xml-persistence>	56
4.4.2.7.4	<Interceptor>	56
4.4.2.8	<Messagedriven> Tag.....	56
4.4.2.9	<UmlControl> Tag	56
4.4.2.10	<Method> tag	56
4.4.2.11	<Variable> Tag	57
4.5	PROJEKT KONFIGURATIONSDATEI	58
4.5.1	<i>Projekt Konfiguration Datei - Struktur</i>	58
4.5.2	<i>Projekt Konfigurationsdatei - Beschreibung</i>	59
4.5.2.1	<Generator> Tag	60
4.5.2.1.1	Regel für Namen-Transformation in Klassen, Methoden und Tabellen.....	61
4.5.2.2	<Project> Tag	61
4.5.2.2.1	Code Format.....	63
4.5.2.3	<Locator> Tag.....	64
4.5.2.4	<Design> Tag.....	64
4.5.2.4.1	<Class>-Tag	64
4.5.2.4.2	<Entity> Tag.....	64
4.5.2.4.3	<Manager>.....	65
4.5.2.4.4	<Session> Tag.....	71
4.5.2.4.5	<Messagedriven> Tag	75
4.5.2.4.6	<Test> Tag	77
4.5.2.4.7	<Dbtable> Tag	77
5	INSTALLATION VON TLGEN	79
A.	VERWENDETE TOOLS UND EXTERNE PROGRAMME	80
B.	BEISPIEL CODE	80
i.	<i>Standard Konfiguration-Datei</i>	80
ii.	<i>Konfiguration-Datei</i>	81
iii.	<i>Klasse und Interfaces</i>	84
iv.	<i>Session Bean</i>	87
v.	<i>Manager Bean</i>	90
vi.	<i>Entity Bean</i>	95
vii.	<i>Interceptor</i>	97
viii.	<i>Timerservice</i>	98
ix.	<i>Webservice</i>	99
x.	<i>Message Driven Bean</i>	100
6.	LITERATURHINWEIS	102

Abbildungsverzeichnis

ABBILDUNG 1: IT PROJEKT-ARCHITEKTUR.....	6
ABBILDUNG 2: TLGEN ARCHITEKTUR-KONZEPT.....	9
ABBILDUNG 3: TLGEN-GENERIERUNGSPROZESS VOR EINER DATENBANK	12
ABBILDUNG 4: UML-KLASSENDIAGRAMM I	14
ABBILDUNG 5: UML-KLASSENDIAGRAMM II	14
ABBILDUNG 6: UML-KLASSENDIAGRAMM III	15
ABBILDUNG 7: UML-KLASSENDIAGRAMM IV	15
ABBILDUNG 8: „ONETOMANY“.....	34
ABBILDUNG 9: „ONETOMANY“ DATENMODELL.....	34
ABBILDUNG 10: „MANYTOMANY“ DARSTELLUNG IM UML DOMAINMODELL.....	36
ABBILDUNG 11: „MANYTOMANY“ DB RELATION	36

Tabellenverzeichnis

TABELLE 1: JAVA-TYPEN	17
TABELLE 2: ANNOTATIONEN	19
TABELLE 3: AUTOMATISCH GENERIERBARE DEFAULT METHODEN	26
TABELLE 4: ANNOTATIONEN FÜR MANAGER KLASSEN.....	28
TABELLE 5: ANNOTATIONEN FÜR ENTITY BEANS (EJB3)	30
TABELLE 6: NAMEN-REGEL.....	41
TABELLE 7: ANNOTATIONEN FÜR MESSAGE DRIVEN BEAN	43

1. Allgemeine Information

„Wissen ist Macht“ hat der englische Philosoph Francis Bacon im sechzehnten Jahrhundert gesagt. Um sich Wissen anzueignen, braucht man Daten.

Daten sind darstellbare Elemente einer Information, mit deren Hilfe Eigenschaften einer Aktivität beschrieben und in Systemen verarbeitet werden. Heutzutage entsprechen diese Systeme den Computern, die in einer digitalen Form Daten verarbeiten.

Durch die stetig ansteigende Computerleistung in Verbindung mit Datenbanksystemen und Vernetzung (z.B. durch Internet) sind wir begraben unter einer Flut an Daten, die Systeme regelrecht zusammenbrechen lassen.

Wir benötigen für den täglichen Tagesablauf eine Fülle an Informationen (z.B. Banküberweisungen, Emailverkehr, CRM/ERP-Systeme), die wir immer öfter nicht mehr in einer effizienten Form erhalten (z.B. Systemabstürze, Unterbrechungen, fehlende Daten). Ein Hauptgrund dieser Problematik findet sich in den Verfahren mit dem die Daten gesucht sowie abgespeichert werden.

Das heutige Standardverfahren für die Datenverwaltung der Informationen bevorzugt den Einsatz von *relationalen Datenbanken* (z. b. Oracle, DB2 von IBM, MySQL, SQL Server von Microsoft etc.) sowie von *Applikation Servern* bevorzugt (z. b. JBoss, WebLogic von Oracle, WebSphere von IBM etc.).

Die zwei Komponenten - Datenbanken und Application Server -, werden in einer Persistenzschicht integriert, welche Teil eines Programms ist, das die Fachlogik darstellt, notwendig für eine bestimmte Aktivität (siehe Abbildung 1).

Der Nutzer (User) verwendet ein Programm über die Bedienoberfläche (GUI oder Web), die sich meist auf einem getrennten Computer, dem Client, befindet. Der Client holt sich die Informationen, die er benötigt, von einem Server, wo sich die Fachlogik des Programms sowie die Datenbank befinden. Nach heutigem technischem Stand wird die Steuerung der Kommunikation zwischen Client und Server durch einen Application Server gesteuert, der gleichzeitig die Daten innerhalb der Datenbank mit Hilfe von so genannten CRUDs (Create, Read, Update, Delete) verwaltet.

Die Persistenzschicht ist ein essentiell wichtiger Teil eines jeden IT Programms und verursacht den Hauptanteil der Programmentwicklungskosten zusammen mit Wartung und Erweiterungen durch neue Features. Der Kostenwand wird durch den Einsatz von relationalen Datenbanken weiter ausgebaut.

Zurzeit sind die wichtigsten Gründe für erhöhte Kosten im IT Sektor:

- Falsche Modellierung von Daten-Modellen sowie die Anwendung von Daten-Modellen, die nicht genormt sind (wichtig ist die dritte Normalform)
- Daten-Modelle, die Historisch gewachsen sind
- Die Persistenzschicht ist nicht optimal gestaltet worden und benötigt einen zu großen Aufwand an Wartung und Weiterentwicklung
- Mischung von technischem mit fachlichem Code (siehe Anmerkung)

Anmerkung:

In der Softwareentwicklung sind drei Typen von Code zu unterscheiden:

- **Technischer Code:** Code, der zu 100 % unabhängig von der Fachlogik ist.
- **Generierbarer Code:** Ist gemischter Code, der aus technischem und fachlogischem Code besteht (nur Datenstruktur). Dieser Code Typ lässt sich zu 100 % generieren und somit die Entwicklung-, Wartung- sowie Weiterentwicklungskosten von IT Projekten enorm reduzieren.

- In Abb. 1 entspricht dieser Code Typ den Session Beans, der Persistence Data (Objects/Interfaces), den Managern, der Entity Beans, BCI, JUnit (Test) und den Interfaces der Business Tier. Das umfasst 40 – 80 % der Entwicklungskosten eines IT Projekt (siehe 1.2.1).
- **Fachlogik-Code:** Ist zu 100 % spezifisch für jedes einzelne IT Projekt. Zu diesem Typ gehören z.B. die Gestaltung der Oberfläche (Präsentation Tier) oder die Kommunikationsschnittstellen für Partnersysteme etc.

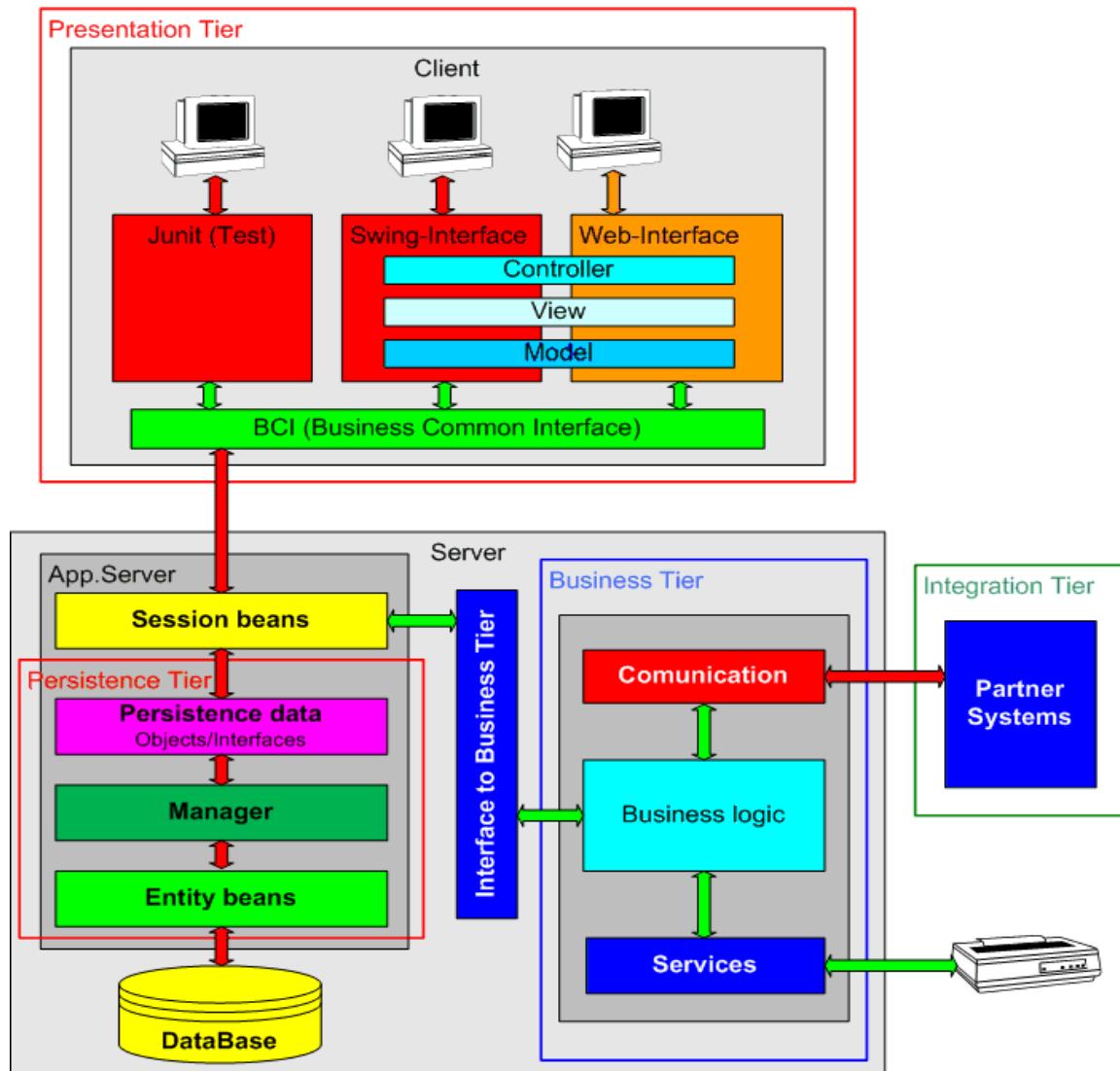


Abbildung 1: IT Projekt-Architektur

TLGen ist ein Generator für den gemischten Java Code auf Basis von EJB 3, eine Thematik, die detailliert in Kapitel 1 erläutert wird (siehe auch www.tlgen.com).

In Kapitel 2 wird die Verwendung von TLGen mit der genauen Offenlegung seiner Features in Kombination mit der Ordnerstruktur der Konfigurationsdatei, woraus man die Projektarchitektur gestalten kann, beschrieben.

In Kapitel 3 werden Beispiele zur Code-Generierung mit Hilfe der Konfigurationsdatei genannt.

Kapitel 4 behandelt die Möglichkeiten, die TLGen ermöglicht, um z.B. die Datenmodelle zu optimieren oder Kreise in Datenmodelle zu vermeiden. Gleichzeitig werden die Logdateien beschrieben, die bei der Generierung entstehen.

Im Kapitel 5 wird die Installation und Verwendung von TLGen in IT Projekte beschrieben.

1.1 Ziele

TLGen ist ein Code-Generator auf Basis von EJB3 mit Java Annotationen. Ziel unserer Arbeit war, die Kosten für die Backend-Softwareentwicklung durch eine komplette Generierung von Code zu minimieren.

Aktuelle Code-Generatoren auf Basis des MDA Ansatzes (Hibernate, TopLink etc.) benötigen nach der Generierung auch einen bedeutenden Arbeitsaufwand durch die Entwickler, nachträgliches Customizing, so dass der angesetzte Zeitgewinn durch weiteren Aufwand verfällt. Der Mix aus generiertem Code und den von Programmierern entwickelten Code (manuelle Entwicklung) führen in der Realität zu einer Anzahl von Problemen (unterschiedliche Logik, Layout, Auseinanderdriften von Modell und Code, Fehler etc.).

Dies ist der Grund, warum der Ansatz von TLGen eine klare Trennung zwischen dem generierten und manuell entwickelten Code fordert. In der IT wurde dies bewiesen, dass sich ein sehr großer Teil von Code vollständig generieren lässt. TLGen ermöglicht diese Generierung mit Hilfe des Daten-Inputs in Form eines Domainmodells (neue Projekte) oder einer schon existierenden Datenbank (Refactoring Projekte von Legacy Systemen) in Verbindung mit zwei Konfigurationsdateien (beinhalten die Steuerungsparameter für die Generierung).

TLGen benötigt zwei Konfigurationsdateien: eine für allgemeine Definitionen, wie Daten-Typen, Konvertierung, etc., die nur selten zu ändern ist (wird mit TLGen mitgeliefert) und eine Projekt-Konfigurationsdatei, die wichtige Informationen beinhaltet, wie Namen, Regeln, Umwandlungen, Projekt-Struktur (wichtig für Legacy-Projekte), Verknüpfung von Tabellen zu bestimmten Session Beans, usw.

Zusätzlich bietet TLGen die Möglichkeit an eine Datenbank oder ein Domainmodell zu analysieren und unterbreitet Vorschläge für deren Optimierung.

Ist die Entwicklungsphase eines IT-Projektes beendet, werden stetig weitere Features für neue Praxis-Anforderungen implementiert und somit die benötigte Datenbank durch Erweiterung angepasst (gemeint ist nur die Datenstruktur). Diese historisch wachsenden Datenbanken führen eher selten zu einer Fortführung optimaler Datenstrukturen und letztendlich zu hohen Wartungskosten in Verbindung mit dem Zwangsstart von Refactoring Projekten.

Das Datenmodell sollte, mindestens in der dritten Normalform vorliegen (siehe z.B. http://de.wikipedia.org/wiki/Normalisierung_%28Datenbank%29). Die Normalisierung ist ein wichtiges Hilfsmittel, um die Konsistenz der Daten zu gewährleisten. Das erleichtert die Wartung sowie die Upgrade-Möglichkeiten für zukünftige Anforderungen der Applikation, ohne die Datensätze zu verändern.

Da es für viele Datenbanken keine grafische Darstellung der Datenstruktur gibt, um so die Möglichkeit der Optimierung des Datenmodells zu gewährleisten, wird die Arbeit der Erstellung eines neuen Modells erheblich erschwert. TLGen bietet jetzt die Möglichkeit an aus einem existierenden Datenbankmodell die Grafik bzw. das Domainmodell zu generieren, zusammen mit Optimierungsvorschlägen. Durch die schnelle Code-Generierung können diese Änderungen sofort für die obligatorischen Projekt-Tests bereitstehen und somit einen schnellen Überblick über das Entwicklungsvorhaben ermöglichen.

Diese grafische Darstellung als Domainmodell (UML) ist sehr wichtig für die Optimierung einer alten Datenbank.

Des Weiteren wird erst durch das neu-erstellte Domainmodell eine Generierung des Backend's möglich, da jede Zeile aus einer Datenbanktabelle als Java Bean-Objekt im generierten Code abgebildet wird (dabei spielt es keine Rolle, ob als Join/Mapping aus mehreren Tabellen oder aus einer Zeile einer Tabelle).

Der generierte Backend-Code entspricht dem Software-Architekturkonzept, welches in Abb. 2 anschaulich erklärt wird.

1.2 TLGen's Architektur-Konzept

In Abbildung 2 wird das Architektur-Konzept von TLGen dargestellt, zusammen mit den Komponenten, welche TLGen generiert:

- Test Klassen(JUnit),
- Data Klassen/Interfaces
- Sessions Beans
- Manager Beans
- Entity Beans
- Message Driven Beans
- Timer Services
- Web Services
- Callback Klassen
- Free Klassen
- Log Dateien (beschreiben im Detail die generierte Elemente)
- Datenbank-Skripte (notwendig, um eine Datenbank zu erstellen).

Für das Ändern einer schon vorhandenen Datenbank werden nur die Differenz-Skripte generiert, um so die existierenden Daten nicht zu löschen.

Für die Änderung einer existierenden Datenbank werden nur die Differenz Skripte generiert, um so die vorhandenen Daten nicht zu löschen. Diese Skripte können direkt in dem Generierungsprozess verwendet werden oder erst später, bei Bedarf (siehe 1.3 und 2).

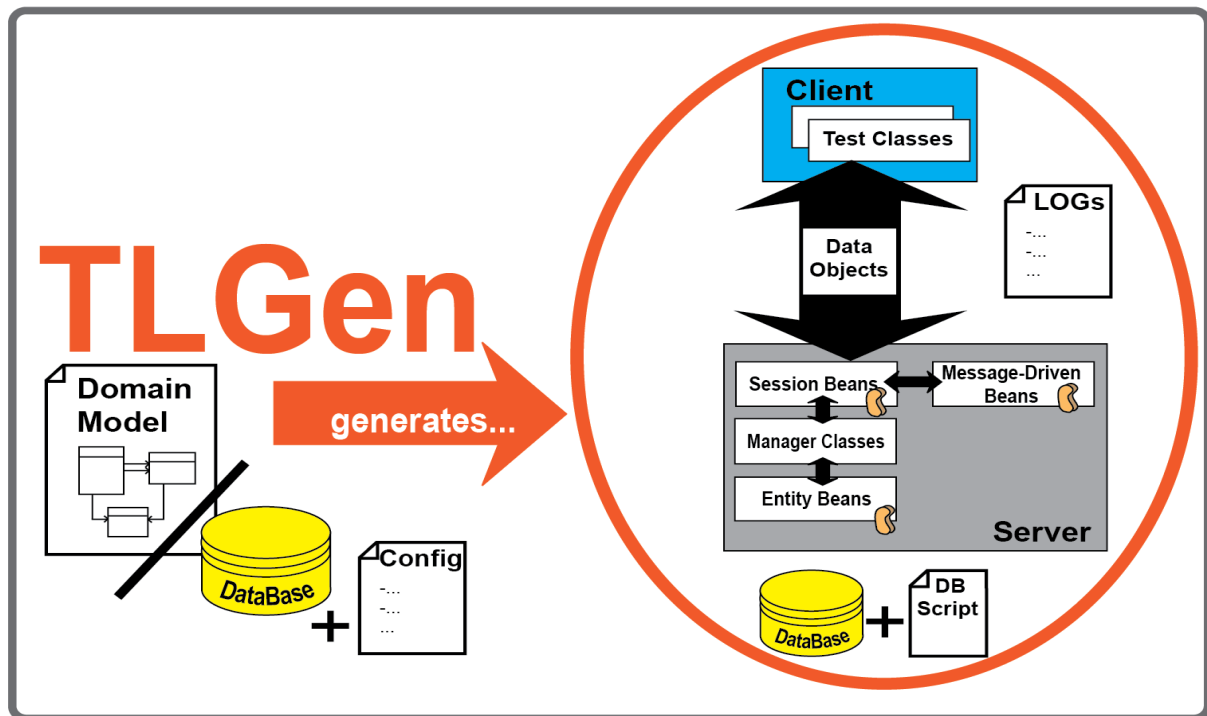


Abbildung 2: TLGen Architektur-Konzept

1.2.1 Input für die Code-Generierung

Damit TLGen den Code generieren kann, wird folgender Input benötigt:

- Ein Datenstruktur-Modell, was entweder ein *Domainmodell* (siehe Abb. 2) sein kann, designed in UML für den Start neuer IT-Projekte (siehe Kapitel 1.3.1) oder ein Datenbankmodell (z. B. eine existierende Datenbank – Abb. 2) beim Einsatz in Legacy-Projekten (siehe Kapitel 1.3.2), die einen Refactoring Prozess benötigen.
- Ein Default Konfigurationsdatei (im XML-Format), welche die Werte für Definitionen beinhaltet, wie z.B. allgemeine Regeln für die Namen-Konventionen, UML Domainmodell Regel, Standard-Methoden für die Datenbankzugriffe, eine Tabelle mit Konvertierung zwischen den Variablen-Typen in Java Code und in der Datenbank.
- Ein Konfigurations-Datei (im XML-Format) mit unterschiedlichen Informationen, die vom Projekt abhängen (siehe Kap. 3).

Oben genannte Input-Informationen werden in Projekten immer von den Designern oder Datenmodellieren erstellt. Lediglich die Konfigurationsdatei ist neu für den Code, denn das Daten- oder Datenbankmodell (DM oder DB) ist immer eine Notwendigkeit in der Entwicklung von IT-Projekten.

1.3 Beschreibung der Möglichkeiten durch TLGen

Im fortlaufenden Kapitel werden alle detaillierten Möglichkeiten beschrieben, die TLGen für die Codegenerierung sowie die Analyse von Datenbanken bietet. Die detaillierte Struktur der Konfigurationsdateien wird in Kap. 4.4 erläutert.

Wie schon im Kapitel 1.2 erwähnt, steuert TLGen den Generierungs-Prozess mit Hilfe von zwei Konfigurations-Dateien, eine für Default-Werte, die in der Regel keinen großen Änderungsbedarf besitzt sowie eine Konfigurations-Datei, die individuell für jedes Projekt ist.

TLGen bietet für die Generierung von Code folgende Möglichkeiten an:

- Das Einbauen von Kommentaren innerhalb des generierten Codes (siehe Kap. 3.1)
- Regel für die Namen-Generierung der Datenbanken sowie für den generierten Code (siehe Kap. 3.3)
- Der Einsatz aller Annotationen im generierten Code, die EJB3 anbietet
- Informationen für die Generierung mit Hilfe von Apache Tool „ant“ und einer „ear“ Datei
- Der generierte Code entspricht dem EJB 3-Standard und ist verwendbar auf allen Applicationen Server, die momentan am Markt vertreten sind (WebLogic, WebSphere, JBoss, GlasFish, IoNas, etc.)
- Folgender Code wird generiert:
 - Session Beans mit allen Annotationen (siehe Kap. 2.3)
 - Manager Klassen für die Steuerung von Entity Beans mit allen möglichen Zugriffen für CRUD (Create, Read, Update, Delete). Beim Manager ist es auch möglich eigene SQLs einzubinden (werden aus den Konfigurationsdateien gelesen) (siehe Kap. 2.5).
 - Entity Beans, wo alle Relationen automatisch generiert werden (von Domainmodell oder aus der Datenbank) - OneToOne, OneToMany, ManyToOne, ManyToMany (siehe Kap. 2.6 und 2.7)
 - Clients Klassen für BCI (Business Common Interfaces); diese ermöglichen die Verbindung zwischen Client und Server (Session Beans).
 - Test Klassen auf Basis von JUnit. Die Daten für Test Klassen können durch TLGen generiert oder von einer XML Datei geladen werden. Auch die Struktur von den XML-Daten-Dateien kann von TLGen generiert werden. Test-Klassen sind in zwei Modi zu verwenden, indem die Daten direkt geprintet oder über die Mechanismen von JUnit getestet werden.
 - Datenbank-Skripte
 - Klassen für die Interfaces für Fachlogik Code
 - Interceptoren
 - Daten Klassen/Interfaces
 - XML Datei für Persistenz (persistence.xml), eine für jede Session Bean

TLGen kann mit den „ant“ Apache Tool genutzt werden, da TLGen einen eigenen Tag „generator“ besitzt.

1.3.1 Code Generierung mit einem Domain Model (UML)

Die Generierung mit Hilfe eines Domainmodells als Informations-Input findet seine Anwendung bei neuen IT-Projekten. Ein IT-Architekt (oder ein Architektur-Team - die Wahl hängt vom Projekt ab) designend zusammen mit der Fachabteilung ein Domainmodell, das die Datenstruktur und dessen Verbindungen für das Programm darstellt. Auf dieser Basis kann man letztendlich mit TLGen den kompletten Code generieren und diesen sofort testen.

Für solch ein Projekt bietet TLGen zu den Eigenschaften aus Kapitel 1.3 zusätzliche Möglichkeiten an:

- Eine Analyse des Domainmodells nach falschen Namen (z. B. zu lang für eine Datenbank), eventuelle Kreise in den Verbindungen zwischen Objekten (Tabellen) und mögliche Verbesserungsvorschläge (siehe Kap. 2.1).
- Innerhalb von einem Domainmodell können Klassen-Typen und Enume verwendet werden. Daten und Klassen-Typen können in jeder beliebigen Art und Weise verschachtelt werden, so wie die benötigte Logik der Applikation.

1.3.2 Code Generierung mit einer schon vorhandenen Datenbank

Nachdem TLGen einen Zugang zur Datenbank erhalten hat, bezieht TLGen alle relevanten Information für die Code-Genierung, wie z.B. Tabellen, Spalten, Relationen, Sequenzen etc. TLGen ist kompatibel zu allen momentan Datenbanken.

Für ein Projekt, dass mit einer Datenbank als Input startet (z.B. Refactoring von Legacy-Systemen) bietet TLGen zu den in Kap. 1.3 aufgeführten Eigenschaften weitere Möglichkeiten an, wie:

- Regel für die Verwendung von Tabellen, Spalten etc. für den generierten Code, konform den existierenden Namen-Konventionen im Code
- Generierung von Datenbank-Skripte für eine neue Datenbank oder nur Änderungs-skripte, die dann nur die Differenzen beinhalten

2. TLGen Code-Generierung und die Analyse von Datenbanken

TLGen ist ein einfaches Tools mit höchstem Grad an technischen Hintergrund für die Generierung von Backend Code auf Basis von EJB3. Wie in den Kapiteln zuvor erklärt, generiert TLGen den Code mit einem Domain- oder Datenbankmodell (siehe Abb. 3) und bietet gleichzeitig Optimierungsmöglichkeiten nach eigener Analyse der Modelle an.

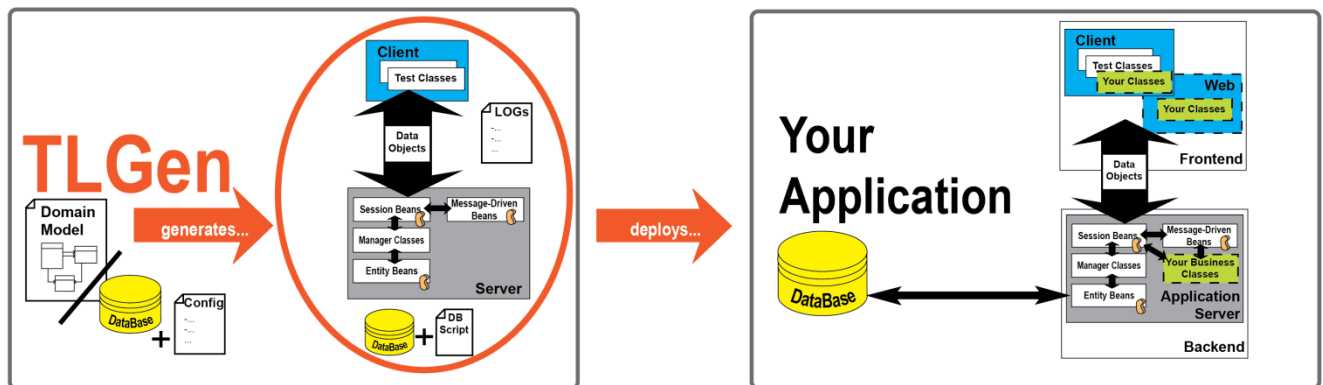


Abbildung 3: TLGen-Generierungsprozess vor einer Datenbank

In diesem Kapitel werden detailliert die generierten Objekte sowie zur Verfügung stehende Steuerungsmöglichkeiten beschrieben.

Allgemeine Features, die zentral für die gesamte Generierung möglich sind (für Details siehe Kap. 4.5):

- Projekt-Name
- Application Server, z.B. „JBoss“, „Weblogic“ etc. Der Application Server ist wichtig für die Verwendung des generierten Codes für „Clients“, weil jeder App.Server verwendet für die Variable „datasource“ eigene Darstellung und für die Connection zwischen Client und Server eigene Klassen, z.B. für JBoss folgende Klassen verwendet:

```
initialcontextfactory="org.jnp.interfaces.NamingContextFactory"
initialcontextpkgprefix="org.jboss.naming:org.jnp.interfaces"
```

1. Listing

- Datenbank-Namen mit Zugriff für die berechtigten User samt Passwörter zum Auslesen der Informationen (Tabellen, Spalten, Sequences, etc., für Legacy Projekte oder Ziele, wo die Tabellen generiert werden sollten)
- Projekt-Steuerungs-Attribut. Ein Refactoring-Projekt durch eine schon vorhandene Datenbank (wichtig bei vorhandener Legacy-IT) oder Entwicklung einer neuen Applikation durch ein Domainmodell. Für die Generierung einer neuen Datenbank kann diese vollständig neu erstellt oder nur die Änderungen übernommen werden ohne schon vorhandene Daten zu löschen.
- Notwendige Informationen für die Generierung von SQL Skripte wie Tablespace etc.
- „ear“ Datei-Name
- Die generierte Code-Formatierung kann frei gewählt werden (Standard Java-Formatierung z.B. mit geschweiften Klammern in derselben oder nächsten Zeile oder

ob z.B. die Import-Deklarationen am Klassenanfang oder direkt im Code mit den vollständigen Import-Pfaden angezeigt wird.)

```
import eu.stardata.core.hlp.dataif.CoLanguageDataIf;
import javax.persistence.OneToOne;

class
{
    private CoLanguageDataIf ...
    ...
}
```

2. Listing

oder ohne Import in Code:

```
class
{
    @ javax.persistence.OneToOne(cascade = {
    javax.persistence.CascadeType.PERSIST}, targetEntity =
    eu.stardata.server.core.formular.entity.PageEntity.class)
    private eu.stardata.server.core.doclist.entity.DocumentlistEntity ...
    ...
}
```

3. Listing

2.1 Analyse des Domainmodells und der Datenbank

Die Relationen von Klassen eines Domainmodells oder von Datenbank-Tabellen können je nach Anforderungen unterschiedlich komplex sein.

Betrachten wir als Beispiel Abbildung 4 eines UML-Klassendiagramms, wo beim Neuanlegen der Klasse C mindestens ein B über zwei verschiedene Pfade gleichzeitig benötigt wird. Doch, wenn noch kein B existiert, muss dieses erst neuangelegt werden. Dies ist in diesem Fall aber nur möglich, falls ein C über eine Relation von Pfad A schon existiert. Das Klassendiagramm ist logisch nicht inkorrekt (z.B. sei A ein Produkt, B ein Element des Produkts und C ein Bestellauftrag). Aber die Klassen des Beispiel Domainmodells, jeweils als Tabelle in einer Datenbank abgebildet, führen zum Problem, dass C B direkt benötigt (*not null*) genauso wie A (*not null*) doch A benötigt und auch ein B (*not null*), dass zu diesem Zeitpunkt schon existieren muss.

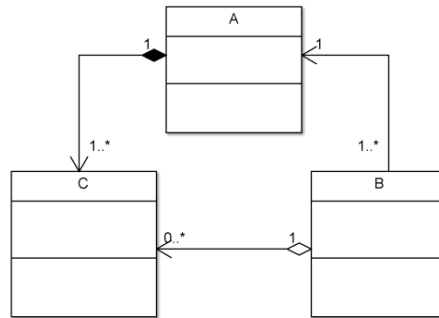


Abbildung 4: UML-Klassendiagramm I

Anmerkung: Ein UML-Klassendiagramm, wo beim Neuanlegen der Klasse C B über 2 verschiedene Pfade benötigt wird. Beim Neuanlegen von B kann dieses dagegen noch nicht existieren, da C noch nicht über die Relation von Pfad A existiert.

Kreise, durch Pfade in der Erreichbarkeit der Klasse „C“, sind hier folgende:

Die Klasse „C“ ist über die Klasse „B“ durch folgende Pfade erreichbar:

- $[B (NOT NULL) \rightarrow A (NOT NULL)] \rightarrow [C]$
- $[B (NOT NULL)] \rightarrow [C]$

Genau in diesem Schema werden alle Pfade in der Erreichbarkeit einer Klasse (hier „C“), über welche Klassen (hier „B“) diese Pfade verlaufen, in der Debug-Log Datei bei der Generierung des Codes durch TLGen dargestellt. Fälle wie in Abbildung 4 werden mit einem „Fatal Error“ in der *Debug-LOG* Datei und *Error-LOG* Datei ausgegeben. Dagegen werden Kreise von Relationspfaden, die nicht über unterschiedliche Klassen in der Zielklasse enden, mit einem „Warning“ angezeigt (vgl. Abbildung 5).

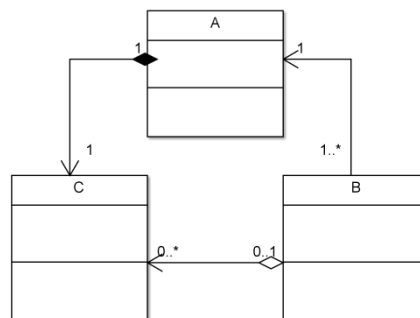


Abbildung 5: UML-Klassendiagramm II

Wie Abb. 1, aber in der Relation von $[B(Null)] \rightarrow [C]$ ist *null* erlaubt. Somit kann B und C über den Relationen von A gleichzeitig angelegt werden.

Direkte Kreise, d.h. Zielklasse ist auch die Startklasse in einem Relationspfad, werden in den *LOG*-Dateien als „Fatal Error“ deklariert (vgl. Abbildung 6), falls alle Klassen im Relationspfad voneinander abhängig sind (hier: $[A (NOT NULL) \rightarrow C (NOT NULL) \rightarrow B (NOT NULL)] \rightarrow [A]$) und nur mit einem „Warning“, falls dies nicht zutrifft (vgl. Abbildung 7: $[A (NOT NULL) \rightarrow C (NULL) \rightarrow B (NOT NULL)] \rightarrow [A]$)

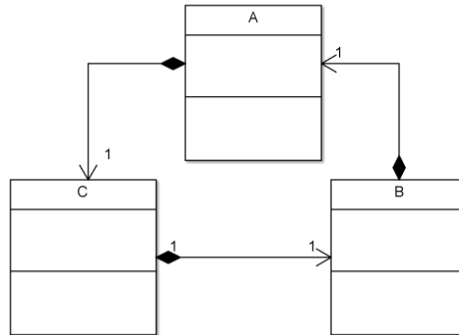


Abbildung 6: UML-Klassendiagramm III

Ein direkter Relationspfad, wo jede Startklasse auch Zielklasse ist und alle voneinander abhängig sind.

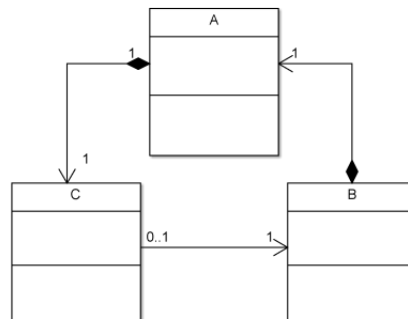


Abbildung 7: UML-Klassendiagramm IV

Ein direkter Relationspfad, wo jede Startklasse auch Zielklasse ist, aber nicht alle voneinander abhängig sind.

Im Folgenden ist ein Knoten äquivalent mit einer Klasse des Domainmodells oder einer Datenbanktabelle.

Die *Debug-LOG* Datei beinhaltet folgende Informationen, um Pfade zu analysieren:

- **„CHECK RELATIONS METHODS - Adjacency list“:**
Die Liste zeigt alle Knoten (TARGET), die über verschiedene Relationspfade erreicht werden können:

Node: 'TARGET':

[ungeordneter Relationspfad] ---> [TARGET] [alle passierten Vorknoten der Pfade aufsummiert]

Beispiel: [C,B] ---> [A] [C(1),B(2)]

- **„CHECK RELATIONS METHODS - CIRCLE PATH (direct)“:**
Die Liste zeigt alle Kreise der Knoten (TARGET), die über verschiedene Relationspfade über den TARGET selber erreicht werden können:

Node: 'TARGET':

[geordneter Relationspfad beginnend mit TARGET] ---> [TARGET]

Beispiel: [A(NULL)->C (NOT NULL)->B (NOT NULL)] ---> [A]

- **„CHECK RELATIONS METHODS – ALL PATHS“:**
Die Liste zeigt alle Pfade der Knoten (TARGET), die über andere Knoten erreicht werden können:

Node: 'TARGET':

[geordneter Relationspfad] ---> [TARGET]

Beispiel: [C (NOT NULL)->B (NOT NULL)] ---> [A]

Wenn neben dem Pfad ein „FATAL“ steht, dann ist TARGET von sich selber abhängig.

- **„CHECK RELATIONS METHODS - CIRCLE PATHS (not direct)“:**
Die Liste zeigt alle Pfade der Knoten (TARGET), die von verschiedenen Knoten (SOURCE) über unterschiedliche Relationspfade erreicht werden können:

'TARGET':

'SOURCE':

[geordneter Relationspfad mit SOURCE] ---> [TARGET]

Beispiel: -'C': [C (NOT NULL)->B (NOT NULL)] ---> [A]

Wenn neben dem Knoten SOURCE „FATAL“ erscheint, dann sind mehrere Pfade, die in TARGET enden, von SOURCE abhängig (siehe oben, z.B. [C->D]--->[A] und [C->B]--->[A]). Wenn ein „Warning“ erscheint, dann sind mehrere Pfade, die in TARGET enden, von SOURCE abhängig, wobei diese über denselben Knoten in TARGET (siehe oben, z.B. [C->D->B]--->[A] und [C->B]--->[A]) enden.

Die Error-LOG Datei beinhaltet folgende Informationen, um Pfade zu analysieren:

- **„Direct circles in relations“**
 - „**Warning**“, wenn es einen Relationspfad gibt, wo Start- und Endknoten derselbe ist.
Beispiel: [A (NULL)->B (NOT NULL)] ---> [A]
 - „**FATAL Error**“, wenn es einen Relationspfad gibt, wo Start- und Endknoten derselbe ist und voneinander abhängig sind.
Beispiel: [A (NOT NULL)->B (NOT NULL)] ---> [A]
- **„Relation Problems?“**
 - „**Warning**“, wenn man den Zielknoten über verschiedene Relationspfade desselben Startknotens erreichen kann und diese voneinander abhängig sind, aber in den Zielknoten immer über denselben Vorgängerknoten endet.
Beispiel: [C (NOT NULL)->B (NOT NULL)] ---> [A] und [C (NOT NULL)->D (NOT NULL)->B (NOT NULL)] ---> [A]
 - „**FATAL Error**“, wenn man den Zielknoten über verschiedene Relationspfade desselben Startknotens erreichen kann und diese voneinander abhängig sind.
Beispiel: [C (NOT NULL)->B (NOT NULL)] ---> [A] und [C (NOT NULL)->D (NOT NULL)] ---> [A]

2.2 Daten Klassen - Interfaces

TLGen kann Daten, Klassen/Interfaces aus einem Domainmodell oder aus einer vorhandenen Datenbank generieren. Es wandelt die Datenbank-Tabellen mit seinen Spalten in Klassen/Interfaces mit ihren Variablen um. Die Daten-Typen von Spalten umwandelt es in Java Daten-Typen wie in Tabelle 1 dargestellt. Für Sonderwünsche der Typ-Verwendung kann in der Default Konfiguration Datei eigene Umwandlung definiert werden (siehe 4.4).

Tabelle 1: Java-Typen

Nr.	Datenbank Type	Java (Code) Type
1	CHAR	char
2	NCHAR	char
3	DATE	java.util.Date
4	TIMESTAMP	java.util.Date
5	DECIMAL	double
6	BINARY_DOUBLE	double
7	BINARY_FLOAT	float
8	INTEGER	int
9	NUMBER	long
10	VARCHAR	java.lang.String
11	VARCHAR2	java.lang.String
12	NVARCHAR2	java.lang.String
13	LONG	java.lang.Long
14	RAW	java.lang.String
15	LONG RAW	byte[]
16	CLOB	java.lang.String
17	NCLOB	java.lang.String
18	BLOB	byte[]
19	TEXT	java.lang.String
20	LONGTEXT	java.lang.String
21	INT	int
22	BIGINT	long
23	DATETIME	java.util.Date
24	SMALLINT	short

In Konfigurationsdateien ist es möglich auch die Klassen/Interface Path mit „extends“ zu definieren. Diese „extends“ sind Basis Klassen/Interfaces, die eventuelle Variablen beinhalten, die die komplette Applikation beinhaltet und kann von TLGen Anwender geschrieben werden. TLGen liefert auch diesen Code für Standard Anwendungen mit. Diese Superklassen sind nicht unbedingt notwendig für das Programm.

TLGen hat auch eigene Basis Klassen/Interfaces, die als Default verwendet werden, falls der TLGen Anwender nicht eigene schreibt (siehe 4.4.2.1). In Konfigurationsdateien werden auch das Path für Klassen „Typen“ und „Enums“ definiert, falls diese in der Applikation verwendet werden.

TLGen generiert für die Relationen, die in der Datenbank- oder in dem Domainmodell vorhanden sind entsprechende Variablen als Liste oder einfache Variablen wie für Interfaces:

```
public abstract List<CoPageDataIf> getPage();
public abstract void setPage(List<CoPageDataIf> arg);
```

4. Listing

und für Klassen:

```
private List<CoPageDataIf> m_page;

public List<CoPageDataIf> getPage() {
    return m_page;
}

public void setPage(List<CoPageDataIf> arg) {
    m_page = arg;
}
```

5. Listing

Für die Klassen/Interfaces, die seitens Domainmodells generiert sind, werden alle Attribute übernommen, persistente und nicht persistente, aber in den SQL Skripten (die für die Datenbank Generierung notwendig sind) werden nur die persistenten Variablen verwendet.

Für die Klassen/Interfaces, die von einer vorhandenen Datenbank generiert werden, können neue Variable eingefügt werden, die nicht persistent sind.

2.3 Session Bean

Session Bean ist eine wichtige Klasse und die erste Klasse, welche von einem Client über RMI aufgerufen wird. Sie hat innerhalb von EJB3 eine vordefinierte Form und kann mit mehreren Annotationen verwendet werden (siehe Tabelle 2).

Mit Hilfe der Steuerung über die Konfigurationsdateien können für die Session Beans folgende Features generiert werden:

- Annotationen mit Parameter
- Selbst geschriebene „extends“ Klassen oder die von TLGen Default Session Bean Basis Klassen
- Interceptoren (siehe 2.3.4)
- Clients (Siehe 2.4)
- Manager Klassen (siehe 2.5)
- Entity Beans (siehe 2.6)
- XML Persistente Datei (siehe 2.3.2)

- Definiert die Methoden, die von Client aufgerufen wird. Diese Methoden können von den Manager Klassen übernommen werden.
- Kann bestimmen, welcher Type von Transaktionen verwendet wird.
- Interface-Klassen die, die Verbindung zwischen generiertem Code und Fachcode ermöglichen.

Tabelle 2: Annotationen

Annotationen	Verwendung in TLGen 2.1	Kommentar
javax.ejb.Stateless	ja	Definiert eine Stateless Session Bean
javax.ejb.Stateful	ja	Definiert eine Stateful Session Bean
javax.ejb.EJB	ja	Definiert eine Lokale References z. B. Name eines Manger Interface (siehe 2.5)
javax.ejb.Remote	ja	Bezeichnet eine Remote Verwendung von Session Bean
javax.ejb.Local	ja	Ist für eine Lokale Verwendung
javax.ejb.ApplicationException	nein	
javax.ejb.EJBs	ja	Definiert eine von mehreren EJB Annotationen
javax.ejb.Init	nein	
javax.ejb.LocalHome	nein	Entspricht nicht der Philosophie von EJB3
javax.ejb.PostActivate	ja	Methode wird nach dem Aktivierung von Session Bean aufgerufen
javax.ejb.PrePassivate	ja	Nur für Stateful Session Bean
javax.ejb.RemoteHome	nein	Entspricht nicht der Philosophie von EJB3
javax.ejb.Remove	ja	Call Methode mit den Annotation PreDestroy (nur für Stateful Session Bean)
javax.ejb.Timeout	ja	Session Bean wird nach Ablauf von timeout verworfen
javax.ejb.TransactionAttribute	ja	Transaction Type (REQUIRED, MANDATORY, etc.)
javax.ejb.TransactionManagement	ja	Transaction management (CONTAINER or BEAN)

Für eine Session Bean kann man eine oder mehrere Manager Klassen mit seiner Entity Beans aufrufen.

Von einem Domainmodell kann für jedes Package eine Session Bean generiert werden und innerhalb der Session Bean für jedes Element-Objekt ein Tandem vom Manager, Entity Bean und Datenbank-Tabelle.

Für die Legacy Projekte (Refactoring), die auf Basis einer vorhandenen Datenbank durchgeführt werden, soll in der Konfigurationsdatei eine Liste von Tabellen, die zu einer Session Bean gehören, vorhanden sein.

Session Bean Code Beispiel:

```

package eu.stardata.server.core.formular;

import eu.stardata.server.core.formular.bci.ExecBci;
import eu.stardata.server.core.formular.bci.TablecolBci;
.....
import javax.ejb.EJB;

@Stateless(name = FormularBci.JNDI_NAME, mappedName = "efp/"+FormularBci.JNDI_NAME+"/remote")
@Remote(FormularBci.class)
public class FormularSessionBean extends BaseSessionBean implements FormularBci
{
    private static final long serialVersionUID = 196502766;

    // Session annotations and fields for local manager
    @EJB
    private EntryBci m_EntryBci;
.....
    /**
     * Session method "createEntry()"
     * @param arg
     * @return
     * @throws PersistenceException
     */
    public CoEntryDataIf createEntry(CoEntryDataIf arg) throws PersistenceException
    {
        try
        {
            return m_EntryBci.create(arg);
        } catch (Exception ex)
        {
            throw new PersistenceException(ex.getMessage(), ex, 1);
        }
    }
}

```

6. Listing

2.3.1 Interfaces für eine Session Bean

Session Bean benötigt ein Interface mit allen definierten Methoden, die von Client aufgerufen werden. Diese Interfaces für Session Bean verwenden zwei Annotationen:

@Remote für eine Remote-Verbindung vom Client sowie @Local für eine lokale Verwendung. In diesem Interface sind auch drei vordefinierte Variablen, die für EJB3 sehr wichtig sind:

- String JNDI_NAME = Wert, ist der notwendige Name für den Client-Aufruf und eindeutig für die gesamte Applikation. TLGen verwendet für diese Variable den Inhalt des kompletten Session Bean Namens, z. B.
"eu.stardata.server.core.formular.FormularSessionBean"
- String MANAGER_NAME = "managerFormular". Diese Variable ist wichtig für die Verwendung der Manager Klasse (siehe Kapitel 2.5).
- String EAR_NAME = "efp". Diese Variable bekommt den Wert aus der Konfigurations-Datei und wird in der XML Persistenz Datei verwendet (siehe Kapitel 2.3.1).

Ein Beispiel für ein Session Bean Interface als „Remote“:

```

package eu.stardata.client.core.formular.bci;

import eu.stardata.core.form.dataif.CoRuletypeDataIf;
.....

```

```

import eu.stardata.core.form.dataif.CoRulefieldDataIf;
import javax.ejb.Remote;
import java.lang.String;
import eu.stardata.core.form.dataif.CoFormularDataIf;
import eu.stardata.server.common.exception.PersistenceException;
import eu.stardata.core.form.dataif.CoExecDataIf;
@Remote
public interface FormularBci
{
    // Client interface Fields
    public final static String JNDI_NAME = "eu.stardata.server.core.formular.FormularSessionBean";
    public final static String MANAGER_NAME = "managerFormular";
    public final static String EAR_NAME = "efp";

    // Session - Client interface methods
    public void clearExec() throws PersistenceException;
    public CoFieldDataIf createField(CoFieldDataIf arg) throws PersistenceException;
    public CoFieldDataIf[] findAllField() throws PersistenceException;
    .....
    public CoElementDataIf createElement(CoElementDataIf arg) throws PersistenceException;

    public void clearWebstyle() throws PersistenceException;
}

```

7. Listing

2.3.2 XML Persistente Datei

Ab EJB 3 ist nur eine XML Datei für das Deployment Description notwendig, alle anderen Dateien der Vor-Versionen sind entweder abgeschafft oder werden als Annotation verwendet. Zusätzlich gibt es noch eine XML Datei „orm.xml“ für das Mapping für Version 3 von EJB, doch diese wird aus Gründen der Performance mit TLGen nicht genutzt. TLGen erlaubt das Mapping direkt im generierten Java Code und zwar in den Entity Beans (siehe Kapitel 2.6 sowie 4.5.3.3)

In der Datei „persistence.xml“, die für jede Session Bean generiert wird, sind automatisch folgende Informationen enthalten:

- In Tag „persistence-unit“ die Parameter **name** → ManagerName Variable und Transaction-type
- In Tag „jta-data-source“ die Data Source im Format, das der jeweilige Applikation Server benötigt (jede hat ein eigenes Format)
- Eine vollständige Liste von allen in der Applikation vorhandenen Entity Beans mit vollständig definierten Namen
- In Tag „property“ allgemeine Informationen für den Application Server

Ein Beispiel für eine generierte „persistence.xml“:

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
  <persistence-unit name = "managerFormular" transaction-type = "JTA">
    <jta-data-source>java:/efpPool</jta-data-source>
    <class>eu.stardata.server.core.user.entity.UserEntity</class>
    <class>eu.stardata.server.core.hlp.entity.OperationEntity</class>
    <class>eu.stardata.server.core.hlp.entity.FormatEntity</class>
    <class>eu.stardata.server.core.hlp.entity.KeyEntity</class>
    <class>eu.stardata.server.core.formular.entity.FieldEntity</class>

```

```

        <class>eu.stardata.server.core.hlp.entity.FieldtypeEntity</class>
.....
        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.Oracle10gDialect"/>
        </properties>
    </persistence-unit>
</persistence>

```

8. Listing

In der Datei „persistence.xml“ können einige Informationen definiert werden, die vom Applikation Server benötigt werden (in einer „properties“ Tag).

Hinweis: Für den Applikation Server „JBoss“ Version 5 sollte die Oracle Datenbank in Version 10 genutzt werden, da diese Oracle Version 11 noch nicht erkennt.

2.3.3 Interface Klassen zu Fachlogik Komponenten

Innerhalb eines Session Bean kann bzw. können eine oder mehrere Klassen definiert werden, die zu oder von anderen Systemen gehören bzw. aufgerufen werden oder für die Kodierung reiner Fachlogik benötigt werden. Wurden diese einmal generiert, lässt der Generator es zu, diese dort abzuspeichern, um den dort manuell entwickelten Code nicht zu löschen.

Nach unserer Erfahrung ist es weder produktiv noch notwendig generierten Code mit selbst entwickelte zu mischen (merging). Daher ermöglicht TLGen eine vollständige Trennung dieser Code-Arten.

2.3.4 Interceptors

Ein Interceptor ist eine einfache Java Klasse deren Methoden immer dann aufgerufen werden, wenn die Methoden der Session Bean benützt werden. Die Interceptoren sind Klassen, die eigentlich zum Beispiel unter anderem zur Überwachung einer Session Bean, zur Übernahme der Informationen-Protokollierung oder Zeitmessen gedacht sind.

Diese Java Klassen können über die Konfigurationsdateien in den generierten Code eingebettet werden. Es können entweder selbst geschriebene Klassen sein oder die von TLGen Default Interceptoren zur Verfügung gestellten.

Ein einfaches Beispiel von einer Interceptor Klasse für die Zeitmessung:

```

public class TimeFormular
{
    // Session annotations and fields for interceptor
    /**
     * Default Constructor
     */
    public TimeFormular()
    {
        super();
    }

    // Session class methods
    /**
     * Interceptor method "timeTrace()"
     * @param invocation
     * @return
     * @throws PersistenceException
     */
}

```

```

@AroundInvoke
public java.lang.Object timeTrace(InvocationContext invocation) throws PersistenceException
{
    long start = 0;
    boolean toTime = true;
    try
    {
        if(toTime)
        {
            start = System.currentTimeMillis();
        }
        return invocation.proceed();
    } catch(Exception ex)
    {
        throw new PersistenceException(ex.getMessage(), ex, 1);
    }
    finally
    {
        if(toTime)
        {
            // Example to time
            long ende = System.currentTimeMillis();
            String klasse = invocation.getTarget().toString();
            String methode = invocation.getMethod().getName();
            System.out.println(klasse + ":" + methode + " -> " + (ende - start) + "ms");
        }
    }
}
}

```

9. Listing

2.4 Clients, BCI (Business Common Interface)

Auf der Client Seite befindet sich das User Interface, mit dem dieser die Applikation nutzen kann. Auf dem Server hingegen befindet sich die Logik der Applikation, mit dem benötigten Code.

TLGen generiert vollständig die Klassen, die notwendig für die Kommunikation zwischen Client und Server sind. Für den Client generiert TLGen auch die Test Klassen (auf Basis von JUnit), die für die Tests der Applikation benötigt wird.

Als Beispiel ein generierter Aufruf vom Server, um einen Satz in eine Datenbank abzuspeichern:

```

CoFormularDataIf ciFormular = getFormularObject();
// make a factory client for call the session bean
FormularBci factory = FormularBciFactory.getFormularBci();
// call the session bean method over the client factory
ciFormular = factory.createFormular(ciFormular);

```

10. Listing

Dieser Serveraufruf ist in zwei Schritten durchzuführen: Zuerst wird die notwendige Factory aufgerufen (diese entspricht in der Regel einer bestimmten Session Bean auf der Server Seite) und letztendlich mit Hilfe der Factory die gewünschte Methode. Im oberen Beispiel ist die Speicherung in der Datenbank eines Formular Objekts dargestellt.

TLGen generiert zwei Klassen/Interfaces, die diese Aufgabe erfüllen. In unserem Beispiel, die Klasse *FormularBciFactory.java* und die Interface *FormularBci.java*, Teil des generierten Code für die Factory:

```

/**
 * getFormularBci()
 */

```

```

public static synchronized FormularBci getFormularBci() throws PersistenceException {
    return (FormularBci) getBciImplementation(FormularBci.class);
}

```

11. Listing

Als auch für die Interfaces:

```

@Remote
public interface FormularBci {

    // Client interface Fields
    public final static String JNDL_NAME = "eu.stardata.server.core.formular.FormularSessionBean";
    public final static String MANAGER_NAME = "managerFormular";
    public final static String EAR_NAME = "efp";
    .....
    public CoFormularDataIf createFormular(CoFormularDataIf arg) throws PersistenceException;
    .....
}

```

12. Listing

2.4.1 Test Klassen

TLGen generiert für die gewünschten Aufrufe die kompletten Test Klassen mit generierten Test-Werten oder aus einer Daten-Datei die geladenen Test-Daten. Für welche Aufrufe die Test Klassen (JUnit) zu generieren sind, soll in der Konfigurations-Datei eingetragen werden (für weitere Details siehe Kapitel 4.4.8 und 4.5.3.6).

Test Klassen kann man testen und die Ergebnisse in einer Datei printen oder direkt auf den Bildschirm ausgeben (z. B. in Eclipse) oder aber im JUnit, wo sie mit Hilfe von Asset Methode zu Verfügung gestellt werden, ansehen. Listing 13 zeigt ein Beispiel für eine Test Klasse:

```

public class FormularWriteReadTest extends TestCase {
    private static final long serialVersionUID = 435512065;

    // Test class data fields
    private static CoFormularDataIf m_clFormular = null;
    /**
     * Default Constructor
     */
    public FormularWriteReadTest() {
        super();
        // Insert your configuration data for application server
        de.stardata.base.config.InitProgramm.put(de.stardata.base.config.ConfigKeyNames.INITIAL_CONTEXT,
"jnp://localhost:9099");
        de.stardata.base.config.InitProgramm.put(de.stardata.base.config.ConfigKeyNames.INITIAL_CONTEXT
_FACTORY, "org.jnp.interfaces.NamingContextFactory");
        de.stardata.base.config.InitProgramm.put(de.stardata.base.config.ConfigKeyNames.INITIAL_PKG_PRE
FIXES, "org.jboss.naming:org.jnp.interfaces");
    }

    // Test class methods
    /**
     * Test method "testCreateFormular()"
     */
    public void testCreateFormular() {
        try {
            m_clFormular = new CoFormularData();
            // initialize data for test class
            m_clFormular.setCoreId(3);
            .....
            m_clFormular.setLanguage(writeLanguage());
            m_clFormular.setMandant(writeMandant());
            // make a factory client for call the session bean

```

```

        FormularBci factory = FormularBciFactory.getFormularBci();
        // call the session bean method over the client factory
        m_clFormular = factory.createFormular(m_clFormular);
        // print the new primary key
        System.out.println("New primary key is = " + m_clFormular.getFormularId());
    } catch(PersistenceException ex) {
        ex.printStackTrace();
    }
}

```

13. Listing Test Klassen Beispiel

2.4.1.1 Verwendung von Test Daten aus externen Dateien

Mit Hilfe der Konfigurations-Datei kann man eine XML Datei generieren, welche die Daten-Struktur einer oder mehrerer Test-Objekte nachbildet. Diese XML kann mit den gewünschten Daten erweitert und für Tests genutzt werden. Solch eine Datei wird in Listing 14 dargestellt:

```

<?xml version="1.0" encoding="UTF-8"?>
<TestMethod xsi:noNamespaceSchemaLocation = "C:/Project-TLGen-
2.1/source/generated/TestDataSchema.xsd" name = "testCreateFormular" xmlns:xsi =
"http://www.w3.org/2001/XMLSchema-instance" type = "void">
    <Variable name = "FormularId" value = "2" type = "long"/>
    <Variable name = "CoreId" value = "2" type = "long"/>
    .....
    <Variable name = "Document_id" value = "2" type = "long"/>
    .....
    <Method name = "writeMandant" type = "eu.stardata.core.cor.dataif.CoMandantDataIf">
        <Variable name = "MandantId" value = "4" type = "long"/>
        .....
        <Variable name = "Code" value = "-628" type = "java.lang.String"/>
    </Method>
</TestMethod>

```

14. Listing

Die verwendete Schema-Datei "TestDataSchema.xsd" hat folgende Struktur:

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--W3C Schema generated by XMLSpy v2009 (http://www.altova.com)-->
<!--Please add namespace attributes, a targetNamespace attribute and import elements according to your
requirements-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <xs:import namespace="http://www.w3.org/XML/1998/namespace"/>
    <xs:element name="Variable">
        <xs:complexType>
            <xs:attribute name="name" use="required" type="xs:anySimpleType"/>
            <xs:attribute name="type" type="xs:anySimpleType"/>
            <xs:attribute name="value" type="xs:anySimpleType"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="Type">
        <xs:complexType mixed="true">
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element ref="Variable"/>
                <xs:element ref="Type"/>
                <xs:element ref="Enum"/>
            </xs:choice>
            <xs:attribute name="name" use="required" type="xs:anySimpleType"/>
            <xs:attribute name="type" type="xs:anySimpleType"/>
        </xs:complexType>
    </xs:element>

```

```

        </xs:complexType>
</xs:element>
<xs:element name="TestMethod">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="Variable"/>
      <xs:element ref="Method"/>
      <xs:element ref="Type"/>
      <xs:element ref="Enum"/>
    </xs:choice>
    <xs:attribute name="name" use="required" type="xs:anySimpleType"/>
    <xs:attribute name="type" type="xs:anySimpleType"/>
  </xs:complexType>
</xs:element>
<xs:element name="Method">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="Type"/>
      <xs:element ref="Enum"/>
      <xs:element ref="Variable"/>
      <xs:element ref="Method"/>
    </xs:choice>
    <xs:attribute name="name" use="required" type="xs:anySimpleType"/>
    <xs:attribute name="type" type="xs:anySimpleType"/>
  </xs:complexType>
</xs:element>
<xs:element name="Enum">
  <xs:complexType>
    <xs:attribute name="name" use="required" type="xs:anySimpleType"/>
    <xs:attribute name="type" type="xs:anySimpleType"/>
    <xs:attribute name="value" type="xs:anySimpleType"/>
  </xs:complexType>
</xs:element>
</xs:schema>

```

15. Listing

2.5 Manager Klassen

Eine Manager Klasse ist eine lokale Stateless Session Bean, die für die Steuerung der Persistenz (Entity Bean) genutzt wird.

Manager Klassen können Default Methoden für Datenbank-Administration für CRUD sein oder aber auch Datenbank-Zugriffsmethoden, generiert auf Basis von selbst geschriebenen SQL's. In Tabelle 3 sind die Default Methoden, welche TLGen automatisch generieren kann:

Tabelle 3: Automatisch generierbare Default Methoden

Nr.	Method Name	Parameter	Return	Funktionalität
1	create"Name"	Objekt	Objekt	Speichern eins neuen Objekts in der Datenbank
2	update"Name"	Objekt	-	Aktualisieren eines Objekts in der Datenbank
3	remove"Name"	Object	-	Löschen eines Objekts in der Datenbank
4	flusch"Name"	Object	-	Beenden des Vorgang
5	close"Name"	-	-	Schließt eine Entity Object
6	clear"Name"	-	-	

7	findByPrimaryKey"Name"	Object mit PK	Object	Lesen ein Objekt von DB mit den PK
8	findAll"Name"	-	Object[]	Lesen aller gespeicherter Objekte
9	findBy"Name"	Parameter List	Object[]	Lesen aller Objekte für die Parameter List

Ein Beispiel für generierten Code aus Tabelle 3, Zeile 9 ist in Listing 16 dargestellt:

```
public class ProductManager extends BaseManager implements ProductBcIf
{
    private String m_getNameAndOffer = "select u from ProductEntity u
where u.name = :name and u.productOfferingId = :productOfferingId";
.....
/**
 * Manager method "findByNameAndOffer()"
 * @param name
 * @param productOfferingId
 * @return
 * @throws PersistenceException
 */
public ProductDataIf[] findByNameAndOffer(String name, String productOfferingId) throws
PersistenceException {
    try {
        ProductDataIf[] listData = null;
        List<?> list = m_manager.createQuery(m_getNameAndOffer).setParameter("name",
name).setParameter("productOfferingId",productOfferingId).getResultList();
        if(list != null && list.size() > 0) {
            listData = new ProductDataIf[list.size()];
            int idx = 0;
            for(Object entity : list) {
                if(entity != null) {
                    listData[idx++] = ((ProductEntity)entity).callProduct();
                }
            }
        }
        return listData;
    } catch(Exception ex) {
        throw new PersistenceException(ex.getMessage(),ex,1);
    }
}
.....
}
```

16. Listing

Das Lesen, Speichern oder Löschen von Objekten kann direkt im Aufruf gesteuert werden, ob nur das „Vater“ Objekt gelesen, gespeichert oder gelöscht wird oder aber zusammen mit seinen „Kindern“ (solange die Beziehungen zwischen diesen „Null“ akzeptiert).

Für das selbst geschriebene SQL bietet TLGen eine Fülle von Modellen an, die sehr einfach in der Konfigurations-Datei zu nutzen sind, mit der letztendlich TLGen den entsprechenden Code generiert. In der Regel ist es nur sehr selten notwendig ein solches SQL zu schreiben, weil durch automatisches Einbeziehen von Relationen werden alle benötigten Objekte gelesen und abgespeichert.

Es folgt ein Ausschnitt einer generierten Manager Klasse:

```
package eu.stardata.server.core.formular.manager;
```

```

import javax.persistence.EntityManager;
.....
import javax.ejb.Local;

@Stateless(name = eu.stardata.server.core.formular.bci.FormularBci.JNDI_NAME)
@Local(eu.stardata.server.core.formular.bci.FormularBci.class)
public class FormularManager extends BaseManager implements
eu.stardata.server.core.formular.bci.FormularBci {
    private static final long serialVersionUID = 644414680;

    // Manager class data fields
    @PersistenceContext(unitName = FormularBci.MANAGER_NAME)
    public EntityManager m_manager = null;
    private String m_getName = "select o from FormularEntity o where o.formular = :formular";
    private String m_SQL_FIND_ALL = "select o from FormularEntity o";
.....
/**
 * Manager method "findByPrimaryKey()"
 * @param arg
 * @return
 * @throws PersistenceException
 */
public CoFormularDataIf findByPrimaryKey(CoFormularDataIf arg) throws PersistenceException {
    try
    {
        CoFormularDataIf classData = null;
        FormularEntity entity = m_manager.find(FormularEntity.class, arg.getFormularId());
        if(entity != null){
            classData = entity.callFormular();
        }
        return classData;
    } catch(Exception ex){
        throw new PersistenceException(ex.getMessage(),ex, 1);
    }
}
.....
}

```

17. Listing

Alle in den Manager Klassen verwendeten Methoden können von den dazugehörigen Session Bean übernommen werden oder auch nicht. Dies kann über die Konfigurations-Datei gesteuert werden (Default bedeutet, dass alle übernommen werden).

In Tabelle 4 die Annotationen, die für eine Manager Klasse verwendet werden.

Tabelle 4: Annotationen für Manager Klassen

Annotationen	Vervendung inTLGen	Kommentar
Stateless	ja	Bezeichnet eine Session Stateless Bean
Local	ja	Ist nur lokal zu verwenden
PersistenceContext	ja	Definiert den Namen des Managers

2.5.1 Manager Interfaces

Weil Manager Klassen eigentlich lokale Session Beans sind, benötigen sie ein eigenes Interface. Diese werden auch automatisch von TLGen generiert.

Ein Beispiel für ein Interface einer Manager Klasse:

18. Listing

```
import eu.stardata.core.form.dataif.CoFormularDataIf;
import java.lang.String;
import eu.stardata.server.common.exception.PersistenceException;
import eu.stardata.server.core.formular.bci.FormularBci;
import eu.stardata.server.core.formular.entity.FormularEntity;

public interface FormularBci {

    // Manager interface data fields
    public String JNDI_NAME = "eu.stardata.server.core.formular.bci.FormularBci";
    public String EAR_NAME = "efp";

    // Manager interface methods
    public CoFormularDataIf findByPrimaryKey(CoFormularDataIf arg) throws PersistenceException;
    .....
    public void close() throws PersistenceException;
}

```

2.6 Entity Beans

Die Entity Beans sind das Tor zur Datenbank, besser gesagt, sie sind die Schnittstellen für eine relationale Datenbank. Eine Entity Bean sollte einer Tabelle in der Datenbank entsprechen und so eine saubere sowie strukturierte Architektur für die Persistenzschicht ermöglichen. Das ist sehr wichtig für die Wartung des Codes und für weitere Entwicklungen. Entity Beans sind POJO's (Old Java Objects), d. h. sie sind normale Java Klassen und anders als die Session Beans, brauchen sie keine Interfaces.

Jede Entity Bean muss einen Standardkonstruktor besitzen, weil es unter anderem Aufgabe einer Manager Klasse ist, eine Instanz für die Entity Bean zu erzeugen.

TLGen generiert in einer Entity Bean die notwendigen Annotationen, Variablen, Default Konstruktor, Mapping Methoden zwischen Daten Objekt Attributen, Tabellen-Spalten, einige Hilfs-Methoden, die diese Mapping unterstützen und die Methoden für die Relationen.

Für eine Entity Bean werden auch mehrere Annotationen und Variablen generiert, wie in Listing 19 dargestellt:

```
package eu.stardata.server.core.formular.entity;
import javax.persistence.GeneratedValue;
.....
import javax.persistence.Entity;

@Entity
@Table(name="CORE_FORMULAR")
@SequenceGenerator(name="SEQ_STORE_FORMULAR", sequenceName="CORE_FORMULAR_SEQ",
initialValue=1, allocationSize=20)
public class FormularEntity {
    private static final long serialVersionUID = 802995295;

    // Entity data field
    private CoFormularDataIf m_formular = null;

    // Fields from relations tables
    private List<PageEntity> m_page = new ArrayList<PageEntity>();
    private List<DocumentlistEntity> m_documentlist = new ArrayList<DocumentlistEntity>();
    private LanguageEntity m_language = null;
    .....
    // constructors
    /**
     * Default Constructor
     */
}

```

```

public FormularEntity() {
}

/**
 * Constructor
 * @param arg
 */
public FormularEntity(CoFormularDataIf arg) {
    m_formular = arg;
    fillFormular();
}

// Helper methods
/**
 * Helper Method "makeFormular"
 */
public CoFormularDataIf makeFormular() {
    if(m_formular == null) {
        m_formular = new eu.stardata.core.form.data.CoFormularData();
    }
    return m_formular;
}
}
}

```

19. Listing

In Tabelle 5 sind die Annotationen dargestellt, die für Entity Beans von EJB3 vorgesehen sind:

Tabelle 5: Annotationen für Entity Beans (EJB3)

Annotationen	Vervendung inTLGen	Kommentar
Entity	ja	Definiert eine Entity Bean
Table	ja	Table, die zu eine Entity gehören
Column	ja	Spalte einer Table (für Mapping)
SequenceGenerator	ja	Definiert die Art von Sequence Generator
GeneratedValue	ja	Generiert eine Sequence (für die automatische Generierung von ID's)
Id	ja	Definiert eine PK
ManyToMany	ja	Relation Typ (siehe 2.6.1.4 und 4.5.3.8)
ManyToOne	ja	Relation Typ (siehe 2.6.1.2 und 4.5.3.8)
OneToMany	ja	Relation Typ (siehe 2.6.1.1 und 4.5.3.8)
OneToOne	ja	Relation Typ (siehe 2.6.1.3 und 4.5.3.8)
Version	ja	Verwendet für die Versionskennzeichnung, wird für Locking verwendet
PostLoad	ja	Hilfsmethoden, welche vor oder nach Processing aufgerufen werden
PostPersist	ja	Hilfsmethoden, welche vor oder nach Processing aufgerufen werden
PostRemove	ja	Hilfsmethoden, welche vor oder nach Processing

		aufgerufen werden
PostUpdate	ja	Hilfsmethoden, welche vor oder nach Processing aufgerufen werden
PrePersist	ja	Hilfsmethoden, welche vor oder nach Processing aufgerufen werden
PreRemove	ja	Hilfsmethoden, welche vor oder nach Processing aufgerufen werden
PreUpdate	ja	Hilfsmethoden, welche vor oder nach Processing aufgerufen werden
JoinColumn	ja	Verbindungsspalte für eine Relation
JoinColumns	ja	Mehrzahl von Verbindungs-Columns
JoinTable	ja	Verbindung DB Tabelle für eine ManyToMany Relation
Embeddable	ja	Verwendet für komplexe Primary Key
NamedQuery	ja	Definiert eine named SQL Query
NamedNativeQuery	ja	Definiert eine named Native Query
OrderBy	ja	Sortiert die Ergebnisse
Enumerated	nein	Vorgesehen für die nächste Version
EmbeddedId	nein	Vorgesehen für die nächste Version
Transient	nein	Vorgesehen für die nächste Version
TableGenerator	nein	Vorgesehen für die nächste Version
Temporal	nein	Vorgesehen für die nächste Version
Lob	ja	Verwendet für Blob und Clob (Große Daten)
SecondaryTable	nein	Vorgesehen für die nächste Version
Basic	ja	Wird zusammen mit Lob verwendet
PersistenceProperty	nein	Vorgesehen für die nächste Version
PersistenceUnit	nein	Vorgesehen für die nächste Version
UniqueConstraint	nein	Vorgesehen für die nächste Version
Inheritance	nein	Vorgesehen für die nächste Version

Innerhalb einer Entity Bean generiert TLGen eine Reihe von Hilfsmethoden, die für das Mapping von Daten-Relationen notwendig sind. Diese Methoden sind:

- fill„Name“(), hat keine Parameter und wird in Constructor aufgerufen. Diese ist notwendig für das Speichern eines neuen Objektes in der Datenbank für eine “ManyToOne” Relation
- add„Name“(), hat keine Parameter und wird vom Manager aufgerufen. Diese ist notwendig für das Speichern eines neuen Objekts in der Datenbank für eine “OneToMany” Relation
- call„Name“(), wird für alle Relationen beim Lesen eines Objektes aus der Datenbank benutzt.

Listing 20 zeigt ein Beispiel-Code:

```
/**
 * Helper Method "callFormular"
```

```

*/
public CoFormularDataIf callFormular() {
    // Relation ManyToOne, read data for "Formular" child, "Language"
    if(getLanguage() != null) {
        makeFormular().setLanguage(getLanguage().callLanguage());
    }
    // Relation ManyToOne, read data for "Formular" child, "Mandant"
    if(getMandant() != null) {
        makeFormular().setMandant(getMandant().callMandant());
    }
    // return the data object
    return makeFormular();
}

```

20. Listing

In Listing 21 ist das Beispiel einer Mapping-Methode mit einem Sequence Generator für Primary Key dargestellt:

```

// Methods from columns
@Id
@Column(name = "FORMULAR_ID")
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "SEQ_STORE_FORMULAR")
public long getFormularId() {
    return makeFormular().getFormularId();
}

public void setFormularId(long arg) {
    makeFormular().setFormularId(arg);
}

```

21. Listing

TLGen generiert auch automatisch komplex verschachtelte Aufrufe für das Mapping. Beispiel:

```

@Column(name = "authenticationFirstname", length = 255)
public String getAuthenticationFirstname() {
    if(makeUser().getAuthentication() != null && makeUser().getAuthentication().getName() !=
null) {
        return makeUser().getAuthentication().getName().getFirstname();
    }
    return null;
}

public void setAuthenticationFirstname(String arg) {
    if(makeUser().getAuthentication() == null) {
        makeUser().setAuthentication(new Authentication());
    }
    if(makeUser().getAuthentication().getName() == null) {
        makeUser().getAuthentication().setName(new Name());
    }
    makeUser().getAuthentication().getName().setFirstname(arg);
}

```

22. Listing

2.6.1 Relationen

Ein wichtiger Bestandteil von Entity Beans sind die Relationen zwischen verschiedenen Objekten, die in einer Datenbank abgespeichert oder ausgelesen werden.

In diesem Kapitel werden die Verwendung von Relationen sowie der von TLGen generiertem Code beschrieben.

2.6.1.1 Relation „OneToMany“

„OneToMany“ Relation wird verwendet, wenn ein Objekt mehrere Subobjekte (Kinder) besitzt. In einer Datenbank wird dieses durch eine Tabelle, die eine „OneToMany“ Relation zu einer anderen Tabelle hat, dargestellt. In Daten-Interface ist so eine Relation durch eine Liste wie in Listing 23 erkennbar:

```
public interface CoFormularDataIf extends BaseDataIf
{
.....
    public abstract List<CoPageDataIf> getPage();
    public abstract void setPage(List<CoPageDataIf> arg);
.....
}
von "Formular" Objekt zu "Page" Object
{
.....
    public abstract CoFormularDataIf getFormular();
    public abstract void setFormular(CoFormularDataIf arg);
.....
}
und von "Page" Objekt zu "Formular" Objekt.
```

23. Listing

und

in Entity Bean wie in Listing 24.

```
// Methods from relations tables
@OneToMany(cascade = {CascadeType.PERSIST,CascadeType.MERGE,CascadeType.REMOVE},
mappedBy = "formular",targetEntity = PageEntity.class)
public List<PageEntity> getPage() {
    return m_page;
}

public void setPage(List<PageEntity> arg) {
    m_page = arg;
}
```

24. Listing

Ob solch ein Objekt gespeichert werden muss oder nicht hängt von der Art der Relation ab, „*not null*“ oder „*null*“. All seine Kinder werden automatisch gespeichert, wenn im Vater Objekt die Kinderdaten vorhanden sind. Der Speicherungsprozess von Daten erfolgt automatisch durch einen einzigen Aufruf von einer „*create*“ Methode (siehe Listing 13). Beim Lesen werden zusammen mit dem Vater auch alle seine Kinder geholt, wenn diese vorhanden sind. Die Tiefe von Relations-Ebenen kann gesteuert werden, d. h. es kann nur der Vater gelesen/gespeichert werden oder die Ebenen können immer weiter um die Kinder erweitert werden. Diese Ebenenerweiterungen werden dann möglich, wenn die Art der Relationen „*null*“ ist. (Beispiel: ein „*Formular*“ hat mehrere „*Pages*“ und jede Page hat mehrere „*Fields*“ und so weiter)

Eine UML-Darstellung für so eine Relation befindet sich in Abbildung 8. Zwischen „Formular“ Objekt und „Page“ Objekt existiert eine „*null OneToMany*“ Relation und zwischen „Page“ Objekt und „Fields“ Objekt eine „*not null OneToMany*“ Relation. Als Datenmodell wird diese Relation in Abbildung 9 dargestellt.

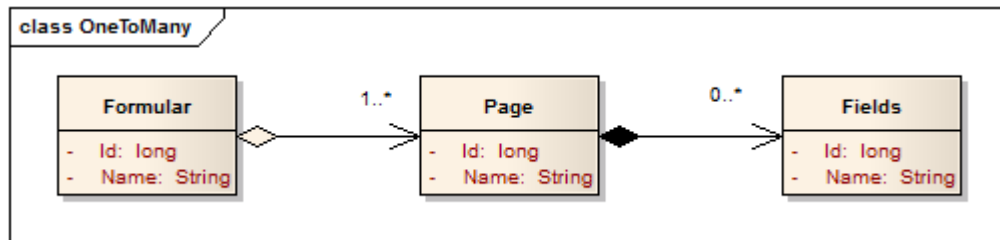


Abbildung 8: „OneToMany“

2.6.1.2 Relation „ManyToOne“

Diese Relation ist eine symmetrische Relation zu „OneToMany“, d. h. ein Objekt kann in mehrere andere Objekte verwendet werden. Auch bei dieser Art von Relation kann eine „not null“ Verbindung existieren und dieser Link muss immer mit Daten versorgt werden, d. h. in der Datenbank muss der Fremdschlüssel in den Kinder-Tabellen „not null“ sein (z.B. ein FK =Fremdschlüssel) mit dem Typ „long“ soll immer größer als 0 sein und in der Tabelle, wo der FK zeigt, muss ein richtiger Satz vorhanden sein).

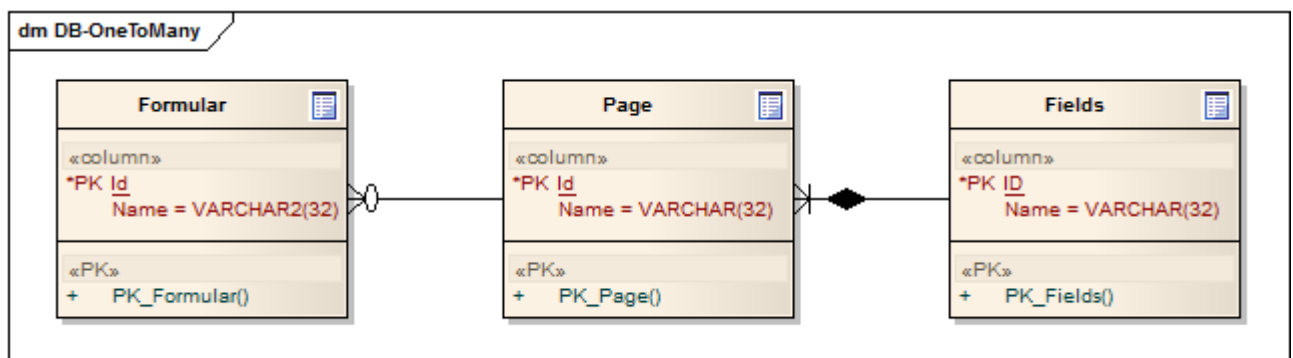


Abbildung 9: „OneToMany“ Datenmodell

Die UML Darstellung ist gleich mit der Abbildung 8 und das Datenmodell wie Abbildung 9, aber die Richtung ist umgekehrt als bei der Relation „*OneToMany*“, das gilt auch für die Code-Darstellung in Listing 22.

Für den Entity Bean Code ist Listing 25 zu beachten:

```

@ManyToOne(optional = true, targetEntity = WebstyleEntity.class)
@JoinColumn(name = "WEBSTYLE_ID", insertable = true, updatable = true, nullable = true,
unique = false, referencedColumnName = "WEBSTYLE_ID")
public WebstyleEntity getWebstyle() {
    return m_webstyle;
}

public void setWebstyle(WebstyleEntity arg) {
    m_webstyle = arg;
}
  
```

25. Listing**2.6.1.3 Relation „OneToOne“**

Die Relation „*OneToOne*“ verbindet zwei Objekte so, dass eines ein einziges Kind besitzt. Auch bei dieser Relation ist eine „*null*“-Verbindung möglich, d. h. der Fremdschlüssel kann null sein, oder „not null“. In diesem Fall muss der Fremdschlüssel immer vorhanden sein und zu einem vorhandenen Satz in der Datenbank die Verbindung anzeigen.

Auch bei dieser Verbindung gibt es einen Vater und ein Kind.

Code für diese Relation ist im Listing 26 für die Daten Interfaces und im Listing 27 für die Entity Bean zu sehen.

```
public interface CoFieldDataIf extends BaseDataIf
{
.....
    public abstract CoTableDataIf getTable();
    public abstract void setTable(CoTableDataIf arg);
.....
}
```

26. Listing

```
public class FieldEntity
{
    private TableEntity m_table = null;
.....
    @OneToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE, CascadeType.REMOVE},
targetEntity = TableEntity.class, optional = true)
    @JoinColumn(name = "TABLE_ID", insertable = false, updatable = false, nullable = true,
unique = false, referencedColumnName = "TABLE_ID")
    public TableEntity getTable() {
        return m_table;
    }

    public void setTable(TableEntity arg) {
        m_table = arg;
    }
.....
}
```

27. Listing**2.6.1.4 Relation „ManyToMany“**

Die Relation „*ManyToMany*“ ist die komplexe Form für das Verbinden mehrerer Objekte/Tabellen. Auf der Code-Ebene hat jedes Objekt eine Liste mit den anderen Objekten wie in der Listing 28:

```
public interface UserDataIf extends BaseDataIf
{
.....
    // Methods from relations tables
    public abstract List<UserRoleDataIf> getUserRole();
    public abstract void setUserRole(List<UserRoleDataIf> arg);
.....
}
Von "User" Objekt, und von "UserRole" Objekt:
```

```

public interface UserRoleDataIf extends BaseDataIf
{
.....
// Methods from relations tables
public abstract List<UserDataIf> getUser();
public abstract void setUser(List<UserDataIf> arg);
.....
}

```

28. Listing

Auf der Datenbank-Ebene ist für diese Relation eine dritte Tabelle notwendig, eine so genannte Mapping Tabelle, wie in Abbildung 11. Die Darstellung in einem Domain Modell ist in Abbildung 10 zu ersehen.

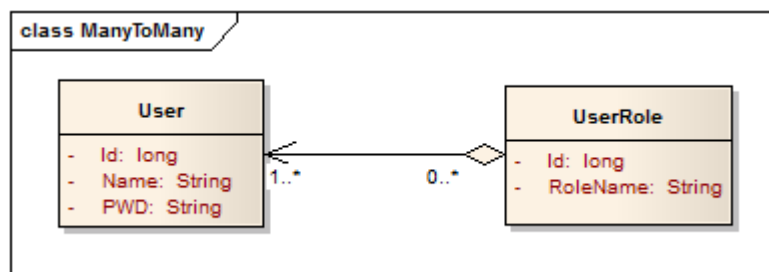


Abbildung 10: „ManyToMany“ Darstellung im UML Domainmodell

Aus einem Domain Modell generiert TLGen für „ManyToMany“-Relationen automatisch die nötige Mapping Tabelle als SQL Skript.

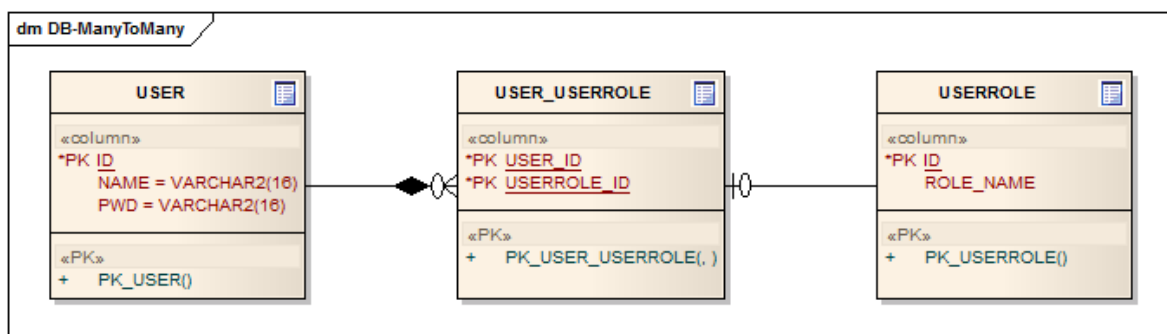


Abbildung 11: „ManyToMany“ DB Relation

In den Entity Beans werden Annotationen mit den dazugehörigen Methoden, wie in unserem Beispiel, sowohl in der „UserEntity“ Bean als auch in „UserRoleEntity“ Bean generiert (siehe Listing 29):

```

@Entity
@Table(name="UserTable")
@SequenceGenerator(name="SEQ_STORE_USER", sequenceName="USERTABLE_SEQ", initialValue=1,
allocationSize=10)
public class UserEntity extends BaseData implements Serializable {
// Entity data field
private UserDataIf m_user = null;
.....
}

```

```

// Methods from relations tables
@ManyToMany(targetEntity = UserRoleEntity.class)
@JoinTable(name = "UserRole_UserTable",inverseJoinColumns = {@JoinColumn(name =
"UserRole_ID",insertable = true, updatable = true, nullable = false, unique = false)},
joinColumns = {@JoinColumn(name = "UserTable_ID",insertable = true, updatable = true,
nullable = false, unique = false)})
public List<UserRoleEntity> getUserRole() {
    return m_userRole;
}

public void setUserRole(List<UserRoleEntity> arg) {
    m_userRole = arg;
}.....
}

```

Für "User" und

```

@Entity
@Table(name="UserRole")
@SequenceGenerator(name="SEQ_STORE_USERROLE", sequenceName="USERROLE_SEQ", initialValue=1,
allocationSize=10)
public class UserRoleEntity extends BaseData implements Serializable {
    private static final long serialVersionUID = 413271821;

    // Entity data field
    private UserRoleDataIf m_userRole = null;
.....
// Methods from relations tables
@ManyToMany(cascade = {CascadeType.PERSIST,CascadeType.MERGE,CascadeType.REMOVE},
mappedBy = "UserRole",targetEntity = UserEntity.class)
public List<UserEntity> getUser() {
    return m_user;
}

public void setUser(List<UserEntity> arg) {
    m_user = arg;
}
.....
}
Für "UserRole"

```

29. Listing

In der Praxis haben wir 3 Möglichkeiten Tabellen zwischen denen eine „ManyToMany“ Relation existiert, abzuspeichern, wie folgt:

- Speichern beider Objekte gleichzeitig; das „User“ Objekt beinhaltet die Daten für dessen „UserRole“ und in der Datenbank werden drei Sätze gespeichert, einer in der „USERROLE“ Tabelle, der andere in die „USER“ Tabelle und der dritte Satz in die Tabelle „USER_USERROLE“ mit beiden ID's (USER_ID und USERROLE_ID).
- Speichern einer „UserRole“ von einem oder mehreren Objekten in der „USERROLE“ Tabelle
- Speichern eines „User“ Objektes, der mit einem „USERROLE_ID“ einer vorhandenen „USERROLE“ versorgt wird, in dem Datenbank Objekt einer „UserRole“. In diesem Fall werden zwei Sätze in der Datenbank gespeichert, einer in die „USER“ Tabelle und ein zweiter in der Tabelle „USER_USERROLE“ mit beiden ID's. Falls der „USERROLE_ID“ zeigt, dass ein „USERROLE“ in dieser Tabelle nicht vorhanden ist, wird eine Exception zurück zu den Client gesendet und eine Roll - Back Transaktion folgt.

2.7 Message Driven-Bean

Bestimmte Aufgaben von komplexen Anwendungen müssen nicht sofort erledigt werden, sondern können mit bestimmten Nachrichten zu einem späteren Zeitpunkt verschoben werden. Für diese Art von Aufgaben hat der Applikation-Server die Message Driven Beans.

Die Kommunikation zwischen Client und Server über Message Driven Beans läuft über dem JMS (Java Message Service). TLGen kann die Message Driven Bean mit sehr wenig Aufwand generieren.

Mit den generierten Message Driver Beans kann die Applikation über Callback-Klassen kommunizieren (siehe 2.8.3) und so ist eine klare Trennung zwischen der technischen und der fachlichen Schicht realisierbar.

Mit Hilfe der Steuerung über die Konfigurationsdateien können für die Message Driven Beans folgende Features generiert werden:

- Annotationen mit Parametern (Tabelle 6)
- Client-Klassen
- Message Driven Bean Klassen
- Callback Klassen für die Verbindung mit der Fachlogik-Schicht.

Tabelle 6: Annotationen für Message Driven Bean

Annotationen	Verwendung in TLGen 2.5	Kommentar
javax.annotation.Resource	ja	Definiert die Ressource des Message Driven Beans
javax.ejb.MessageDriven	ja	Definiert ein Message Driven Bean
javax.ejb.ActivationConfigProperty	ja	Setzt die Eigenschaften des Message Driven Beans

2.8 Services

Die Applikation Server bieten für EJB 3 zwei Services an, Timer-Services, mit deren Hilfe ein bestimmter Prozess zu einem bestimmten Zeitpunkt aufgerufen werden kann (periodisch wiederholbar) und Web-Services, die ein Session-Bean über Internet/Intranet aufrufen können. TLGen kann beide Services generieren.

2.8.1 Timer-Service

Ein Timer-Service kommt in der Applikation zum Einsatz, wenn eine zeitgesteuerte Verarbeitung benötigt wird. Der Methodenaufruf kann wahlweise einmalig oder in einem bestimmten Zeitintervall dauerhaft erfolgen.

TLGen generiert ein Timer-Service und seine Komponenten, wie den Client des Timer-Service, Timer-Service und eine Callback Klasse (siehe 2.8.3) für die Implementierung von eigener Fachlogik, die von diesem Service benötigt wird.

Timer-Service werden von TLGen in drei Varianten generiert:

1. Vollautomatisch, in diesem Fall werden die Start-, Stopp-Zeit Aufrufe und die periodische Wiederholung automatisch nach der Timer-Service Initialisierung gesteuert.
2. Start - und Stopp - Zeit wird durch die Client-Aufrufe gesteuert.
3. Verwendet die automatische und manuelle Steuerungs-Möglichkeit.

Die Timer-Services können z.B. von einem Session-Bean oder von einem Message Driven-Bean verwendet werden.

2.8.2 Web-Services

Ein Webservice ist ein beliebiger Service (z. B. ein Session-Bean), der von einem Client über z.B. das Internet aufgerufen werden kann.

Es werden zwei Arten von Webservice unterschieden, RPC (Remote Procedure Call) und DOCUMENT. RPC ist ein Aufruf einer Methode (z. B. eine Methode von einem Session-Bean) über Web.

TLGen generiert für diese Services den Client, den Webservice und eine Callback Klasse (siehe 2.8.3), falls diese nötig ist.

2.8.3 Callback Klasse

Diese Art von Klasse ist als Schnittstelle zwischen dem technischen Code, der von TLGen generiert wird, und dem fachlichen Code, der die Fachlogik des Applikation beinhaltet, gedacht.

Die Generierung der Klasse kann wie folgt gesteuert werden:

- **merge** = 0, TLGen generiert diese Klasse immer wieder neu und überschreibt vorhandene.
- **merge** = 1, diese Klasse wird nur generiert, falls sie nicht existiert.

- **merge** = 2, wird eine Neue generiert und mit der Implementierung der vorhandenen zusammengefasst.

2.9 Free Klasse

Diese „freien“ Klassen sind für den eigenen Programm-Teil des Projekts gedacht und beinhaltet die Fachlogik. Dieser muss selbst implementiert werden. Falls der Generator eine freie Klasse findet, welche angegeben wurde und sich innerhalb des Ordners „generated“ befindet, kann er, abhängig von ein Parameter „merge“, wie folgt diese überschreiben oder neu generieren:

- **merge** = 0, TLGen generiert diese Klasse immer wieder neu und überschreibt vorhandene.
- **merge** = 1, diese Klasse wird nur generiert, falls sie nicht existiert.
- **merge** = 2, wird eine Neue generiert und mit der Implementierung der vorhandenen zusammengefasst.

2.10 Generierung von Datenbank und SQL Skripte

TLGen kann ein Datenbank SQL Skript aus einem Domainmodell, mit seinen Tabellen, Spalten, Relationen, Indexes und Sequenzen generieren. Wenn der Datenbankzugriff (User und Password) in der Konfiguration Datei vorhanden ist, verwendet TLGen das generierte Skript, um die Datenbank zu generieren.

Dies ist auch möglich, wenn die Datenbank schon vorhanden ist. Dann werden nur die Änderungen (es wird auch ein Änderung-Skript generiert) generiert, und somit werden die schon vorhandenen Daten nicht gelöscht.

Skript-Beispiele werden in Listing 30 gezeigt:

```

----- Drop all Tables -----
DROP TABLE Contract CASCADE CONSTRAINTS;
DROP TABLE Product CASCADE CONSTRAINTS;
.....
----- Create Tables -----
CREATE TABLE Product (
    Product_ID NUMBER(19,0) NOT NULL,
    DB_VERSION NUMBER(16) NULL,
    description VARCHAR2(255) NULL,
    .....
    shortName VARCHAR2(255) NULL,
    CustomerService_ID NUMBER(19,0) NULL,
    PRIMARY KEY (Product_ID) USING INDEX TABLESPACE XINDEX
) Tablespace XDATA;
.....
----- Foreign key -----
ALTER TABLE Product ADD CONSTRAINT FK201A7DE FOREIGN KEY (CustomerService_ID) REFERENCES
CustomerService;
ALTER TABLE Contract ADD CONSTRAINT FKF96859 FOREIGN KEY (Customer_ID) REFERENCES
Customer;
ALTER TABLE Contract ADD CONSTRAINT FK31C7538 FOREIGN KEY (AccessControlContext_ID)
REFERENCES AccessControlContext;
ALTER TABLE Contract ADD CONSTRAINT FK7FE790 FOREIGN KEY (Offer_ID) REFERENCES Offer;
.....
----- Create Sequences -----
CREATE SEQUENCE Contract_SEQ START WITH 1 INCREMENT BY 1 CACHE 10 MINVALUE 1;
CREATE SEQUENCE Product_SEQ START WITH 1 INCREMENT BY 1 CACHE 10 MINVALUE 1;

```

30. Listing

Die sofortige Generierung einer Datenbank ist wichtig für den Datenbank Optimierungs-Prozess (neue und alte Projekte) siehe Kap. 2.1

2.11 Regel für die Generierung von Namen

TLGen verwendet im Code-Generierungs-Prozess zwei Arten von Regeln für die Namen-Generierung:

- Die Domainmodell-Namen sind eigentlich Namen nach den Java Konventionen für Datenbank-Namen. In diesem Fall verwendet die Generierung der Datenbank Namen, schon vorhandene Namen aus dem Domainmodell.
- Bei einer schon vorhandenen Datenbank (Legacy Projekte) sind in den meisten Fällen Namen, die anders sind als die Java Namenskonventionen. Für diesen Fall sind acht Namensumwandlungen möglich:
 - Aus den Groß geschriebenen Namen, aus mehreren Teilen mit ein „*underscore*“ verbunden, wird ein Name kleingeschrieben, ohne „*underscore*“, bedeutet das die Default Einstellung, siehe Tab 6, Zeile 1
 - „*underscore*“ wird nicht mehr verwendet, d.h. der erste Buchstabe ist groß, Zeile 2
 - „*underscore*“ wird nicht verwendet, der Rest ist gleich, Zeile 4 und 5
 - „*underscore*“ wird verwendet, der restliche Name bleibt gleich, Zeile 6
 - „*underscore*“ wird verwendet, und der Rest verwendet die Java Namen-Konvention, Zeile 7
 - keine Namen-Änderung.

Tabelle 7: Namen-Regel

Nr.	Datenbank Format	Java Format
1	CORE_FORMULAR	Formular
2	CORE_FORMULAR testCoreFormular	CoreFormular TestCoreFormular
3	CORE_FORMULAR testCoreFormular FORMULAR	CoreFormular TestCoreFormular Formular
4	CORE_FORMULAR testCoreFormular	COREFORMULAR TestCoreFormular
5	CORE_FORMULAR testCoreFormular	CoreFormular Testcoreformular
6	CORE_FORMULAR	COREFORMULAR
7	CORE_FORMULAR	Core_formular
8	CORE_FORMULAR	CORE_FORMULAR

Diese Regeln werden einmal für das gesamte Projekt gesetzt, aber wenn unterschiedliche Regeln für die Namen innerhalb einer Session Bean mit seinen Komponenten erwünscht sind, müssen diese gesetzt werden. Details siehe auch Kapitel 4.4.

Diese Regeln können auch von dem Domain Modell in Bezug auf der Datenbank verwendet werden, aber die meisten Datenbanken machen keinen Unterschied zwischen Klein- und Gross- Schreibung.

2.12 Log Dateien Beschreibung

TLGen erzeugt im Generierungs-Prozess mehrere Log-Dateien, in welchen alle Schritte eingetragen werden. Durch diese Log-Dateien kann kontrolliert werden, ob das Projekt-Design den gewünschten Ergebnissen entspricht.

Die Verwendung von Log-Dateien ist einstellbar und der Name ist wählbar, d. h. es kann bzw. können nur eine oder mehrere verwendet werden. Sind es mehrere, weil die Information in diesen Dateien sehr groß ist, wird die Verwendung dieser Dateien somit erleichtert.

TLGen kann folgende Log-Dateien verwenden:

- Eine Log-Datei für die Generierung allgemeiner Schritte (beschreibt nur wichtige Schritte)
- Eine Log-Datei für die Generierung von Session Beans
- Eine Log-Datei für die Generierung von Entity Beans
- Eine Log-Datei für die Generierung von Test Classen
- Eine Log-Datei für die Generierung von Daten Classen/Interfaces
- Eine Log-Datei für die Generierung von Manager Classen.

Es gibt eine Error Log-Datei, in der die Generierung Error, wie zu lange Namen, Fehler im Domainmodell (z. B. Kreise in Relationen) eingetragen werden. Ein Beispiel für eine Error Log-Datei befindet sich in Listing 31 (siehe auch Kap. 2.1):

```
Time Start : Sat Feb 05 13:29:11 CET 2011
0 Domain Error | Class-name is too lang[35] + 'AverageFlatDurationCalculationParam'
```

```
==== Direct circles in relations =====
Warning: TARGET [Order] in a path starting with himself as SOURCE [Order]:  [Order(NOT NULL)-
>Customer(NULL)->Asset(NOT NULL)->AssetItem(NOT NULL)] ---> [Order]
Warning: TARGET [Offer] in a path starting with himself as SOURCE [Offer]:  [Offer(NOT NULL)-
>Customer(NULL)->Asset(NOT NULL)->AssetItem(NOT NULL)->Order(NOT NULL)->Contract(NOT NULL)] --->
[Offer]
.....
==== Relation Problems? =====
FATAL Error: [ServiceDescriptionItem] needed by different mandatory relations starting from [AssetItemStatus]
[AssetItemStatus->AssetItem->Order->Contract->ServiceDescription] ---> [ServiceDescriptionItem]
[AssetItemStatus->AssetItem->OrderItem] ---> [ServiceDescriptionItem]
FATAL Error: [ServiceDescription] needed by different mandatory relations starting from [AssetItem]
[AssetItemStatus->AssetItem->Order->Contract] ---> [ServiceDescription]
[AssetItemStatus->AssetItem->Order->Contract->Offer] ---> [ServiceDescription]
Warning: [ProductRelation] can be reached over [AssetItem] in different relations:
[AssetItemStatus->AssetItem->OrderItem->ServiceDescriptionItem->Product] ---> [ProductRelation]
[ServiceDescriptionItem->AssetItem->OrderItem->Product] ---> [ProductRelation]
```

[OrderItem->ServiceDescriptionItem->AssetItem->Product] ---> [ProductRelation]

31. Listing

2.13 Message Driven Bean

In komplexen Anwendungen müssen die Aufgaben manchmal nicht sofort erledigt werden, sondern erst zu einem späteren Zeitpunkt, nachdem der Anwendung eine Nachricht zugekommen ist.

Für die Message Driver Bean läuft die Kommunikation zwischen Client und Server über JMS (Java Message Service) ab. TLGen kann die Message Driver Bean mit sehr viel weniger Aufwand generieren.

Die so generierte Message Driver Bean kann über die Callback Klasse (siehe 2.8.3) mit der Applikation kommunizieren und so eine klare Trennung zwischen der technischen und der fachlichen Schicht realisieren.

Mit Hilfe der Steuerung über die Konfigurationsdateien können für die Message Driver Bean folgende Features generiert werden:

- Annotationen mit Parameter (Tabelle 8)
- Client Klassen
- Message Driver Bean Klassen selber
- Callback Klasse für die Verbindung mit der Fachlogik-Schicht

Tabelle 8: Annotationen für Message Driven Bean

Annotationen	Verwendung in TLGen 2.5	Kommentar
javax.annotation.Resource	ja	Definiert die Ressource Message Driven Bean
javax.ejb.MessageDriven	ja	Definiert eine Message Driven Bean
javax.ejb.ActivationConfigProperty	ja	Setzt die Properties für eine Message Driven Bean

2.14 Services

Der Applikation Server für EJB 3 bietet zwei Services an, eine Time Service, mit deren Hilfe ein bestimmter Prozess zu einer bestimmten Zeit mehrmals aufgerufen werden kann und ein Web Service, die ein Aufruf einer Session Bean über das Inter/Intra-Net ermöglicht. TLGen kann beide Services generieren.

2.14.1 Timer Service

Ein Timer Service kommt in einer Applikation zum Einsatz, wenn eine zeitgesteuerte Verarbeitung nötig ist.

Der Methodenaufruf kann wahlweise erfolgen, einmalig oder aber in einem bestimmten Zeitintervall dauerhaft.

TLGen generiert ein Timer Service mit seinen Komponenten wie Client für Timer Service Start, Timer Service selbst als auch eine Callback Klasse (siehe 2.8.3) für die Implementierung der Fachlogik, notwendig für dieses Service.

Timer Service werden von TLGen in drei Varianten generiert:

4. Vollkommen automatisch; In diesem Fall sollten Start-, Stopp-Zeit und die periodische Wiederholung automatisch nach einer Timer Service Initialisierung, erfolgen.
5. Start- und Stopp-Zeit, gesteuert durch einen Client Aufruf
6. Automatische und manuelle Steuerungsmöglichkeiten

Der Timer Service kann mit einer Session Bean oder mit einer Message Driven Bean verwendet werden.

2.14.2 Web Services

Eine Web Service ist ein beliebiger Service (z. B. eine Session Bean), der von einem entfernt liegenden Client über ein World Wide Web verwendetes Protokoll aufgerufen werden kann.

Es gibt zwei Webservice-Arten, RPC(Remote Procedure Call) und DOCUMENT. RPC ist eigentlich ein Aufruf von einer Methode (z. B. eine Methode einer Session Bean) über das Web.

Der Unterschied zwischen einem Aufruf vom Client zu einer Session Bean (über Applikation Server Techniken) und einem Client Aufruf einer WebServices liegt in der Aufrufart. Diese basiert auf XML und ist standardisiert. Somit ist jeder Aufruf in der Lage sich mit einfachen Mitteln an der Kommunikation zu beteiligen.

TLGen generiert für diesen Service den Client, den Webservice selber als auch eine Callback Klasse (siehe 2.8.3), falls nötig.

2.14.3 Callback Klasse

Diese Art von Klasse befindet sich als Schnittstelle zwischen dem technischen Code, komplett von TLGen generiert, und dem Fach-Code, welcher die Fachlogik der Applikation beinhaltet.

Diese Klasse kann in drei Modi generiert werden und hat, abhängig von einem Parameter „merge“, folgende Werte:

- merge = 0, TLGen generiert diese Klasse immer wieder neu
- merge = 1, diese Klasse wird generiert nur wenn sie nicht existiert
- merge = 2, wird sie neu generiert und der manuell erstellte Code mit dem neu generierten Code gemergt

2.15 Free Klasse

Diese Klassen sind für den Programmteil, welcher die Fachlogik beinhaltet, gedacht. Der Inhalt dieser Klasse ist manuell zu implementieren. Findet der Generator diese Klassen, abhängig vom Parameter „merge“, dann ergeben sich folgende Werte:

Folgende Attribute werden für diesen Tag verwendet:

- **merge** = 0, TLGen generiert diese Klasse immer wieder neu
- **merge** = 1, diese Klasse wird generiert nur wenn sie nicht existiert
- **merge** = 2, wird neu generiert und der manuell erstellte Code mit dem neu generierten Code gemergt

3. Vergleich zwischen TLGen und anderen Code Generatoren

EJB (Enterprise Java Beans) sind standardisierte Komponenten innerhalb eines Applikation Servers, mit dem Ziel, komplexe, mehrschichtige sowie verteilte Software Systeme mit der Programmiersprache Java zu vereinfachen.

Die erste Spezifikation von EJB wurde von IBM im Jahr 1997 durchgeführt und von Sun Microsystems mit der Bezeichnung EJV 1.0 und 1.1 angenommen. Kurz danach erschienen die ersten Applikation Server „WebSphere“ von IBM und „WebLogic“ von BEA auf dem Markt.

Das EJB-Konzept sollte die Softwareentwicklung von komplexen verteilten Applikationen stark vereinfachen. Das ist aber nur teilweise gelungen, da der persistente Teil (auf Basis von Entity Beans) erhebliche Performance-Verluste verbuchte. Aus diesem Grund sind eine Reihe von Applikationen als „Verbesserungs-Tools“ (z.B. „Hibernate“, „iBatis“, „TopLink“ etc) entwickelt worden, hauptsächlich um diesen Zustand, zu korrigieren.

Version 2 von EJB verbessert diesen Zustand wesentlich, eine bessere Performance ist vorhanden. Aber geblieben ist eine gewisse Schwierigkeit durch sehr komplexe und aufwendig zu entwickelnde XML Dateien, notwendig für das Deployment von Applikation. Als Hilfe dafür ist das Tool „XDoclet“ gedacht, welches mit Hilfe von eigenen Tags relativ einfach diese komplexe XML Dateien generiert.

Ab Version 3.0 und 3.1 von EJB sind alle Probleme von EJB sehr gut gelöst worden. Eine vernünftige Verwendung der Annotationen (neue Features von Java) lässt die komplette Generierung des notwendigen Java Codes, nur auf Basis von Domainmodell oder Datenmodell, zu. Die Generierungs-Steuerung (die nicht sehr umfangreich ist) übernimmt eine Konfigurations-Datei.

Tools wie Hibernate sind für EJB 3 eigentlich nicht mehr notwendig, sie verkomplizieren sogar die gesamte Applikation. Diese Applikationen bergen folgende Nachteile:

- Die Applikation muss in der Running Zeit diese Tools in der Applikation einbeziehen:
 - das verkompliziert die Applikation
 - veranlasst eine Tool-Versions-Abhängigkeit sowie
 - Miteinbeziehung der Tool-Fehler
- Verwendet eine extra Code-Schicht als von EJB3 vorgesehen ist
- All diese Tools bringen nicht eine komplette neue Lösung in der Verwendung von Datenbanken mit sich, sondern sind nur eine Extraschicht für einen Applikation Server, um die Entwicklung mit allen diesen Nachteilen zu erleichtern. Eine reine Nutzung von EJB3 ohne zusätzliche Tools (wie z.B. Hibernate, Toplink, etc.) ist im direkten Vergleich zur Tool-Nutzung immer im Vorteil (siehe auch Anhang 1 - folgt in Kürze).

Die Verwendung der Tools (wie z.B. Hibernate, TopLink, iBatis, etc.) mit einem EJB3 Applikation Server ist weder erforderlich noch empfehlenswert, es verkompliziert nur das IT-Programm ohne dabei Vorteile zu erbringen. So ein Tool ist nur für kleine Applikationen, die keinen Applikation Server benötigt, zu gebrauchen.

In Gegensatz zu diesen Tools (Hibernate, TopLink, iBatis, etc.) bringt TLGen durch eine komplette Generierung des notwendigen Java Codes eine wesentliche Vereinfachung in der Verwendung von EJB3, so dass sich die Entwicklung nur auf die Fachlogik einer Applikation konzentrieren kann. Die Verwendung von TLGen bringt folgende Vorteile:

- Komplette Code-Generierung: Test Klassen, Client Technische Klassen, Session Beans, Manager Beans, Entity Beans, „persistence.xml“ Dateien, Daten Klassen/Interfaces und Interceptor Klassen
- TLGen beteiligt sich nicht bei der Applikation in Running Zeit, d. h. keine Abhängigkeit von der TLGen Version
- Der von TLGen generierte Code hat das gleiche Layout wie der Code, welcher ein Programmierer selber schreiben könnte
- Durch die sofortige Code-Generierung mit Test Klassen, können die Designer und Architekten die Ergebnisse von Konzepten sofort ansehen und eventuell optimieren (siehe Kap. 2.1)

4. TLGen Generierung

4.1 Was wird generiert

4.2 Wie wird Code generiert?

TLGen generiert den Java Code mit Hilfe eines Domain Models (UML) für neue Projekte, einer existierenden Datenbank für Legacy Projekte und zwei Konfiguration-Dateien. Spezifisch für TLGen sind nur die *Konfiguration-Dateien* im XML Format, eine Default-Konfigurationsdatei und die projektspezifischen Konfigurationsdateien.

Der Aufwand zur Erstellung der projektspezifischen Konfigurationsdatei ist sehr gering, da üblicherweise nur ein paar Parameter, abhängig vom eigenen Projekt, angepasst werden müssen. Die restlichen Parameter werden automatisch von den Default-Werten übernommen (siehe Kap 4.3 und 4.4).

Diese Default-Werte können durch die Default *Konfiguration-Datei* überschrieben, im Generierungsprozess verwendet werden und gelten dann für alle projektspezifischen Konfigurationsdateien.

Default-Werte können in der projektspezifischen Konfigurationsdatei überschrieben werden.

Diese Informationen-Verkettung für den Generierungsprozess ist für Firmen gedacht, welche viele Projekte haben und bestimmte Standard-Werte für alle Projekte benötigen, z.B. Layout des generierten Codes. Die Unterschiede zwischen Projekte (fachprojektspezifisch) werden mit Hilfe der projektspezifischen Konfigurations-Datei geregelt. So kann die Default Konfigurations-Datei nur allgemeine Informationen beinhalten, welche alle untergeordneten Projekte dann verwenden.

Für Firmen, die wenige Projekte entwickeln, ist die detaillierte Steuerung von Projekt-Hierarchien (mit der *Default-Konfigurationsdatei* und *mehreren Projekt-Konfigurationsdateien*) für die Code- Generierung uninteressant und kann deshalb entfallen.

4.3 Ordnerstruktur für den generierten Code

Eine übersichtliche und fachgerechte Strukturierung von Ordnern für den neu generierten Code vereinfacht die Wartung und Weiterentwicklung von neuen oder refaktorierten Projekten.

Für den neu generierten Code schlagen wir folgende Ordner-Struktur vor (siehe Abbildung 12).

Für den generierten Code gibt es für die Pfad-Struktur „de.stardata.demo“ (mit dem Projektnamen „demo“) folgende drei Unterordner:

- **bussines** Ordner, in welchem die Daten-Objekte generiert werden
- **client**, der Code für die Client-Aufrufe:
 - **bci** (business common interface), Ordner, wo die Verbindungszugriffe zwischen Client und Server generiert werden
 - **message** Ordner, wo die Client Message Klassen generiert werden sollten
 - **test** Ordner, wo die JUnit-Test Klassen generiert werden
- **server** Ordner, wo Server Klassen generiert werden
 - **message** Ordner für die Message Beans

- **persistence** Ordner, wo sich alle Klassen, notwendig für die Daten-Persistence, befinden wie die Session-Beans, Manager Beans, Entity Beans, etc.

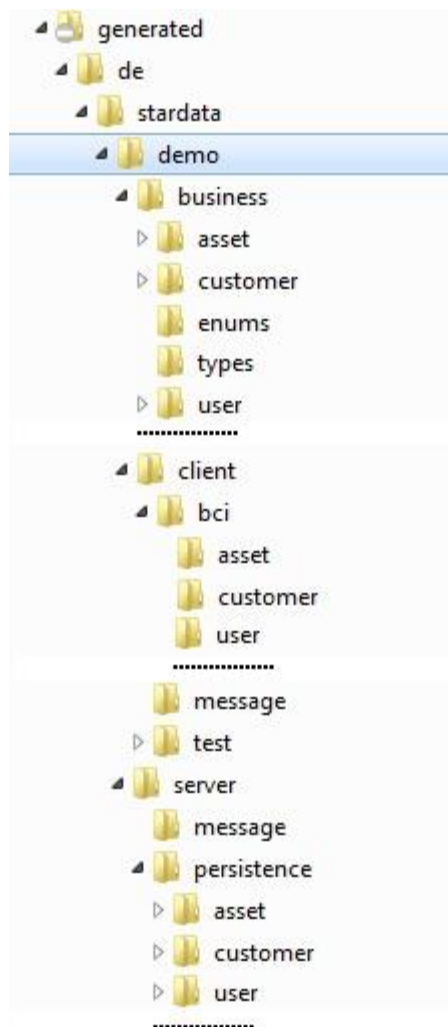


Abbildung 12: Ordner-Struktur

Diese Ordner-Struktur ist nur eine Empfehlung vom TLGen Generator Team, denn eigentlich kann jeder die eigene Struktur über die Konfigurations-Datei definieren.

4.4 Default Konfigurations Datei

Wie zuvor schon beschrieben, beinhaltet diese *Konfiguration-Datei* die allgemeinen Informationen, die in mehreren Projekten verwendet werden können.

In den folgenden Kapiteln werden die Parameter und Einstellungen genauer beschrieben.

Für die Code-Generierung mit Hilfe von TLGen aus einem Domain-Model (UML) oder aus einer vorhandenen Datenbank (z.B. für Legacy-Projekte) heraus gibt es nur einen Unterschied, nämlich das Setzen eines Parameters.

4.4.1 Default Konfigurations-Datei - Struktur

Die Default-Konfigurationsdatei hat folgende XML-Struktur (siehe 32 Listing):

```
<?xml version="1.0" encoding="UTF-8"?>
<Standard xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\Project-TLGen-2.2\config\template\DefaultSchema.xsd">
  <Project/>
  <Locator></Locator>
  <Class>
    <Extends/>
  </Class>
  <Entity>
    <Implements/>
    <Extends/>
    <Annotation/>
  </Entity>
  <Manager>
    <Extends/>
    <Implements/>
    <Annotation/>
    <Method>
      <Exception/>
      <Parameter/>
    </Method>
  </Manager>
  <Session>
    <Extends/>
    <Annotation/>
    <Client>
      <Extends/>
      <Annotation/>
      <Variable/>
    </Client>
    <Xml-persistence>
      <Property/>
    </Xml-persistence>
    <Interceptor>
      <Method>
        <Annotation/>
        <Parameter/>
        <Exception/>
      </Method>
    </Interceptor>
    <Freeclass>
    </Freeclass>
  </Session>
  <Message>
  </Message>
  <UmlControl>
    <Column/>
    <ColType/>
    <NameChange/>
    <Relation/>
    <Association/>
    <RelationType/>
    <AggregationType/>
    <DirectionType/>
    <AssociationType/>
    <AssociationSubType/>
    <Navigability/>
    <Compare/>
  </UmlControl>
</Standard>
```

32 Listing

Die Default-Konfigurationsdatei verwendet folgende Tags:

- Standard Tag, ist der Main Tag.
- Locator
- Class
- Entity
- Manager
- Session
 - Client
 - Xml-persistence
 - Interceptor
 - Freeclass
- Message
- UmlControl

Die detaillierte Beschreibung dieser Tags und dessen Attribute sind im folgenden Kapitel dargestellt.

4.4.2 Default Konfigurations-Datei - Beschreibung

Die Default-Konfigurationsdatei beinhaltet allgemeine Einstellungen und Parameter und ist für eine Gruppe von Projekten gedacht, welche in der Projekt Konfigurationsdatei überschrieben werden können.

Bemerkung: Strings in den Attributen innerhalb eines Tags welche mit einem „%“ beginnen und enden (z.B. %package%) sind Ersetzungsnamen oder Platzhalter, welche der Generator dann automatisch ersetzt (z.B. alle Entities mit Ihren richtigen *packages*).

Des Weiteren sind alle Werte eines Tag-Attributs, wie **name**=„com.tlgen.common.bci.ServiceLocator“ (hier eine Klasse), nur Beispiele, sie müssen aber nicht verwendet werden. TLGen wurde mit diesen getestet und entsprechen auch den Standard-Aufrufen.

4.1.1.1 <Standard> Tag

Ist der Main Tag für diese XML. Es gibt folgende Attribute:

- **name**: Projekt Name
- **company**: Firmen-Name
- **Commentary**: Kurzbeschreibung des Projekts

4.4.2.2 <Project> Tag

Siehe Kapitel 4.5.2.2.

4.4.2.3 <Locator> Tag

Ist ein optionaler Tag. Dieser Tag beinhaltet die Klasse für die RMI-Verbindung. Gibt es keine Eintragungen in diesem Tag, wird die Default-Klasse verwendet. Folgende Attribute werden mit den Default-Werten verwendet:

- **name** = „com.tlgen.common.bci.ServiceLocator“, Standard Locator-Klasse
- **instance** = „getInstance“, ist der Methoden-Name für die Instanz
- **local** = „getLocalReference“, Methoden-Name des lokalen RMI-Aufrufs
- **remote** = „getRemoteReference“ Methoden-Name des Remote-Aufrufs

4.4.2.4 <Class> Tag

Dieser ist ein optionaler Tag und kann auch über die projektspezifische Konfigurationsdatei über die <Class>-Tags überschrieben werden. Der Tag generiert Daten-Klassen und Ihre Interfaces. Dazu gehören folgende Attribute:

- **path** = „de.stardata.demo.business“, Klassen/Interface-Pfad und optionales Attribut
- **name** = "de.stardata.demo.business.%package%.data.%classname%Data|de.stardata.demo.business.%package%.dataif.%classname%DataIf", der Name beinhaltet zwei Teile, für die Klassen bzw. für die Interfaces, getrennt durch "|". Die Attribute der zwei Platzhalter sind:
 - **%package%**, ist der Package-Name. Dies wird vom Generator mit dem Package Namen aus dem Domain Modell ersetzt oder aber aus den festgelegten Namen aus der projektspezifischen Konfigurationsdatei.
 - **%classname%** dieses Attribut wird automatisch vom Generator mit dem Objekt- Namen aus dem Domain Modell ersetzt oder mit den Werten der Tabellen-Namen (eventuell durch die Rules geändert) aus der Datenbank.
- **pathEnum** = "de.stardata.demo.business.%package%.%classname%", wird für die Generierung von Enums verwendet. Die Platzhalter haben die gleiche Bedeutung wie zuvor erwähnt.
- **pathType** = "de.stardata.demo.business.%package%.%classname%", wird für die Typen Klassen verwendet.

Dieser Tag hat zwei Kind-Tags, die optional sind und folgende Standard-Einstellung haben:

- **<Extends name="com.tlgen.common.data.BaseData"/>**, kann eigene Super- Klassen verwenden.
- **<Extends name="com.tlgen.common.data.BaseDataIf" type="interface"/>**

4.4.2.5 Entity> Tag

Dieser Tag versorgt den Generator mit Informationen über die Entities. Es ist ein optionaler Tag. Fehlt dieser, werden die Default-Informationen (z.B. mit Hilfe des Domain Modells) verwendet oder können über die projektspezifische Konfigurationsdatei überschrieben werden.

Entity Tag hat folgende Attribute:

- path = "de.stardata.server.persistence" (Beschreibung wie unter 4.4.2.4)
- name = "de.stardata.server.persistence.%package%.entity.%classname%Entity" (Beschreibung wie unter 4.4.2.4)
- seqstrategy="SEQUENCE" ist der Default-Wert. Dieses Attribut ist für die automatische Generierung von einem Primary Key notwendig (z.B. für die Datenbank Oracle).

Dieser Tag hat folgende Kinder-Tags, die optional sind:

- <Implements> mit Attribut *Name="java.io.Serializable"*
- <Extends> mit Attribut *Name="com.tlgen.common.ejb.entity.BaseEntityBean"*, z.B. eine Standard Super-Klasse wie
<Extends name="com.tlgen.common.ejb.entity.BaseEntityBean"/>
- <Annotation>, dieser Tag ist optional und der Generator kann damit folgende Annotationen in die Entities generieren:
 - *"javax.persistence.Entity"*
 - *"javax.persistence.Table"* der Generator übernimmt automatisch den Tabellen-Namen
 - *"javax.persistence.SequenceGenerator"*

4.4.2.6 <Manager> Tag

Dieser Tag versorgt den TLGen Generator für die Generierung mit Informationen von Manager Beans und hat folgende Attribute:

- name="eu.stardata.server.persistence.%package%.manager.%classname%Manager" (Beschreibung wie unter 4.4.2.4)
- transactiontype: z.B. "JTA", ist ein Standard Wert.
- Exception: Zur Verwendung einer eigenen Exception-Klasse, z.B. "com.tlgen.common.exception.PersistenceException".

Dieser Tag hat folgende Unter-Tags, die optional sind:

- <Extends> mit Attribut *Name=" com.tlgen.common.ejb.manager.BaseManager "*
- <Annotation>, dieser Tag ist optional und folgende Annotationen sind z. B. für die Entities möglich:
 - *"javax.ejb.Stateless"*
 - *"javax.ejb.Local"*

4.4.2.6.1 Fields für die Manager-Beans

Für den Manager werden folgende notwendige Fields automatisch generiert (für den Tag **<Variable>** siehe auch 4.4.2.11):

1. name="JNDI_NAME" vartype="string" content="%interfacemanager%" fieldtype="1" abstract="true"
2. name="EAR_NAME" varitype = "string" content="%earname%" fieldtype="1" abstract="true"
3. name="m_manager" vartype = "javax.persistence.EntityManager" type="%clientdata%.MANAGER_NAME". Für dieses Field werden folgende Annotation generiert:
 - a. name="javax.persistence.PersistenceContext", mit folgenden Parametern:
 - i. name="properties" type="Text-Prop"/>
 - ii. name="unitName" type="%clientdata%.MANAGER_NAME"

4.4.2.6.2 Methoden für Manager

Für den Tag **<Method>** siehe auch 4.4.2.10. Die Manager Bean-Methoden können in drei Kategorien eingeteilt werden: Es sind Methoden, die intern vom Manager Bean gebraucht werden, Methoden, die vom Client über die Session-Beans aufgerufen werden und für die Verwaltung des Persistenz-Prozesses notwendig sind und als dritte Kategorie sind die Methoden für die Persistenz-Verwaltung, welche vom User über die Konfigurationsdatei übergeben werden (siehe 4.5.2.4.3.1).

Die Methoden im Manager können auch eine Exception werfen(??) und werden mittels Tags **<Exception>** individuell gesteuert. Dieser Tag hat ein Attribut „name“ und braucht den kompletten Namen der Exception-Klasse.

Folgende Interne Methoden werden für den Manager Bean standardmäßig generiert:

1. name="getManager" return="%interfacemanager%" methodtype="4", holt vom Bean-Container den aktuellen Manager.
2. name="getEntityByPrimarykey" return="%Klassentity%" methodtype="5"

Folgende Methoden werden standardmäßig für die Remote-Verwendung generiert:

1. name="create" return="%interfacedata%" methodtype="4", speichert ein neues Objekt in der Datenbank ab. Der Platzhalter %interfacedata% wird vom Generator automatisch mit dem richtigen Interface Namen ersetzt.
2. name="update" methodtype="4", ändert ein Datenobjekt in der Datenbank
3. name="remove" methodtype="4", löscht ein Objekt in der Datenbank
4. name="flush" methodtype="4"
5. name="close" methodtype="4"
6. name="clear" methodtype="4"
7. name="findByPrimarykey" return="%interfacedata%" methodtype="5", sucht ein Objekt in der Datenbank nach dessen PrimaryKey. Der Platzhalter %interfacedata% wird vom Generator mit dem Interface Name ersetzt.

8. `name="findAll"` `return="%interfacedata%[]"` `methodtype="5"`, holt alle Objekten für diesen Manager-Bean aus der Datenbank. Der Platzhalter `%interfacedata%` wird vom Generator mit dem Interface Name ersetzt.

4.4.2.7 <Session> Tag

Dieser Tag versorgt den Generator mit allgemeinen Informationen für die Generierung der Session-Beans. Falls dieser nicht vorhanden ist, werden die Default-Daten für die Generierung verwendet.

Der Session-Bean Tag hat folgende Attribute:

- `name="de.stardata.demo.server.persistence.%package%.%classname%SessionBean"`, ist der Name (`%classname%`) und dazu der Pfad –Platzhalter (`%package%`) für alle Session Beans.
- `managername="manager%classname%"`, gibt Informationen dem Manager, die für die „`persistence.xml`“ Datei notwendig sind (siehe 4.4.2.7.4)
- `transactiontype`, Transaktions-Typ (z.B. JTA).

Session Bean hat z.B. folgende mögliche Annotationen:

- `name="javax.ejb.Stateless"`, oder „`javax.ejb.Stateful`“
- `name="javax.ejb.Remote"`

Werden Session-Beans als *Webservices* oder *Timerservices* generiert, können auch andere Annotationen verwendet werden (siehe 4.5.2.4.4.4, 4.5.2.4.4.5)

4.4.2.7.1 Methods für Session

Ein Session-Bean übernimmt alle oder nur die auserwählten Methoden von den Manager Beans, aufgerufen über den jeweiligen Session-Bean (Details siehe 4.4.2.6.2).

Die Namen der Methoden werden aus dem Namen der Methode und dem Manager-Namen zusammengesetzt, da z.B. der Aufruf „`findByPrimaryKey`“ für jedes Entity nützlich ist und von mehreren Managern benötigt wird.

Z.B. das Manager *Formular* hat im *Session-Bean* den zusammengesetzten Aufrufnamen aus „`findByPrimaryKey`“ und seinem Namen: „`findByPrimaryKeyFormular`“.

Diese Namen werden bis zum Client gleichbleibend verwendet, d.h. auch in den Client-Interfaces/Daten-Klassen (DAOs).

4.4.2.7.2 <Client>

Dieser ist ein Tag innerhalb der Session-Beans und dient zur Generierung von Client-Aufrufen. Auch dieser Tag ist optional, wird er nicht verwendet, generiert der Generator Standard-Aufrufe.

Dieser Tag hat folgende Attribute:

- `name="de.stardata.demo.client.bci.%package%.%classname%Bci"`
- `exception = "com.tlgen.common.exception.PersistenceException"`

4.4.2.7.3 <Xml-persistence>

Der Tag setzt die *Properties* der *persistence.xml*, die z.B. in einer EAR benötigt werden, um die eigene Applikation in einem *Application Server* zu installieren. Diese Werte sind Applikation-Server abhängig. Statt dem Attribute *value* wird hier *type* für <Property> verwendet.

Beispiel: *JBoss 5.x* brauchte bei Verwendung der Oracle-Datenbank 11.x die folgende Information, um eine Verbindung zur Datenbank aufzubauen:

```
<Property name="hibernate.dialect" type="org.hibernate.dialect.Oracle10gDialect" />
```

4.4.2.7.4 <Interceptor>

EJB3 bietet die Möglichkeit eigene *Interceptors* zu verwenden. Folgende Attribute können für diesen Tag verwendet werden:

- *name*="de.stardata.demo.server.persistence.%package%.TimeCore", kompletter Klassen-Name des Interceptors (Pfad)
- *path*="C:/Project-TLGen-2.2/source/generated", ist der Pfad, wohin(??) die Klasse generiert wird. Ist dieser nicht vorhanden, wird der Interceptor in denselben Pfad wie der Session-Bean hinein generiert.

Ein Interceptor kann mehrere Methoden mit folgenden Daten haben:

- *name*="timeTrace", interceptor name
- *return*="java.lang.Object" , Rückgabewert
- *finally*="C:/Project-TLGen-2.2/resources/FinallyTime.xml"

4.4.2.8 <Messagedriven> Tag

Dieser wird für Message-Driven-Beans verwendet und hat folgendes Attribut:

- *name*="de.stardata.demo.server.persistence.%package%.message"

4.4.2.9 <UmlControl> Tag

Dieser Tag wird für die Generierung der Klassen aus den UML-Domain-Daten-Modellen benötigt. Da dies feste Einstellungen sind und z.B. die Standard UML-Relationen der Klassen festlegen, sollten diese Werte nicht geändert werden.

4.4.2.10 <Method> tag

Der Element <**Method**> hat folgende Attribute:

- *name*, ist der Methode-Name
- *return*, legt den Rückgabewert der Methode fest. Dieser kann neben den Standardtypen wie String, int, long, boolean etc. auch ein Array der Interfaces der Aufrufenden-Klasse sein, z.B. *%interfacedata%[]* (TLGen ersetzt den Platzhalter automatisch).

- **manager**, ist der Manager-Bean Name
- **client**, ist der Client Name (nur für die Test-Klassen) über welchen diese Methode aufgerufen werden kann.
- **typerwd**, ist für den Datenbankzugriff-Typ (nur für die Test-Klassen), dieses Attribut kann folgende Werte haben: 0 für die Abspeicherung eines Objekts in der Datenbank, 1 für das Lesen, 2 für das Ändern und 3 für das Löschen.
- **methodtype**, ist eine Zahl und gibt dem Generator Informationen über Methode-Type. Folgende Typen sind erlaubt:
 - ***METHOD_TYPE_ERROR = -1; method build is not possible***
 - ***METHOD_TYPE_FREE = 0; free methods, SQL free methods***
 - ***METHOD_TYPE_COLUMN = 1; methods generate from columns***
 - ***METHOD_TYPE_RELATION = 2; methods generate from relation***
 - ***METHOD_TYPE_CLASS = 3; from configuration file (user method)***
 - ***METHOD_TYPE_STANDARD = 4; standard method***
 - ***METHOD_TYPE_SQL_STANDARD = 5; Query SQL-EJB Format (Default, from config-default XML, Methods createQuery)***
 - ***METHOD_TYPE_SQL_EJB = 6; Query SQL-EJB Format (from config XML file, Method createQuery())***
 - ***METHOD_TYPE_SQL_NAMED = 7; Query SQL-Named (Method createNamedQuery(), EJB named query type)***
 - ***METHOD_TYPE_SQL_NATIVE = 8; Query SQL-Native (Method createNativeQuery(), EJB native query type)***
 - ***METHOD_TYPE_SQL_OBJECT = 9; Query SQL method over "QueryObject" for store procedure***
 - ***METHOD_TYPE_SQL_MAPPINGS = 10; annotations for SQL Mapping in entity beans***
 - ***METHOD_TYPE_PRIMKEY = 11; method for primary key***
 - ***METHOD_TYPE_CONSTRUCTOR = 12; method for class constructor***
 - ***METHOD_TYPE_HELPER = 13; helper method***
 - ***METHOD_TYPE_FACTORY = 14; factory method***

4.4.2.11 <Variable> Tag

Das Element <**Variable**>. hat folgende Attribute:

- **name**, Name des Elements
- **vartype**, definiert den Field-Typ
- **content** = Ein „Platzhalter“, welcher durch den Generator mit den notwendigen Informationen ersetzt wird.
- **fieldtype**, mögliche Werte 0,1 oder 2. 0= Field wird in eine Klasse generiert, 1 in dessen Interface und 2 in beide. 0 ist Standard-Wert.
- **abstract** = true oder false (true ist der Standard-Wert) und ist nur intern für den Generator; falls *Field-Content* eine Klasse ist, wird der Klassen-Pfad im ***import***-Bereich der Klasse hinein generiert.

4.5 Projekt Konfigurationsdatei

Die Projekt Konfigurationsdatei gibt dem TLGen-Generator projektspezifische Daten über das Projekt. Die Daten aus der Konfigurationsdatei überschreiben die vorhergehenden Daten, die Default-Werte sowie die aus der Standard-Konfigurationsdatei.

4.5.1 Projekt Konfiguration Datei - Struktur

Die Default Konfiguration-Datei hat folgende XML Struktur:

```
<?xml version="1.0" encoding="UTF-8"?>
<Generator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\Project-TLGen-2.2\config\template\ConfigSchema.xsd">
  <Project>
    <Design>
      <Commentary/>
      <Class>
        <Extends/>
      </Class>
      <Entity> </Entity>
      <Manager>
        <Extends/>
        <Implements/>
        <Method>
          <Exception/>
          <Parameter/>
          <Sql/>
        </Method>
      </Manager>
      <Session>
        <Extends/>
        <Annotation/>
        <Client>
          <Extends/>
          <Annotation/>
          <Variable/>
        </Client>
        <Timerservice>
          <Extends/>
          <Annotation/>
          <Variable/>
          <Method/>
        </Timerservice>
      <Xml-persistence>
```

```

        <Property/>
    </Xml-persistence>
    <Method>
        <Exception/>
        <Parameter/>
    </Method>
</Session>
<Messagedriven>
    <Extends/>
    <Annotation/>
    <Client/>
    <Bean>
        <Callback/>
    </Bean>
</Messagedriven>
<Test>
    <Extends/>
    <Import/>
    <Method>
        <Exception/>
        <Parameter/>
    </Method>
</Test>
<Dbtable>
    <Relation>
        <JoinColumn/>
        <JoinTable>
            <JoinColumn/>
            <InverseJoinColumn/>
        </JoinTable>
    </Relation>
</Dbtable>
</Design>
</Project>
</Generator>

```

33 Listing

4.5.2 Projekt Konfigurationsdatei - Beschreibung

Die Konfigurationsdatei erlaubt detaillierte Einstellungsmöglichkeiten und kann alle vorherigen Daten überschreiben.

4.5.2.1 <Generator> Tag

Dieser ist das Hauptelement und versorgt den Generator mit allgemeinen Informationen über die Code-Generierung. Folgende Attribute können verwendet werden (siehe Tabelle 7):

Tabelle 9: Attribute für den Generator Tag

Nr.	Attribute Name	Pflichtfeld	Beispiel-Wert	Kommentar
1	name	no	Demo	Project name
2	inputsource	yes	database	
3	datasource	yes	java:/efpPool	Conditioned from App.Server
4	appserver	yes	JBoss	
5	standardConfigFile	yes	C:/Project-TLGen-2.5/config/config_tlgen_demo_default.xml	
6	umlFile	yes/no	C:\Project-TLGen-2.5\doc\model\DomainModel-Demo-01.xml	Is mandatory only for type= guml and is the XMI file from the UML domain model
7	databaseConnect	yes	jdbc:oracle:thin:@localhost:1521:orcl	Connect string to the database
8	database	yes	ORACLE	Database name
9	userPwd	yes	demo/demo	For access to the database
10	type	yes	guml/gdb	guml - generates from a domain model - gdb - generates from a database.
11	mapping	no	true	
12	rule	no	3	Default is 0, siehe 4.5.2.1.1
13	databaseScript	yes/no	C:/Project-TLGen-2.5/doc/db/demo_schema.sql	Is mandatory for makeDatabase=true and type=guml
14	makeDatabase	no	true	Generates a new Database with the help of a generated SQL-script
15	indexTablespace	yes/no	DEMOINDEX	Mandatory only if generating a database
16	dataTablespace	Yes/no	DEMODATA	Mandatory only if generating a database
17	makeJavaCode	no	true	
18	initProgram	no		
19	keyName	no		
20	reference	no	true	

4.5.2.1.1 Regel für Namen-Transformation in Klassen, Methoden und Tabellen

Mit TLGen kann folgende Regel für die Namen-Generierung verwendet werden:

- **RULE_DEFAULT_VALUE = 0**, Default value, no underscore, if all is upper, then first char is upper, rest is lower and the rest is not changed. Examples:
 - `'CORE_FORMULAR_PAGE_FIELD'` -> `'CoreFormularPageField'`
 - `'bruttoNettoSpecification'` -> `'BruttoNettoSpecification'`
 - `'_FIELDLONGDATA_'` -> `'Fieldlongdata'`
- **RULE_NO_UNDERSCORE = 1**, No underscore. Examples:
 - `'CORE_FORMULAR_PAGE_FIELD'` -> `'COREFORMULARPAGEFIELD'`
 - `'bruttoNettoSpecification'` -> `'bruttoNettoSpecification'`
 - `'_FIELDLONGDATA_'` -> `'FIELDLONGDATA'`
- **RULE_NO_UNDERSCORE_FIRST_UPPER = 2**, No underscore, first char is upper, the rest is not changed. Examples:
 - `'CORE_FORMULAR_PAGE_FIELD'` -> `'COREFORMULARPAGEFIELD'`
 - `'bruttoNettoSpecification'` -> `'BruttoNettoSpecification'`
 - `'_FIELDLONGDATA_'` -> `'FIELDLONGDATA'`
- **RULE_DEFAULT_WITHOUT_FIRST = 3**, Default, without the first tail before the first underscore (for legacy programs). Examples :
 - `'CORE_FORMULAR_PAGE_FIELD'` -> `'FormularPageField'`
 - `'bruttoNettoSpecification'` -> `'BruttoNettoSpecification'`
 - `'_FIELDLONGDATA_'` -> `'Fieldlongdata'`
- **RULE_NO_UNDERSCORE_FIRST_UPPER_REST_LOWER = 4**, No underscore, first char is upper, rest is lower. Examples :
 - `'CORE_FORMULAR_PAGE_FIELD'` -> `'CoreFormularPageField'`
 - `'bruttoNettoSpecification'` -> `'Bruttonettospecification'`
 - `'_FIELDLONGDATA_'` -> `'Fieldlongdata'`
- **RULE_NO_UNDERSCORE_FIRST_UPPER_ALL_REST_LOWER = 5**, No underscore, first char is upper, all rest is lower. Examples:
 - `'CORE_FORMULAR_PAGE_FIELD'` -> `'Coreformularpagefield'`
 - `'bruttoNettoSpecification'` -> `'Bruttonettospecification'`
 - `'_FIELDLONGDATA_'` -> `'Fieldlongdata'`
- **RULE_NO_CHANGE = 6**, No change

4.5.2.2 <Project> Tag

Dieses Element versorgt den Generator mit den projektspezifischen Daten. Die Attribute für diesen Tag sind in der Tabelle 8 aufgeführt.

Tabelle 10: Attribute für den Projekt-Tag

Nr.	Attribute Name	Pflichtfeld	Beispiel-Wert	Kommentar
1	name	no	Project StarData Demo	Project name
2	initialcontext	yes	jnp://localhost:9099	Conditioned from the App.Server
3	initialcontextfactory	yes	org.jnp.interfaces.NamingContextFactory	Conditioned from the App.Server
4	initialcontextpkgprefix	yes	org.jboss.naming:org.jnp.interfaces	Conditioned from the App.Server
5	managermethods	no	Asset, Product, etc.	List with all managers
6	methods	no	public	
7	sequencegen	yes	SEQ_STORE	Type from generator for primary key
8	format	no	0 or 1	Default is 0 (see 4.5.2.2.1)
9	import	no	true	Default is 'false'. Do not use import in the class and all name is full qualified
9	data	no	false	Default is 'false'. TLGen generates classes and interfaces data by default, if 'true', then only data classes.
10	list	no	true	Default is 'true', For Lists the class <i>List</i> is used, if 'false' arrays are used.
11	testRelation	no	true,true,true,true	Relations in Test Classes
12	earname	yes	demo	The name from ear file
13	MappedNameRemote	no	Examples in the text	Overwrites the standard generated attribute mappedname of the EJB3 annotation @Stateless/@Statefull in the Session-Beans
14	JNDIName	no	Examples in the text	Overwrites the standard generated attribute name of the EJB3 annotation @Stateless/@Statefull in the Session-Beans.
15	JNDINameRemote	no	Examples in the text	Overwrites the standard generated JNDI name for the Session-Beans and will be stored in the client Interface of the Session-Beans to call the remote Session-Bean.

Die JNDI-Namen der Session-Beans werden von TLGen anhand der Attribute *appserver*="{jboss|weblogic|websphere}" (Generator-Tag), *earname* (Project-Tag) sowie den Packagenamen automatisch zusammengestellt und in den Attributen *name* und *mappedName* der Annotationen *@Stateless*/*@Statefull* gespeichert. Der JNDI-Name des Session-Beans sowie, falls nötig, der Remote-JNDI-Name (jeder Application Server generiert beim „deployen“ andere Namen), werden in den Client-Interfaces der Session-Beans als statische Variablen gespeichert.

Z.B. reicht bei einem bestimmten Application Server nur der Interfacename des Session-Beans, um diesen lokal im Application Server zu finden. Für einen Remote-Aufruf ist aber der Name aus dem EAR-Namen, seinem Package-JAR, dem Session-Bean Namen und dem Interfacenamen nötig. Dies wird für die drei oben genannten Application Server automatisch generiert und mit der Klasse *ServiceLocator* aus den Commons Code-Beispielen automatisch bei einem Remote-Aufruf beachtet.

Doch die automatischen Namen können mit Hilfe der Attribute *MappedNameRemote*, *JNDIName* und *JNDINameRemote* des Projekt-Tags mit eigenen Definitionen überschrieben werden. Im Folgenden einige Beispiele womit der Generator die Ersetzungsnamen innerhalb von %...% ersetzt:

- `MappedNameRemote= "%earname%/jndiname%/remote%"`
- `JNDIName="de.stardata.test.server.persistence.%package%.%classname%SessionBean"`
- `JNDINameRemote="ejb/%earname%/package%.jar/interfacemanager%#clientdata%"`
- `JNDINameRemote="ejb/%earname%/package%.jar/%classname%SessionBean#clientdata%"`
- `JNDINameRemote="ejb/%earname%/package%.jar/de.stardata.test.server.persistence.%package%.%classname%SessionBean#de.stardata.test.client.bci.%package%.%classname%Bci"`

Ersetzungsnamen:

- `%interfacemanager%` entspricht hier folgender Ersetzung:
de.stardata.test.server.persistence.%package%.%classname%SessionBean
- `%interfacemanagershort%` entspricht hier folgender Ersetzung:
%classname%SessionBean
- `%clientdata%` entspricht hier folgender Ersetzung:
de.stardata.test.client.bci.%package%.%classname%Bci
- `%package%` oder `%classname%` wird aus den jeweiligen Designs und der Standard-Konfigurationsdatei bestimmt (siehe entsprechende Kapitel).

4.5.2.2.1 Code Format

TLGen generiert den Code in zwei Formate:

1. *format* = 0, Example:

```
„Method-Name“() oder „Class-Name”
{
  ...Code ...
}
```

2. *format* = 1, Example:

```
“Method-Name”() oder “Class-Name” {
  ...Code ...
```

}

4.5.2.3 <Locator> Tag

Diese Daten überschreiben die Daten von 4.4.2.4.

4.5.2.4 <Design> Tag

<Design>-Elemente sind eine logische Fachgruppe innerhalb eines Projektes und können beliebig viele sein. Eine Design-Komponente ist als eine fachspezifische logische Einheit gedacht. Kinder-Elemente können mehrere Session-Beans, Datenobjekte, Manager- und Entity-Beans sein. Der- <Design>-Tag hat folgendes Attribut:

- **name**, ist der Name für Design-Komponente

4.5.2.4.1 <Class>-Tag

Dieser Tag wird zur Generierung von Business-Daten-Klassen verwendet. Dessen Daten überschreiben die Daten von 4.4.2.4.

Ist ein optionales Element. Dieses Element generiert Daten-Klassen und Interfaces und hat folgende Attribute:

- **name** = "de.stardata.demo.business.product..data.%classname%Data|de.stardata.demo.business.product.dataif.%classname%DataIf" hat zwei Teile, einen für die Klassen und einen für die Interfaces, getrennt durch „|“ Für die automatischen Namen sind Platzhalter verwendet worden:
 - **%classname%** dieses Attribut wird automatisch vom Generator durch den Objekt Namen aus dem Domain Model ersetzt oder den Tabellen-Namen oder geänderten Tabellen-Namen (durch Rules), welche aus der Datenbank gelesen wurden (z.B. bei einem Legacy-Projekt).
- **type** = „public“, d.h. Klassen und Interfaces sollen **public** sein.

Dieser Tag hat zwei Kinder-Elemente-, die optional sind, und folgende Standard-Einstellungen:

- **<Extends name="com.tlgen.common.data.BaseData"/>**, kann eigene super Class verwenden.
- **<Extends name="com.tlgen.common.data.BaseDataIf" type="interface"/>**

4.5.2.4.2 <Entity> Tag

Dieser Tag wird verwendet, um den Generator mit Informationen über die Entities zu versorgen. Er ist optional und überschreibt die Daten von 4.4.2.5, falls vorhanden.

Der Entity-Tag hat folgende Attribute:

- **name** = "de.stardata.server.persistence.product.entity.%classname%Entity" (Beschreibung wie in 4.5.4.2.1)
- **seqstrategy**="SEQUENCE" ist Default-Wert. Dieses Attribut ist für die automatische Generierung von Primary Key notwendig (z.B. Oracle).

Dieser Tag hat folgende **Kinder-Tags**, die optional sind:

- **<Implements>** mit Attribut *name="java.io.Serializable"*
- **<Extends>** mit Attribut *name="com.tlgen.common.ejb.entity.BaseEntityBean"*
- **<Annotation>**, dieser Tag ist optional und der Generator generiert folgende Annotationen für die Entities:
 - *"javax.persistence.Entity"*
 - *"javax.persistence.Table"* der Generator übernimmt automatisch den Tabellen- Namen
 - *"javax.persistence.SequenceGenerator"*

Folgende Kinder-Tags sind möglich (mit Beispiel der Standard-Klasse):

<Extends name="com.tlgen.common.ejb.entity.BaseEntityBean"/>

4.5.2.4.3 <Manager>

Dieser Tag versorgt den TLGen Generator mit Informationen für die Generierung von Manager- Beans und hat folgende Attribute:

- *name="eu.stardata.server.persistence.product.manager.%classname%Manager"* (Beschreibung wie in 4.5.4.2.1)
- *transactiontype = "JTA"* (Standard-Wert)
- *exception = "com.tlgen.common.exception.PersistenceException"* (Standard-Wert)

Dieser Tag hat folgende Kinder-Tags, die optional sind:

- **<Extends>** (Standard-Attribut *name="com.tlgen.common.ejb.manager.BaseManager"*)
- **<Annotation>**, dieser Tag ist optional und der Generator generiert z.B. folgende Annotationen für die Entities:
 - *"javax.ejb.Stateless"*
 - *"javax.ejb.Local"*

Die Manager *Fields*-Beschreibung siehe unter 4.4.2.6.1

4.5.2.4.3.1 Methoden für Manager

Wie in 4.4.2.6.2 bereits erwähnt, gibt es drei Möglichkeiten, Methoden um die Manager zu generieren, auch die internen Methoden des Managers, Standard Remote-Methoden oder eine userdefinierte Methode für die Persistente-Datenverwaltung. Die ersten zwei sind in 4.4.2.6.2 beschrieben.

Die dritte Art von Manager Methoden sind die Methoden mit Methodentyp *methodtype=6, 7, 8 und 9* (siehe 4.4.2.11). Diese Methoden definieren ein SQL-Query und brauchen eine oder mehrere **<Parameter>**-Tags für die Beschreibung der Input-Daten sowie ein **<Sql>**-Tag.

Für diese Methoden sind folgende Attribute notwendig:

- ***name="NamedQuery01"***, ist der Methoden Name
- ***manager="Formular"***, benennt den Manager und wohin die Methode generiert wird.

- **mapping**="false", besagt, ob eine Mapping notwendig ist oder nicht (Default ist „false“, aber für die Native-SQLs ist fast immer eine Mapping nötig)
- **return**="java.util.List<%interfacedata%>", ist der erwartete Return Type. Dieser Platzhalter wird vom Generator durch das richtige Interface ersetzt.
- **methodtype**="6", 7, 8, oder 9.

Im Folgenden erfolgt die Beschreibung der Methoden-Typen.

4.5.2.4.3.1.1 Manager Method-Typ 6

Method-Typ 6 sind Methoden, die für EJB-SQL gedacht sind. Beispiel:

```
<Method name="findByNameAndOffer" parameter="java.lang.String" return="%interfacedata%[]"
manager="Product" methodtype="6">
    <Parameter name="name" fullName="name" type="java.lang.String" />
    <Parameter name="productOfferingId" fullName="productOfferingId" type="java.lang.String" />
    <Sql name="getNameAndOffer" sql="select u from ProductEntity u where u.name = :name and
u.productOfferingId = :productOfferingId" />
</Method>
```

32 Listing

In diesem Beispiel wird bzw. werden eine oder mehrere **Product** Objekt(e) nach dem Produkt-Namen und einer ID productOfferingId gesucht.

TLGen generiert daraus in den Manager-Bean der Klasse ProductManager ein Feld und eine Zugriffsmethode:

```
private String m_getNameAndOffer = "select u from ProductEntity u where u.name = :name and
u.productOfferingId = :productOfferingId";

/**
 * Manager method "findByNameAndOffer()"
 * @param name
 * @param productOfferingId
 * @return
 * @throws PersistenceException
 */
public ProductDataIf[] findByNameAndOffer(String name, String productOfferingId) throws
PersistenceException {
    try {
        ProductDataIf[] arrayData = null;
        List<?> list =
m_manager.createQuery(m_getNameAndOffer).setParameter("name",name).
        setParameter("productOfferingId",productOfferingId).getResultList();
        if(list != null && list.size() > 0) {
            arrayData = new ProductDataIf[list.size()];
            int idx = 0;
            for(Object entity : list) {
                if(entity != null) {
                    arrayData[idx++] = ((ProductEntity)entity).callProduct();
                }
            }
        }
        return arrayData;
    } catch(Exception ex) {
        throw new PersistenceException(ex.getMessage(),ex);
    }
}
```

}

33 Listing

4.5.2.4.3.1.2 Manager Methode-Typ 7

Methoden von Typ 7 sind für Named SQLs gedacht.

```
<Method name="NamedQuery02" manager="Formular" return="java.util.List<#60;%interfacedata%#62;"
methodtype="7">
  <Parameter name="developName" type="java.lang.String" />
  <Sql name="Formulartest" sql="SELECT f FROM FormularEntity f WHERE f.developer LIKE
:developName" />
</Method>
```

34 Listing

TLGen generiert daraus eine Methode in der Manager-Klasse (Listing 35) und eine Annotation `@javax.persistence.NamedQueries` in der dazugehörigen Entity-Klasse (Listing 36):

```
/**
 * Manager method "NamedQuery02()"
 * @param developName
 * @return
 * @throws PersistenceException
 */
public List<CoFormularDataIf> NamedQuery02(String developName) throws PersistenceException {
    try {
        List<CoFormularDataIf> list = null;
        List<?> entityList = m_manager.createNamedQuery("sqlFormulartest").
            setParameter("developName", developName).getResultList();
        if(entityList != null && entityList.size() > 0) {
            list = new ArrayList<CoFormularDataIf>();
            for(Object entity : entityList) {
                if(entity != null) {
                    list.add(((FormularEntity)entity).callFormular());
                }
            }
        }
        return list;
    } catch(Exception ex) {
        throw new PersistenceException(ex.getMessage(), ex);
    }
}
```

35 Listing

```
@NamedQueries
({
    @NamedQuery( name = "sqlFormulartest", query = "SELECT f FROM FormularEntity f WHERE
f.developer LIKE :developName")
})
```

36 Listing

4.5.2.4.3.1.3 Manager Methode-Typ 8

Methoden von Typ 8 werden für die Native SQL Queries verwendet. Ein Beispiel für den Eintrag in der Konfiguration-Datei befindet sich in Listing 37.

```
<Method name="NativeQuery02" manager="Formular" mapping="true"
return="java.util.List<#60:%interfacedata%&#62;" methodtype="8">
  <Sql sql="SELECT * FROM CORE_FORMULAR, CORE_PAGE WHERE CORE_FORMULAR.
FORMULAR_ID = CORE_PAGE.FORMULAR_ID AND CORE_FORMULAR.FORMULAR_ID = ?" />
</Method>
```

37 Listing

Der notwendige Parameter für diese Methode wird von TLGen automatisch aus den Datenbank-Informationen heraus generiert. TLGen generiert aus diesem Eintrag in der Konfigurationsdatei eine Entity-Klasse (Listing 38) und die dazu gehörige Daten-Klasse sowie deren Interfaces (Listing 39):

```
package eu.stardata.server.persistence.formular.entity;

import eu.stardata.business.form.dataif.CoPageDataIf;
import javax.persistence.NamedNativeQueries;
import javax.persistence.SqlResultSetMapping;
import eu.stardata.business.form.dataif.CoNativeQuery02DataIf;
import javax.persistence.NamedNativeQuery;
import javax.persistence.EntityResult;
import javax.persistence.SqlResultSetMappings;
import javax.persistence.Column;
import eu.stardata.server.persistence.formular.entity.FormularEntity;
import eu.stardata.server.persistence.formular.entity.PageEntity;
import eu.stardata.business.form.dataif.CoFormularDataIf;
import javax.persistence.Id;
import javax.persistence.Entity;

@Entity
@NamedNativeQueries
({
    @NamedNativeQuery( name = "NativeQuery02", query = "SELECT * FROM CORE_FORMULAR,
CORE_PAGE WHERE CORE_FORMULAR.FORMULAR_ID = CORE_PAGE.FORMULAR_ID AND
CORE_FORMULAR.FORMULAR_ID = ?", resultSetMapping = "NativeQuery02")
})
@SqlResultSetMappings
({
    @SqlResultSetMapping(name = "NativeQuery02",
        entities =
        {
            @EntityResult(entityClass = FormularEntity.class ),
            @EntityResult(entityClass = PageEntity.class )
        }
    )
})
public class NativeQuery02Entity {

    private static final long serialVersionUID = 597104846;

    // Entity data field
    private CoNativeQuery02DataIf m_nativeQuery02 = null;

    // constructors
    /**
     * Default Constructor
     */
    public NativeQuery02Entity() {
    }

    /**
     * Constructor
     * @param arg
     */
}
```

```

public NativeQuery02Entity(CoNativeQuery02DataIf arg) {
    m_nativeQuery02 = arg;
    fillNativeQuery02();
}

// Helper methods
/**
 * Helper Method "makeNativeQuery02()"
 */
public CoNativeQuery02DataIf makeNativeQuery02() {
    if(m_nativeQuery02 == null) {
        m_nativeQuery02 = new eu.stardata.business.form.data.CoNativeQuery02Data();
    }
    return m_nativeQuery02;
}

/**
 * Helper Method "fillNativeQuery02()"
 */
public void fillNativeQuery02() {
}

/**
 * Helper Method "addNativeQuery02()"
 */
public void addNativeQuery02() {
}

/**
 * Helper Method "callNativeQuery02()"
 */
public CoNativeQuery02DataIf callNativeQuery02() {
    // return the data object
    return makeNativeQuery02();
}

// Methods from class
public CoFormularDataIf getFormular() {
    return makeNativeQuery02().getFormular();
}

public void setFormular(CoFormularDataIf arg) {
    makeNativeQuery02().setFormular(arg);
}

public CoPageDataIf getPage() {
    return makeNativeQuery02().getPage();
}

public void setPage(CoPageDataIf arg) {
    makeNativeQuery02().setPage(arg);
}

@Id
@Column(name = "FORMULAR_ID")
public long getFormularId() {
    return makeNativeQuery02().getFormularId();
}

public void setFormularId(long arg) {
    makeNativeQuery02().setFormularId(arg);
}
}

```

38 Listing

```

package eu.stardata.business.form.data;

import eu.stardata.business.form.dataif.CoPageDataIf;
import eu.stardata.business.form.dataif.CoFormularDataIf;
import eu.stardata.business.form.dataif.CoNativeQuery02DataIf;
import com.tlgen.common.data.BaseData;

public class CoNativeQuery02Data extends BaseData implements CoNativeQuery02DataIf {

    private static final long serialVersionUID = 883471837;
    // Methods from columns
    private long m_formularId;
    // Methods from class
    private CoFormularDataIf m_formular;
    private CoPageDataIf m_page;

    // Methods from columns
    public long getFormularId() {
        return m_formularId;
    }

    public void setFormularId(long arg) {
        m_formularId = arg;
    }

    // Methods from class
    public CoFormularDataIf getFormular() {
        return m_formular;
    }

    public void setFormular(CoFormularDataIf arg) {
        m_formular = arg;
    }

    public CoPageDataIf getPage() {
        return m_page;
    }

    public void setPage(CoPageDataIf arg) {
        m_page = arg;
    }
}

package eu.stardata.business.form.dataif;

import eu.stardata.business.form.dataif.CoPageDataIf;
import eu.stardata.business.form.dataif.CoFormularDataIf;
import com.tlgen.common.data.BaseDataIf;

public interface CoNativeQuery02DataIf extends BaseDataIf {
    // Methods from columns
    public abstract long getFormularId();
    public abstract void setFormularId(long arg);

    // Methods from class
    public abstract CoFormularDataIf getFormular();
    public abstract void setFormular(CoFormularDataIf arg);

    public abstract CoPageDataIf getPage();
    public abstract void setPage(CoPageDataIf arg);
}

```

39 Listing

4.5.2.4.3.1.4 Manager Methode-Typ 9

Store Procedures sind von **TLGen** für die Version 3 vorgesehen.

4.5.2.4.3.2 Manager Interface

TLGen generiert für jede Manager-Klasse automatisch ein Interface, notwendig für die Verwendung von Manager-Beans (siehe 5.2).

4.5.2.4.4 <Session> Tag

Dieser Tag versorgt den Generator mit allgemeinen Informationen für die Generierung eines Session-Beans. Diese Daten überschreiben die Daten aus 4.4.2.7. TLGen kann in zwei Varianten verwendet werden:

1. Die Code Generierung aus einem UML Domain-Model ist besonders für neue Projekte geeignet. Um den Zeit- und Kosten-Aufwand zu minimieren, sollten für die Generierung die meisten Informationen über die Standard Konfigurations-Datei und nur die spezifischen Informationen wie Name, Native Queries und Test Klassen über die Konfigurationsdatei erfolgen.
2. Für die Code Generierung aus einer existierenden Datenbank ist die Verwendung der Konfigurationsdatei intensiver nötig als die der Standard Konfiguration-Datei, da die gelesenen Daten über die Tabellen aus der Datenbank auf fachlogische Kriterien über die Design-Session Beans verteilt werden müssen (siehe 5.2.4).

Das Session-Bean Element kann folgende Attribute verwenden:

- *name*="de.stardata.demo.server.persistence.product.ProductSessionBean", ist der vollständige Klassenname (Pfad und mit Platzhalter für alle Session Beans).
- *managername*="managerProduct", gibt Informationen dem *manager*, die für die „*persistence.xml*“-Datei notwendig sind (siehe 4.5.2.4.3).
- *transactiontype*="JTA" gibt die Information für die Transaction-Typ.

Session Bean hat z.B. folgende mögliche Annotationen:

- *name*="javax.ejb.Stateless", oder „*javax.ejb.Stateful*“
- *name*="javax.ejb.Remote"
- *name*="javax.interceptor.Interceptors", nur wenn die Session Bean eine Interceptor-Klasse verwenden.

Wenn die Session Bean als Webservice oder als Timer Service verwendet werden, können auch andere Annotationen verwendet werden (siehe 4.5.2.4.4.4, 4.5.2.4.4.5).

4.5.2.4.4.1 Methoden für Session

Session Bean übernimmt alle oder nur die auserwählten Methoden von Manager Beans, die über den Session-Bean aufgerufen werden (Details in 4.4.2.6.2).

Diese Methoden haben den Namen der Methode und dazu noch den Manager Name; z.B. die Methode „*findByPrimaryKey*“ für das Manager *Formular* hat den Namen

„*findByPrimaryKeyFormular*“. Diese Zusammensetzung von Namen ist notwendig, weil der Name „*findByPrimaryKey*“ in mehrere Manager-Beans, verwaltet über diese Session-Beans, vorhanden sein kann.

Dieser Name wird bis zum Client hindurch gleichbleibend verwendet.

4.5.2.4.4.2 <Client> Tag

Ist ein Kindelement der Session-Beans und dient zur Generierung von Client-Aufrufen. Auch dieser Tag ist optional.

Dieser Tag hat folgende Attribute mit folgenden Standard Werten:

- *name*="de.stardata.demo.client.bci.product.ProductBci"
- *exception* = "com.tlgen.common.exception.PersistenceException"

Aus diesen Informationen generiert TLGen eine Interface (Listing 40), notwendig für ein Session- Bean, und eine Factory Class, welche den Aufruf seitens des Client erleichtert (Listing 41):

```
package de.stardata.demo.client.bci.product;

import javax.ejb.Remote;
import de.stardata.demo.business.product.dataif.ProductPriceDataIf;
import de.stardata.demo.business.product.dataif.ProductAttributeDataIf;
import java.lang.String;
import de.stardata.demo.business.product.dataif.ClassificationDataIf;
import de.stardata.demo.business.product.dataif.ProductRelationDataIf;
import com.tlgen.common.exception.PersistenceException;
import de.stardata.demo.business.product.dataif.ProductDataIf;

@Remote
public interface ProductBci {

    // Client interface Fields
    public final static String JNDI_NAME = "de.stardata.demo.server.persistence.
product.ProductSessionBean";
    public final static String MANAGER_NAME = "managerProduct";
    public final static String EAR_NAME = "demo";

    .....
    public ProductDataIf createProduct(ProductDataIf arg) throws PersistenceException;
    public ProductDataIf findByPrimaryKeyProduct(ProductDataIf arg) throws PersistenceException;
    public void removeProduct(ProductDataIf arg) throws PersistenceException;

    ....
}
```

40 Listing

```
package de.stardata.demo.client.bci.product;

import de.stardata.demo.client.bci.product.ProductBci;
import com.tlgen.common.exception.PersistenceException;
import com.tlgen.common.bci.BciFactory;

public class ProductBciFactory extends BciFactory {

    private static final long serialVersionUID = 1033431309;

    /**
     * getProductBci()
```

```

    */
    public static synchronized ProductBci getProductBci() throws PersistenceException {
        return (ProductBci)getBciImplementation(ProductBci.class);
    }
}

```

41 Listing

Mit Hilfe **von** Interface und der Factory-Klasse ist der Aufruf seitens Client sehr einfach (Listing 42):

```

ProductData m_clProduct = new ProductData();
// Versorgung von Product Attribute mit Werten
.....
ProductBci factory = ProductBciFactory.getProductBci();
// Speichern von Product Object in die Datenbank
m_clProduct = factory.createProduct(m_clProduct);
// Lesen von den Product Object für ein Id
clProduct.setProductId(10);
ProductData product = factory.findByPrimaryKeyProduct(m_clProduct);

```

42 Listing

4.5.2.4.4.3 <Xml-persistence> Tag

Der Tag setzt die *Properties* der *persistence.xml*, z.B. notwendig in einer EAR, um die eigene Applikation in einem *Application Server* zu installieren. Diese Werte sind Applikation-Server abhängig. Statt der Attribute *value* wird hier *type* für <Property> verwendet.

```
<Property name="hibernate.dialect" type="org.hibernate.dialect.Oracle10gDialect" />
```

Beispiel: *JBoss Version 5.x* brauchte bei Verwendung der Oracle-Datenbank Version 11.x die folgende Information, um eine Datenbank-Verbindung aufzubauen:

```
<Property name="hibernate.dialect" type="org.hibernate.dialect.Oracle10gDialect" />
```

Diese Daten überschreiben komplett oder nur teilweise die Daten aus 4.4.2.7.3.

4.5.2.4.4.4 <Timerservice> Tag

Ein Timerservice kann mit einem Session Bean oder einer Message-Driven Bean verwendet werden. Damit kann eine Methode entweder an einem bestimmten Tag und zu einer bestimmten Zeit aufgerufen werden oder aber alle *n* Sekunden. Diese Methode über eine *Callback* Class kann einen Prozess steuern, starten, periodisch aufrufen oder stoppen.

Timerservice kann in drei unterschiedlichen Varianten verwendet werden:

- Voll automatisch, in diesem Fall sollten Start, Stopp-Zeit und die periodische Wiederholung initialisiert werden (Attribut *type*="auto").
- Start und Stopp-Zeit wird durch einen Client-Aufruf gesteuert (Attribut *type*="manuell", das ist die Default-Einstellung).
- Man verwendet die automatische und die manuelle Steuerungsmöglichkeit (Attribut *type*="merge") (siehe 2.8.3).

Es folgt eine Beschreibung der notwendigen Konfigurationsdaten.

Folgende Kinderelemente sind für diesen Tag notwendig:

- **<Variable>** (Daten-Struktur dieses Tags siehe 4.4.2.11) für folgende Felder:
 - **timerService**, mit der Annotation „*javax.annotation.Resource*“ und dem *vartype* = "*javax.ejb.TimerService*", wird für die Zeit-Steuerungs Methoden benötigt.
 - Eine Kommando Variable mit folgende Daten: *vartype* = „string“, um den *TimerService* zu beenden.
- **<Method>** (Daten-Struktur dieses Tags siehe 4.4.2.10) sind Methoden, die der *TimerService* benötigt. Folgende Methoden sind notwendig:
 - **name** = „init%Name%“, notwendig für die automatische Verwendung von *TimerService* und dessen Initialisierung mit folgenden notwendigen Parameter (siehe Listing 43):
 - **<Parameter>** *name*="start", definiert den Beginn dieser Service, *type*="date" ist die Start-Zeit
 - **<Parameter>** *name*="stop", definiert das Ende dieser Service, *type*="date", ist die Ende-Zeit
 - **<Parameter>** *name*="repeat", definiert die Wiederholungsrate; *type*="long", ist die Wiederholung von Zeit in Sekunden.
 - **name** = „timeout“, wird bei beiden Varianten der *TimerService*-Verwendung benötigt. Diese hat eine Annotation „*@javax.ejb.Timeout*“ und ruft die Callback Methode „*timeout*“ auf.
 - **name** = „start%Name%“, startet diese Service nach einem Client-Aufruf, ist für die manuelle *TimerService*-Verwendung notwendig
 - **name**="stop%Name%", stoppt manuell den Prozess.
- **<Callback>** ist eine Klasse, deren Methoden von *TimerService* Methoden aufgerufen werden, um Prozesse zu steuern. Werden mehrere Methoden verwendet, sollen die Namen dieser Methoden mit den Namen der *TimerService*-Methoden identisch sein. Diese Klasse wird vom Generator generiert. Die Inhalte dieser Methoden müssen per Hand programmiert werden. Dieser Tag hat folgende Attribute und Kinderelemente:
 - **Name** (Attribute) = „*eu.stardata.client.monitor.CallBackServiceMonitor*“, ist der *Callback* Klassenname
 - **Path** (Attribute) = "C:/Project-TLGen-2.5/source/test", ist der Ort, wo diese Klasse generiert wird. **<Method>** (Kindelement); Ist die aufzurufende Methode und kann mehrmals vorkommen.

```
<Method type="public void" name="initMonitor">
  <Parameter type="15.04.2011 10:30" name="start"/>
  <Parameter type="16.05.2011 9:30" name="stop"/>
  <Parameter type="600" name="repeat"/>
</Method>
```

43 Listing

4.5.2.4.4.5 <Webservice>

Ein Webservice ist ein beliebiger Service (z. B. ein Session-Bean), der von einem Client über z.B. das Internet aufgerufen werden kann.

Es werden zwei Arten von Webservice unterschieden, RPC (Remote Procedure Call) und DOCUMENT. RPC ist ein Aufruf einer Methode (z. B. eine Methode von einem Session-Bean) über Web.

TLGen generiert für diese Services den Client, den Webservice und eine Callback Klasse (siehe 2.8.3), falls diese nötig ist.

4.5.2.4.4.6 <Interceptor> Tag

Dieser Tag wird für die Generierung von Session-Bean *Interceptoren* verwendet und hat folgende Attribute:

- **name**=*eu.stardata.server.persistence.formular.TimeFormular*, ist der *Interceptor* Klassen-Name
- **path**=*C:/Project-TLGen-2.5/source/generated*, ist der Ort, wo die *Interceptor* Klasse generiert wird.

Ein *Interceptor* kann eine oder mehrere Methoden haben (siehe 4.4.2.10).

4.5.2.4.4.7 <Freeclass> Tag

Dieses Element wird für die Generierung einer Klasse, die als Schnittstelle zu dem Programmteil, welcher die Fachlogik beinhaltet, verwendet. Der Inhalt dieser Klasse muss selber implementiert werden. Falls der Generator diese Klasse in dem angegebenen Pfad findet, generiert er, abhängig vom Attribut „merge“, die Differenz aus beiden, ohne die vorhandenen Methoden zu löschen.

Folgende Attribute werden für diesen Tag verwendet:

- **name**=*eu.stardata.server.persistence.user.MappingEngine*, ist der Klassen-Name mit Pfad.
- **path**=*C:/Project-TLGen-2.5/source/user*, ist der Ort, wo diese Klasse generiert wird.
- **merge**, dieses Attribut hat folgende Werte:
 - **merge** = 0, TLGen generiert diese Klasse immer wieder neu.
 - **merge** = 1, diese Klasse wird, nur wenn sie nicht existiert, generiert.
 - **merge** = 2, wird eine neue Klasse generiert und mit dem vorhandenen per Hand implementierten Code zusammengefasst.

Diese Klasse kann eine oder mehrere Methoden haben (siehe 4.4.2.10).

4.5.2.4.5 <Messagedriven> Tag

Die Daten in diesem Element verwendet TLGen für die Generierung von Message-Driven-Bean. Dieser hat ein einziges Attribut; **name** = *Chat* und zwei Kinderelemente **<Client>** (siehe 4.5.2.4.5.1) und **<Bean>** (siehe 4.5.2.4.5.2).

Ein Beispiel für die Konfigurationsdatei ist in 5.2.2 und für den generierten Code in 5.2.10 zu finden.

4.5.2.4.5.1 <Client> Tag

Der Client-Tag hat folgende Attribute:

- **name**="eu.stardata.client.bci.chat.ChatBci|eu.stardata.client.message.chat.ChatMessageClient", hat zwei Teile, der erste Teil beinhaltet den vollständigen Namen für das Client-Interface und der zweite Teil für die Client-Klasse.

Der <Client> Tag benötigt zwei Kunderelemente:

1. **<Commentary>**, wird benutzt, um die Message Driven Bean zu beschreiben,
2. **<Variable>** (für die Daten Struktur siehe 4.4.2.11), beschreibt mehrere Felder:
 - a. **name**="JNDI_NAME_SENDER", ist das Senderfeld und braucht noch ein Attribut **content**="gueue/queueA", JNDI für den Sender (der Inhalt ist frei wählbar) und ein Attribut **comtype**="0". Dieses Feld ist Pflicht.
 - b. **name**="JNDI_NAME_RECEIVER", ist das Senderfeld und braucht noch ein Attribut **content**="topic/topicA", JNDI für den Receiver (der Inhalt ist frei wählbar) und ein Attribut **comtype**="1". Dieses Feld wird, nur wenn eine Antwort vom Server notwendig ist, verwendet.
 - c. **name**="ACKNOWLEDGE_MODE", dieses Field bestimmt wie der App.Server diese Nachricht quittiert oder überhaupt nicht. Im Attribut **content** sind zwei Eintragungen möglich, „Auto-acknowledge“ als Standard, bei diesem quittiert der APP.Server sofort die erhaltene Nachricht und „Dups-ok-acknowledge“, bei diesem lässt sich der Server für die Quittierung etwas Zeit.

Bemerkung: **comtype** Attribut kann folgenden Inhalt haben: „0“ oder „queue“ für „Point-To-Point“ Kommunikation und „1“ oder „topic“ für eine „Publish-and-Subscribe“ Kommunikation.

4.5.2.4.5.2 <Bean>-Tag

Dieser Tag hat ein Attribut und mehrere Kinder-Tags. Dessen Attribut ist ein vollständiger Name, z.B. **name**="eu.stardata.server.message.chat.ChatMessageBean". Der Message Driven Bean braucht eine Annotation, „@javax.ejb.Message“, um die Kommunikationseigenschaften zu definieren. Diese Informationen können über die Konfiguration-Datei dem Generator wie in Listing 44 übergeben werden.

```
<Annotation name="javax.ejb.ActivationConfigProperty">
  <Parameter type="MessageFormat = 'ChatMessage'" name="messageSelector"/>
</Annotation>
```

44. Listing

Folgende Parameter würden in der Konfigurationsdatei nach dem Muster von Listing 44 definiert werden:

1. **Parameter type** = "%connectclass%" **name** = "destinationType". Das Attribut **type** kann "javax.jms.Queue" oder "javax.jms.Topic" sein und der Platzhalter wird vom Generator mit dem **comtype** Attribut vom Client ersetzt.
2. **Parameter type** = "%connect%" **name** = "destination"; der Platzhalter wird vom Generator mit Variable "JNDI_NAME_SENDE" Inhalt ersetzt, in unserem Beispiel mit „gueue/queueA“.
3. **Parameter type** = " MessageFormat = 'ChatMessage' " **name** = "messageSelector"; über diesen Parameter für Kommunikation-Properties kann bestimmt werden, welche Nachrichten dem Bean zugestellt werden.
4. **Parameter type**="Auto-acknowledge" **name** = "acknowledgeMode" sind optional.

5. **Parameter type**="NonDurable" **name**="subscriptionDurability" sind Standard und für wichtige Nachrichten, die unbedingt zugestellt werden müssen, sollte der **type** = "Durable" gewählt werden.

Die Namen der zuvor beschriebenen Parameter sind nicht wählbar.

4.5.2.4.5.2.1 <Callback>-Tag

Dieser Tag hilft dem Generator eine *Callback* Klasse zu generieren, wo dann die Fachlogik programmiert werden kann: Diese Klasse wird jederzeit, wenn eine *Message* zugestellt wird, aufgerufen. Folgende Attribute sind für diesen Tag zu verwenden:

- **name** (Attribute) = „eu.stardata.client.chat.CallBackServerChatMessage“, ist der *Callback* Klassen-Name.
- **path** (Attribute) = "C:/Project-TLGen-2.5/source/test", ist der Ort, wo diese Klasse generiert wird.
- **merge** (Attribute) siehe 2.8.3

Die *Callback Class* kann eine oder mehrere Methoden haben (Listing 45):

```
<Method name="callBackChat" return="javax.jms.Message">
  <Exception name="com.tlgen.common.exception.PersistenceException"/>
  <Parameter name="message" type="javax.jms.Message"/>
</Method>
```

45 Listing

4.5.2.4.6 <Test> Tag

Mit Hilfe des <Test>-Tags generiert TLGen eine Test-Klasse, abgeleitet vom „junit.framework.TestCase“. Folgende Attribute können für diesen Tag verwendet werden:

- **name** = „eu.stardata.client.test.formular.FormularWriteReadTest“ ist der Name der Test Klasse und ein Pflicht-Feld,
- **filename** = "eu.stardata.client.test.formular.TestDataForFormular"; Ist dieses Attribut vorhanden, wird eine XML-Datei mit der Datenstruktur der Test- Klasse generiert. Existiert diese Datei schon, werden die dort gespeicherten Daten für die Initialisierung der Attribute der Testklasse für die „create“-Test-Methode verwendet (z.B. ein Datenbank-Insert).

4.5.2.4.6.1 Methoden für Test Klassen

Über die Test-Methoden werden die Aktionen „create“ für die Speicherung eines neuen Objekts in der Datenbank, „update“ für die Änderung eines Objekts, „find%Name%“ für das Lesen eines oder mehrerer Objekt bzw. Objekte und „delete“ für das Löschen eines Objekts in der Datenbank gesteuert. Die Daten-Struktur der Methoden siehe 4.4.2.10.

4.5.2.4.7 <Dbtable> Tag

Dieser Tag wird nur für die Generierung von Code auf Basis einer Datenbank verwendet und hilft zur Gruppierung bestimmter Tabellen eines Design/Session-Beans.

Das einzige Attribut ist **name**, welches gleich mit dem Tabellen-Namen sein muss. TLGen liest alle anderen Informationen über dieser Tabelle. Für jede Tabelle werden seitens TLGen eine Daten-Klasse und dessen *Interface* generiert (siehe 4.5.2.4.1).

In diesem Tag können Methoden, die nicht für die Datenpersistenz sondern nur für die Fachlogik verwendet werden, mit Hilfe von **<Method>**-Tag definiert werden (4.4.2.10).

Hier können auch die Relationen zwischen den Tabellen mit Hilfe des **<Relation>**-Tags definiert werden (4.5.2.4.7.1).

4.5.2.4.7.1 <Relation>-Tag

Ein Relation-Tag hat folgende Attribute:

1. **name** = „*OneToMany*“ definiert einen von der vier Relation-Typen: „*OneToOne*“, „*OneToMany*“, „*ManyToOne*“ und „*ManyToMany*“,
2. **table** = „*Table-Name*“ ist der Tabellen-Name mit dem diese Relation durchgeführt wird
3. **cascade** = „*Cascade-Type*“; Folgende mögliche Cascade-Typen: „*PERSIST*“, „*MERGE*“, „*REMOVE*“, „*REFRESH*“ oder „*ALL*“,
4. **optional** = „*true*“ oder „*false*“, (siehe 1)
5. **fetchType** = „*EAGER*“ oder „*LAZY*“ (siehe 1)
6. **direction** = „*true*“, „*false*“ ist eine Standard-Einstellung und gilt nur für die Relationen „*OneToOne*“ und „*ManyToMany*“, wo ersichtlich ist, was für eine Tabelle die „*Main*“-Table (Master/Slave-Prinzip) ist.

Der **<Relation>**-Tag hat zwei Kinder-Tags:

1. **<JoinColumn>** versorgt den Generator mit den Daten über die beteiligten Spalten (Columns) zu dieser Relation. Dieser hat folgende Attribute:
 - a. **name** = „*Column-Name*“ ist der eigene Spalten-Name für diese Relation.
 - b. **referencedColumnName** = „*Column-Name*“ ist der Spalten-Name anderer Tables, beteiligt bei dieser Relation.
 - c. **insertable** = „*true/false*“
 - d. **updatable** = „*true/false*“
 - e. **nullable** = „*true/false*“
2. **<JoinTable>** versorgt den Generator mit Informationen über die dritte Table (notwendig für eine „*ManyToMany*“ Relation). Dieser hat folgende Attribute:
 - a. **name** ist der Tabellen-Name
 - b. **catalog** ist der Datenbank-Katalog
 - c. **schema** ist das Datenbankschema des Users

Dieser Tag hat zwei Kinderelemente:

1. **<JoinColumn>**, siehe Punkt 1, wo ***name*** der Spaltenname ist.
2. **<InverseJoinColumn>** die Spalte der anderen Tabelle.

5 Installation von TLGen

TLGen wird als eine JAR Datei geliefert. Zur Verwendung benötigt man keine Besonderheiten.

In source/common sind ein paar Klassen (Super-, Exception- oder Client Remote-Service-Klassen), die TLGen als Standard-Klassen in der Generierung verwendet. Diese können erweitert oder geändert werden und sind optional.

Der TLGen Generierungsvorgang kann über Apache ANT verwendet werden, da TLGen einen eigenen Tag „*generator*“ besitzt (siehe Listing 46)

```
<target name="generator" depends="generator-init" description="--> generate helper file">
  <tlgen configFile = "${output}/config/config_tlgen_demo.xml"
    configStandardFile = "${output}/config/config_tlgen_demo_default.xml"
    databaseConnect = "jdbc:oracle:thin:@localhost:1521:ORCL"
    databaseDriver = "oracle.jdbc.driver.OracleDriver"
    userPwd = "Demo/demo"
    database = "Oracle"
    logFile = "${output}/log/generator_demo.log"
    logTest = "${output}/log/test_demo.log"
    logEntity = "${output}/log/entity_demo.log"
    pathToGenerate = "${output}/source/generated"
    debugValue = "debug"
    classpathref = "tlgen.class.path" >
    <fileset dir="${src.server}">
      <include name="**/${subsystem.dir}/**/*SessionBean.java"/>
    </fileset>
  </tlgen>
</target>
```

Der Tag „*generator*“ benötigt folgende Attribute:

- **configFile** ist die Konfiguration-Datei mit dessen Adresse
- **configStandardFile** ist die Standard Konfiguration-Datei mit dessen Adresse
- **databaseConnec**, ist der Connect String für eine Datenbank
- **databaseDriver** ist der Datenbank-Treiber
- **userPwd** ist der User und dessen Passwort für die Access zur Datenbank
- **database** ist die Datenbank (z. B „Oracle“)
- **logFile** Datei-Name für den Platz, wo alle Informationen über die Generierung geschrieben werden sollten
- **logTest** für die generierten Test-Klassen
- **logEntity** für die generierten Entity-Klassen,
- **logManager** für die generierten Manager-Klassen
- **logSessio0**, für die generierten Session Bean-Klassen
- **pathToGenerate** ist die Adresse, wo der generierte Code gespeichert werden sollte
- **debugValue** ist „*debug*“, hier werden die Debug Prints auf der Konsole geschrieben
- **classpathref** ist die Adresse, wo sich die benötigten JARS befinden (siehe Liste von 5.1)

Eine komplette ANT Datei ist im TLGen Example Paket ersichtlich.

a. Verwendete Tools und Externe Programme

Liste von verwendeten JAR's für die Generierung:

- Tlgen25.jar
- ant.jar
- junit.jar
- Andere jars, z.B. Datenbank-Verbindung oder Application Server Client/Server.

b. Beispiel Code

Es folgen Beispiele für den generierten Code.

i. Standard Konfiguration-Datei

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ===== -->
<!-- config standard demo for Project-TLGen-2.5 -->
<!-- ***** -->
<!-- Titus Livius Rosu, Titus Rosu, StarData GmbH -->
<!-- 2011/03/21 -->
<!-- ===== -->
<Standard name="Standard-BCI"
  company="StarData GmbH"
  path="de.stardata.demo.server.persistence"
  commentary="***** do not change with a file editor *****"
  >
  <Project testRelation="false,true,false,false"/>
  <Class path="de.stardata.demo.business">
  </Class>
  <Entity path="de.stardata.demo.server.persistence">
  </Entity>
  <Mapping name="de.stardata.demo.business.%package%.mapping.
  %classname%Mapping" classType="Mapping">
  </Mapping>
  <Manager name="de.stardata.demo.server.persistence.%package%.manager.
  %classname%Manager" classType="Manager"
  transactiontype="JTA"
  container="false"
  exception="com.tlgen.common.exception.PersistenceException">
  <Extends name="com.tlgen.common.ejb.manager.BaseManager"/>
  <Implements name="de.stardata.demo.server.persistence.%package%.managerif.
  %classname%ManagerIf"/>
  </Manager>
  <Session name="de.stardata.demo.server.persistence.%package%.
  %classname%SessionBean" classType="Session"
  managername="manager%classname%"
  transactiontype="JTA">
  <Extends name="com.tlgen.common.ejb.session.BaseSessionBean"/>
  <Client name="de.stardata.demo.client.bci.%package%.%classname%Bci" type="public"
  exception="com.tlgen.common.exception.PersistenceException">
  </Client>
  <Interceptor name="de.stardata.demo.server.persistence.%package%.TimeCore" path
  ="C:/Project-TLGen-2.2/source/generated">
```

```

        <Method name="timeTrace" return="java.lang.Object" finally="C:/Project-TLGen-
2.2/resources/FinallyTime.xml">
            </Method>
        </Interceptor>
        <Xml-persistence type="persistence">
            <Property name="hibernate.dialect" type="org.hibernate.dialect.Oracle10gDialect" />
            <Property name="show_sql" type="true" />
        </Xml-persistence>
    </Session>
    <Message name="de.stardata.demo.server.persistence.%package%.message" classType="Message" >
    </Message>
    <UmlControl name="DomainModel"
        lenString="255"
        versionDb="DB_VERSION"
        versionDbType="NUMBER(12,0)"
        primKeyType="long"
        endPrimKey="_ID"
        endSequence="_SEQ"
        sequenceCache="10"
        sequenceInit="1"
        sequenceIncrement="1"
        dataTablespace="ADMDATA"
        indexTablespace="ADMINDEX">
        <Column name="DB_VERSION" type="long" dbType="NUMBER" dataLen="16"
locking="true"/>
        <NameChange name="Order" type="OrderTable"/>
        <NameChange name="User" type="UserTable"/>
        <NameChange name="Access" type="AccessTable"/>
    </UmlControl>
</Standard>

```

ii. Konfiguration-Datei

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- ===== -->
<!-- config demo for Project-TLGen-2.5 -->
<!-- ***** -->
<!-- Titus Livius Rosu, Titus Rosu, StarData GmbH -->
<!-- 2011/03/21 -->
<!-- ===== -->
<Generator name="demo"
    inputsource="database"
    datasource="java:/demoPool"
    appserver="JBoss"
    standardConfigFile="C:/Project-TLGen-2.5/config/config_tlgen_demo_default.xml"
    umlFile="C:\Project-TLGen-2.5\doc\model\DomainModel-Demo-01.xml"
    databaseConnect="jdbc:oracle:thin:@localhost:1521:orcl"
    database="ORACLE"
    userPwd="ccp/ccp"
    type="guml"
    mapping="true"
    databaseScript="C:/Project-TLGen-2.5/doc/db/demo_schema.sql"
    makeDatabase="false"
    makeJavaCode="true"
    makeDatabaseDiff="false"
    indexTablespace="CCPINDEX"
    dataTablespace="CCPDATA"
    initProgram="de.stardata.base.config.InitProgramm"
    keyName="de.stardata.base.config.ConfigKeyNames"
    reference="true" >
    <Project name="Project StarData Demo for DomainModel"
        initialcontext = "jnp://localhost:9099"
        initialcontextfactory="org.jnp.interfaces.NamingContextFactory"

        initialcontextpkgprefix="org.jboss.naming:org.jnp.interfaces"

```

```

    type="guml"
    methods="public"
    sequencegen="SEQ_STORE"
    format="1"
    import="true"
    testRelation="true,true,true,true"
    persistence="true"
    multiReturnType="0"
    earname="demo">
<!-- ===== -->
<!--           Design von Product Session           -->
<!-- ===== -->
<Design name="product"
    methodreference="false">
    <Commentary name="This is a generate file"/>
    <Commentary name="do not change with a file editor"/>
    <Commentary name="file for Demo jar"/>
    <Manager>
        <Method name="findByName" parameter="java.lang.String" return=
            "%interfacedata%[]" manager="Product" methodtype="6">
            <Exception name="com.tlgen.common.
                exception.PersistenceException"/>
            <Parameter name="name" fullName="name" type="java.lang.String"
/>

                <Sql name="getName" sql="select o from ProductEntity o
                    where o.name = :name" />
            </Method>
            <Method name="findByNameAndOffer" parameter="java.lang.String"
                return="%interfacedata%[]" manager="Product" methodtype="6">
                <Exception name="com.tlgen.common.exception.
                    PersistenceException"/>
                <Parameter name="name" fullName="name" type="java.lang.String"
/>

                    <Parameter name="productOfferingId" fullName="productOfferingId"
                        type="java.lang.String" />
                <Sql name="getNameAndOffer" sql="select u from ProductEntity u
                    where u.name = :name and u.productOfferingId = :productOfferingId"
/>

                    </Method>
            <Method name="findByfindByMaterialNumber" parameter="java.lang.String"
                return="%interfacedata%[]" manager="Product" methodtype="6">
                <Exception name="com.tlgen.common.exception.
                    PersistenceException"/>
                <Parameter name="materialNumber" fullName="materialNumber"
                    type="java.lang.String" />
                <Sql name="getOffer" sql="select n from ProductEntity n where
                    n.materialNumber = :materialNumber" />
            </Method>
        </Manager>
        <Test name="de.stardata.demo.client.test.product.ProductWriteRead" type="public"
            num="3" fileName="de.stardata.demo.client.test.product.TestDataForProduct">
            <Extends name="junit.framework.TestCase"/>
            <Import name="com.tlgen.common.trace.Trace"/>
            <Method name="create" client="Product" manager="Product" typerwd="0">
            <Exception name="com.tlgen.common.exception.
                PersistenceException"/>
            <Parameter type="%interfacedata%" manager="Product"/>
            </Method>
            <Method name="findByPrimaryKey" client="Product"
return="%interfacedata%"
                manager="Product" typerwd="1">
            <Exception name="com.tlgen.common.exception.
                PersistenceException"/>
            <Parameter type="%interfacedata%" manager="Product"/>
            </Method>
            <Method name="findByName" client="Product" return="%interfacedata%[]"
                manager="Product" typerwd="1">
            <Exception name="com.tlgen.common.exception.

```

```

        PersistenceException"/>
        <Parameter name="name" type="java.lang.String" />
    </Method>
    <Method name="findByNameAndOffer" client="Product" return=
"%interfacedata%[]" manager="Product" typerwd="1">
        <Exception name="com.tlgen.common.exception.
PersistenceException"/>
        <Parameter name="name" type="java.lang.String" />
        <Parameter name="offer" type="java.lang.String" />
    </Method>
</Test>
<Test name="de.stardata.demo.client.test.product.ProductReadAll" type="public">
    <Extends name="junit.framework.TestCase"/>
    <Import name="com.tlgen.common.trace.Trace"/>
    <Method name="findAll" return="%interfacedata%[]" manager="Product"
client="Product" typerwd="1">
        <Exception name="com.tlgen.common.exception.
PersistenceException"/>
    </Method>
</Test>
</Design>
<!-- ===== -->
<!-- Design von user Session -->
<!-- ===== -->
<Design name="user" methodreference="false">
    <Commentary name="This is a generate file"/>
    <Commentary name="do not change with a file editor"/>
    <Commentary name="file for Demo jar"/>
    <Test name="de.stardata.demo.client.test.user.UserRoleWriteRead" default="false">
        <Extends name="junit.framework.TestCase"/>
        <Import name="com.tlgen.common.trace.Trace"/>
        <Method name="create" client="User" manager="UserRole" typerwd="0">
            <Exception name="com.tlgen.common.exception.
PersistenceException"/>
            <Parameter type="%interfacedata%" manager="UserRole"/>
        </Method>
        <Method name="findByPrimaryKey" client="User" return="%interfacedata%"
manager="UserRole" typerwd="1">
            <Exception name="com.tlgen.common.exception.
PersistenceException"/>
            <Parameter type="%interfacedata%" manager="UserRole"/>
        </Method>
    </Test>
    <Test name="de.stardata.demo.client.test.user.UserWriteRead" default="false">
        <Extends name="junit.framework.TestCase"/>
        <Import name="com.tlgen.common.trace.Trace"/>
        <Method name="create" client="User" manager="User" typerwd="0">
            <Exception name="com.tlgen.common.exception.
PersistenceException"/>
            <Parameter type="%interfacedata%" manager="User"/>
        </Method>
        <Method name="findByPrimaryKey" client="User" return="%interfacedata%"
manager="User" typerwd="1">
            <Exception name="com.tlgen.common.exception.
PersistenceException"/>
            <Parameter type="%interfacedata%" manager="User"/>
        </Method>
    </Test>
    <Test name="de.stardata.demo.client.test.user.WriteAccessControlContext"
default="false">
        <Extends name="junit.framework.TestCase"/>
        <Import name="com.tlgen.common.trace.Trace"/>
        <Method name="create" client="User" manager="AccessControlContext"
typerwd="0">
            <Exception name="com.tlgen.common.exception.
PersistenceException"/>
            <Parameter type="%interfacedata%"/>
        </Method>
    </Test>

```

```

</Test>
<Freeclass name = "de.stardata.demo.server.persistence.user.MappingEngine" path
  ="C:/Project-TLGen-2.1/source/user" merge="1">
  <Extends name="com.tlgen.common.ejb.session.BaseSessionBean"/>
  <Method name="startProcess" return="java.lang.String"
manager="UserRole">
    <Exception name="com.tlgen.common.exception.
PersistenceException"/>
    <Parameter name="user" type="java.lang.String"
manager="UserRole"/>
  </Method>
</Freeclass>
</Design>
<!-- ===== -->
<!--          Design von Chat-Rolen Session          -->
<!-- ===== -->
<Design name="Chat" methodreference="false">
  <Commentary name="This is a generate file"/>
  <Commentary name="Message test for chat"/>
  <Messagedriven name="Chat">
    <Commentary name="Messagedriven test comment"/>
    <Client name="eu.stardata.client.bci.chat.ChatBcij
eu.stardata.client.message.chat.ChatMessageClient">
      <Commentary name="Client test comment"/>
      <!--Extends name=com.tlgen.common.ejb.message.
BaseMessageClient/-->
      <!--Extends name=com.tlgen.common.bci.BciFactory num = "1"/-->
      <Variable name="JNDI_NAME_SENDER"
content="queue/queueA" comtype="0"/>
      <Variable name="JNDI_NAME_RECEIVER"
content="topic/topicA" comtype="topic"/>
    </Client>
    <Bean name="eu.stardata.server.message.chat.ChatMessageBean">
      <Annotation name="javax.ejb.ActivationConfigProperty">
        <Parameter type="MessageFormat =
'ChatMessage'" name="messageSelector"/>
      </Annotation>
      <!-- init the callback Class for receiver chat -->
      <Callback name =
"eu.stardata.client.chat.CallBackServerChatMessage"
path = "C:/Project-TLGen-2.5/source/test" merge="1">
        <Implements name="java.io.Serializable"/>
        <Method name="callBackChat"
return="javax.jms.Message">
          <Exception name="com.tlgen.common.exception.
PersistenceException"/>
          <Parameter name="message"
type="javax.jms.Message"/>
        </Method>
      </Callback>
    </Bean>
  </Messagedriven>
</Design>
</Project>
</Generator>

```

iii. Klasse und Interfaces

```
package de.stardata.demo.business.product.dataif;
```

```
import de.stardata.demo.business.production.dataif.CustomerServiceDataif;
import de.stardata.demo.business.product.dataif.ProductPriceDataif;
import de.stardata.demo.business.product.dataif.ProductAttributeDataif;
import java.lang.String;
```

```

import de.stardata.demo.business.product.dataif.ClassificationDataIf;
import de.stardata.demo.business.product.dataif.ProductRelationDataIf;
import com.tlgen.common.data.BaseDataIf;
import java.util.List;

public interface ProductDataIf extends BaseDataIf {
    // Methods from columns
    public abstract String getName();
    public abstract void setName(String arg);

    public abstract String getDescription();
    public abstract void setDescription(String arg);

    public abstract String getProductOfferingId();
    public abstract void setProductOfferingId(String arg);

    public abstract boolean getIsVisibleInCustomerOfferOrInvoice();
    public abstract void setIsVisibleInCustomerOfferOrInvoice(boolean arg);

    public abstract String getShortName();
    public abstract void setShortName(String arg);

    public abstract String getMaterialNumber();
    public abstract void setMaterialNumber(String arg);

    public abstract long getProductID();
    public abstract void setProductID(long arg);

    public abstract long getDbVersion();
    public abstract void setDbVersion(long arg);

    // Methods from relations tables
    public abstract List<ProductAttributeDataIf> getProductAttribute();
    public abstract void setProductAttribute(List<ProductAttributeDataIf> arg);

    public abstract List<ProductPriceDataIf> getProductPrice();
    public abstract void setProductPrice(List<ProductPriceDataIf> arg);

    public abstract List<ProductRelationDataIf> getProductRelation();
    public abstract void setProductRelation(List<ProductRelationDataIf> arg);

    public abstract List<ClassificationDataIf> getClassification();
    public abstract void setClassification(List<ClassificationDataIf> arg);

    public abstract CustomerServiceDataIf getCustomerService();
    public abstract void setCustomerService(CustomerServiceDataIf arg);
}

```

```

package de.stardata.demo.business.product.data;

```

```

import de.stardata.demo.business.production.dataif.CustomerServiceDataIf;
import de.stardata.demo.business.product.dataif.ProductPriceDataIf;
import java.lang.String;
import de.stardata.demo.business.product.dataif.ProductAttributeDataIf;
import com.tlgen.common.data.BaseData;
import de.stardata.demo.business.product.dataif.ClassificationDataIf;
import de.stardata.demo.business.product.dataif.ProductRelationDataIf;
import java.util.List;
import de.stardata.demo.business.product.dataif.ProductDataIf;

```

```

public class ProductData extends BaseData implements ProductDataIf {

    private static final long serialVersionUID = 137977358;
    // Methods from columns
    private String m_name;
    private String m_description;
    private String m_productOfferingId;

```

```
private boolean m_isVisibleInCustomerOfferOrInvoice;
private String m_shortName;
private String m_materialNumber;
private long m_productID;
private long m_dbVersion;
// Methods from relations tables
private List<ProductAttributeDataI> m_productAttribute;
private List<ProductPriceDataI> m_productPrice;
private List<ProductRelationDataI> m_productRelation;
private List<ClassificationDataI> m_classification;
private CustomerServiceDataI m_customerService;

// Methods from columns
public String getName() {
    return m_name;
}

public void setName(String arg) {
    m_name = arg;
}

public String getDescription() {
    return m_description;
}

public void setDescription(String arg) {
    m_description = arg;
}

public String getProductOfferingId() {
    return m_productOfferingId;
}

public void setProductOfferingId(String arg) {
    m_productOfferingId = arg;
}

public boolean getIsVisibleInCustomerOfferOrInvoice() {
    return m_isVisibleInCustomerOfferOrInvoice;
}

public void setIsVisibleInCustomerOfferOrInvoice(boolean arg) {
    m_isVisibleInCustomerOfferOrInvoice = arg;
}

public String getShortName() {
    return m_shortName;
}

public void setShortName(String arg) {
    m_shortName = arg;
}

public String getMaterialNumber() {
    return m_materialNumber;
}

public void setMaterialNumber(String arg) {
    m_materialNumber = arg;
}

public long getProductID() {
    return m_productID;
}

public void setProductID(long arg) {
    m_productID = arg;
}
```

```

    public long getDbVersion() {
        return m_dbVersion;
    }

    public void setDbVersion(long arg) {
        m_dbVersion = arg;
    }

    // Methods from relations tables
    public List<ProductAttributeDataIf> getProductAttribute() {
        return m_productAttribute;
    }

    public void setProductAttribute(List<ProductAttributeDataIf> arg) {
        m_productAttribute = arg;
    }

    public List<ProductPriceDataIf> getProductPrice() {
        return m_productPrice;
    }

    public void setProductPrice(List<ProductPriceDataIf> arg) {
        m_productPrice = arg;
    }

    public List<ProductRelationDataIf> getProductRelation() {
        return m_productRelation;
    }

    public void setProductRelation(List<ProductRelationDataIf> arg) {
        m_productRelation = arg;
    }

    public List<ClassificationDataIf> getClassification() {
        return m_classification;
    }

    public void setClassification(List<ClassificationDataIf> arg) {
        m_classification = arg;
    }

    public CustomerServiceDataIf getCustomerService() {
        return m_customerService;
    }

    public void setCustomerService(CustomerServiceDataIf arg) {
        m_customerService = arg;
    }
}

```

iv. Session Bean

```

package eu.stardata.server.persistence.user;

import eu.stardata.server.persistence.user.managerif.UserManagerIf;
import javax.interceptor.Interceptors;
import eu.stardata.server.persistence.user.managerif.RoleManagerIf;
import javax.ejb.Stateless;
import eu.stardata.client.bci.user.UserBci;
import eu.stardata.server.persistence.user.TimeUser;
import javax.ejb.Remote;
import com.tlgen.common.ejb.session.BaseSessionBean;
import java.lang.String;

```

```

import javax.ejb.EJB;
import eu.stardata.business.user.dataif.CoUserDataIf;
import com.tlgen.common.exception.PersistenceException;
import eu.stardata.business.user.dataif.CoRoleDataIf;

@Stateless(name = UserBci.JNDI_NAME, mappedName = "efp/"+
            UserBci.JNDI_NAME+"/remote")
@Remote(UserBci.class)
@Interceptors(TimeUser.class)
public class UserSessionBean extends BaseSessionBean implements UserBci {

    private static final long serialVersionUID = 457686048;

    // Session annotations and fields for local manager
    @EJB
    private UserManagerIf m_UserManagerIf;

    @EJB
    private RoleManagerIf m_RoleManagerIf;

    /**
     * Default Constructor
     */
    public UserSessionBean() {
        super();
    }

    // Session class methods
    /**
     * Session method "clearUser()"
     * @throws PersistenceException
     */
    public void clearUser() throws PersistenceException {
        try {
            m_UserManagerIf.clear();
        } catch (Exception ex) {
            throw new PersistenceException(ex.getMessage(), ex);
        }
    }

    /**
     * Session method "findAllUser()"
     * @throws PersistenceException
     */
    public CoUserDataIf[] findAllUser() throws PersistenceException {
        try {
            return m_UserManagerIf.findAll();
        } catch (Exception ex) {
            throw new PersistenceException(ex.getMessage(), ex);
        }
    }

    /**
     * Session method "findByNameUser()"
     * @param name
     * @return
     * @throws PersistenceException
     */
    public CoUserDataIf[] findByNameUser(String name)
        throws PersistenceException {
        try {

```

```

        return m_UserManagerIf.findByName(name);
    } catch(Exception ex) {
        throw new PersistenceException(ex.getMessage(),ex);
    }
}

/**
 * Session method "findUserByUsrAndPwdUser()"
 * @param username
 * @param pwd
 * @return
 * @throws PersistenceException
 */
public CoUserDataIf findUserByUsrAndPwdUser(String username, String
pwd)
    throws PersistenceException {
    try {
        return m_UserManagerIf.findUserByUsrAndPwd(username,
pwd);
    } catch(Exception ex) {
        throw new PersistenceException(ex.getMessage(),ex);
    }
}

/**
 * Session method "clearRole()"
 * @throws PersistenceException
 */
public void clearRole() throws PersistenceException {
    try {
        m_RoleManagerIf.clear();
    } catch(Exception ex) {
        throw new PersistenceException(ex.getMessage(),ex);
    }
}

/**
 * Session method "updateRole()"
 * @param arg
 * @return
 * @throws PersistenceException
 */
public void updateRole(CoRoleDataIf arg) throws PersistenceException
{
    try {
        m_RoleManagerIf.update(arg);
    } catch(Exception ex) {
        throw new PersistenceException(ex.getMessage(),ex);
    }
}

/**
 * Session method "flushUser()"
 * @throws PersistenceException
 */
public void flushUser() throws PersistenceException {
    try {
        m_UserManagerIf.flush();
    } catch(Exception ex) {
        throw new PersistenceException(ex.getMessage(),ex);
    }
}

/**

```

```

    * Session method "createUser()"
    * @param arg
    * @return
    * @throws PersistenceException
    */
    public CoUserDataIf createUser(CoUserDataIf arg)
    throws PersistenceException {
        try {
            return m_UserManagerIf.create(arg);
        } catch(Exception ex) {
            throw new PersistenceException(ex.getMessage(), ex);
        }
    }
}

```

v. Manager Bean

```

package eu.stardata.server.persistence.user.manager;

import javax.persistence.EntityManager;
import javax.ejb.Local;
import eu.stardata.server.persistence.user.managerif.UserManagerIf;
import com.tlgen.common.ejb.manager.BaseManager;
import eu.stardata.server.persistence.user.managerif.RoleManagerIf;
import com.tlgen.common.bci.ServiceLocator;
import javax.ejb.Stateless;
import eu.stardata.client.bci.user.UserBci;
import eu.stardata.server.persistence.user.entity.UserEntity;
import javax.persistence.PersistenceContext;
import java.lang.String;
import javax.ejb.EJB;
import eu.stardata.business.user.dataif.CoUserDataIf;
import eu.stardata.business.user.data.CoUserData;
import java.util.ArrayList;
import com.tlgen.common.exception.PersistenceException;
import eu.stardata.business.user.dataif.CoRoleDataIf;
import java.util.List;

@Stateless(name = UserManagerIf.JNDI_NAME)
@Local(UserManagerIf.class)
public class UserManager extends BaseManager implements UserManagerIf {

    private static final long serialVersionUID = 434486446;

    // Manager class data fields
    @PersistenceContext(unitName = UserBci.MANAGER_NAME)
    public EntityManager m_manager = null;
    private String m_getName =
        "SELECT u FROM User u WHERE u.username=:name";
    private String m_SQL_FIND_ALL = "select o from UserEntity o";
    private String m_singel = "SELECT u FROM User u
        WHERE u.username=:name AND u.pwd=:pass";

    @EJB
    private RoleManagerIf m_RoleManagerIf = null;

    // Manager class methods
    /**
     * Manager method "update()"
     * @param arg
     * @return

```

```

    * @throws PersistenceException
    */
    public void update(CoUserDataIf arg) throws PersistenceException {
        try {
            UserEntity entity = new UserEntity(arg);
            m_manager.merge(entity);
        } catch (Exception ex) {
            throw new PersistenceException(ex.getMessage(), ex);
        }
    }

    /**
     * Manager method "clear()"
     * @throws PersistenceException
     */
    public void clear() throws PersistenceException {
        try {
            m_manager.clear();
        } catch (Exception ex) {
            throw new PersistenceException(ex.getMessage(), ex);
        }
    }

    /**
     * Manager method "findByName()"
     * @param name
     * @return
     * @throws PersistenceException
     */
    public CoUserDataIf[] findByName(String name)
        throws PersistenceException {
        try {
            CoUserDataIf[] arrayData = null;
            List<?> list = m_manager.createQuery(m_getName).
                setParameter("name", name).getResultList();
            if (list != null && list.size() > 0) {
                arrayData = new CoUserDataIf[list.size()];
                int idx = 0;
                for (Object entity : list) {
                    if (entity != null) {
                        arrayData[idx++] = ((UserEntity)entity).
                            callUser();
                    }
                }
            }
            return arrayData;
        } catch (Exception ex) {
            throw new PersistenceException(ex.getMessage(), ex);
        }
    }

    /**
     * Manager method "flush()"
     * @throws PersistenceException
     */
    public void flush() throws PersistenceException {
        try {
            m_manager.flush();
        } catch (Exception ex) {
            throw new PersistenceException(ex.getMessage(), ex);
        }
    }
}

```

```

/**
 * Manager method "remove()"
 * @param arg
 * @return
 * @throws PersistenceException
 */
public void remove(CoUserDataIf arg) throws PersistenceException {
    try {
        UserEntity entity = m_manager.find(UserEntity.class,
            arg.getUserId());
        m_manager.remove(entity);
    } catch(Exception ex) {
        throw new PersistenceException(ex.getMessage(), ex);
    }
}

/**
 * Manager method "getEntityByPrimarykey()"
 * @param arg
 * @return
 * @throws PersistenceException
 */
public UserEntity getEntityByPrimarykey(CoUserDataIf arg)
    throws PersistenceException {
    try {
        UserEntity entity = m_manager.find(UserEntity.class,
            arg.getUserId());
        return entity;
    } catch(Exception ex) {
        throw new PersistenceException(ex.getMessage(), ex);
    }
}

/**
 * Manager method "findAll()"
 * @throws PersistenceException
 */
public CoUserDataIf[] findAll() throws PersistenceException {
    try {
        CoUserDataIf[] listData = null;
        List<?> list = m_manager.
            createQuery(m_SQL_FIND_ALL).getResultList();
        if(list != null && list.size() > 0) {
            listData = new CoUserDataIf[list.size()];
            int idx = 0;
            for(Object entity : list) {
                if(entity != null) {
                    listData[idx++] = ((UserEntity)entity).
                        callUser();
                }
            }
        }
        return listData;
    } catch(Exception ex) {
        throw new PersistenceException(ex.getMessage(), ex);
    }
}

/**
 * Manager method "findUserByUsrAndPwd()"
 * @param username

```

```

    * @param pwd
    * @return
    * @throws PersistenceException
    */
    public CoUserDataIf findUserByUsrAndPwd(String username, String pwd)
        throws PersistenceException {
        try {
            CoUserDataIf obj = (CoUserDataIf)m_manager.
                createQuery(m_singel).setParameter("username",username).
                setParameter("pwd",pwd).getResultList();
            return obj;
        } catch(Exception ex) {
            throw new PersistenceException(ex.getMessage(),ex);
        }
    }

    /**
     * Manager method "findByPrimaryKey()"
     * @param arg
     * @return
     * @throws PersistenceException
     */
    public CoUserDataIf findByPrimaryKey(CoUserDataIf arg)
        throws PersistenceException {
        try {
            CoUserDataIf classData = null;
            UserEntity entity = m_manager.find(UserEntity.class,
                arg.getUserId());
            if(entity != null) {
                classData = entity.callUser();
            }
            return classData;
        } catch(Exception ex) {
            throw new PersistenceException(ex.getMessage(),ex);
        }
    }

    /**
     * Manager method "getManager()"
     * @throws PersistenceException
     */
    public UserManagerIf getManager() throws PersistenceException {
        try {
            return (UserManagerIf)ServiceLocator.getInstance().
                getLocalReference(UserManagerIf.class);
        } catch(Exception ex) {
            throw new PersistenceException(ex.getMessage(),ex);
        }
    }

    /**
     * Manager method "create()"
     * @param arg
     * @return
     * @throws PersistenceException
     */
    public CoUserDataIf create(CoUserDataIf arg) throws
PersistenceException {
        try {
            UserEntity entity = new UserEntity(arg);
            m_manager.persist(entity);
            entity.addUser();

```

```

CoUserDataIf dataIf = entity.callUser();
// save the "CoRoleDataIf" object in the database
if(arg.getRole() != null && arg.getRole().size() > 0) {
    // get the new primary key from the "CoUserDataIf"
object
    List<CoUserDataIf> list = new ArrayList
        <CoUserDataIf>();
    CoUserDataIf data = new CoUserData();
    data.setUserId(dataIf.getUserId());
    list.add(data);
    for(CoRoleDataIf dataInvIf : arg.getRole()) {
        if(dataInvIf != null) {
            dataInvIf.setUser(list);
            dataInvIf = m_RoleManagerIf.
                create(dataInvIf);
        }
    }
    return dataIf;
} catch(Exception ex) {
    throw new PersistenceException(ex.getMessage(), ex);
}
}

/**
 * Manager method "close()"
 * @throws PersistenceException
 */
public void close() throws PersistenceException {
    try {
        m_manager.close();
    } catch(Exception ex) {
        throw new PersistenceException(ex.getMessage(), ex);
    }
}
}

```

```
package eu.stardata.server.persistence.user.managerif;
```

```

import eu.stardata.server.persistence.user.entity.UserEntity;
import eu.stardata.server.persistence.user.managerif.UserManagerIf;
import java.lang.String;
import eu.stardata.business.user.dataif.CoUserDataIf;
import com.tlgen.common.exception.PersistenceException;

```

```
public interface UserManagerIf {
```

```
    // Manager interface data fields
```

```
    public String JNDL_NAME = "eu.stardata.server.persistence.user.managerif.UserManagerIf";
    public String EAR_NAME = "efp";
```

```
    // Manager interface methods
```

```
    public void update(CoUserDataIf arg) throws PersistenceException;
```

```
    public void clear() throws PersistenceException;
```

```
    public CoUserDataIf[] findByName(String name) throws PersistenceException;
```

```
    public void flush() throws PersistenceException;
```

```
    public void remove(CoUserDataIf arg) throws PersistenceException;
```

```
    public UserEntity getEntityByPrimarykey(CoUserDataIf arg) throws PersistenceException;
```

```

    public CoUserDataIf[] findAll() throws PersistenceException;

    public CoUserDataIf findUserByUsrAndPwd(String username, String pwd) throws PersistenceException;

    public CoUserDataIf findByPrimaryKey(CoUserDataIf arg) throws PersistenceException;

    public UserManagerIf getManager() throws PersistenceException;

    public CoUserDataIf create(CoUserDataIf arg) throws PersistenceException;

    public void close() throws PersistenceException;
}

```

vi. Entity Bean

```

import javax.persistence.GeneratedValue;
import javax.persistence.Table;
import eu.stardata.server.persistence.user.entity.RoleEntity;
import javax.persistence.Column;
import javax.persistence.ManyToMany;
import com.tlgen.common.ejb.entity.BaseEntityBean;
import javax.persistence.SequenceGenerator;
import javax.persistence.GenerationType;
import java.lang.String;
import eu.stardata.business.user.dataif.CoUserDataIf;
import javax.persistence.Id;
import java.io.Serializable;
import java.util.ArrayList;
import javax.persistence.CascadeType;
import java.util.List;
import javax.persistence.Entity;

@Entity
@Table(name="CORE_USER")
@SequenceGenerator(name="SEQ_STORE_USER", sequenceName="CORE_USER_SEQ",
initialValue=1, allocationSize=20)
public class UserEntity extends BaseEntityBean implements Serializable {

    private static final long serialVersionUID = 1396589398;

    // Entity data field
    private CoUserDataIf m_user = null;

    // Fields from relations tables
    private List<RoleEntity> m_role = new ArrayList<RoleEntity>();

    // constructors
    /**
     * Default Constructor
     */
    public UserEntity() {
    }

    /**
     * Constructor
     * @param arg
     */
    public UserEntity(CoUserDataIf arg) {
        m_user = arg;
        fillUser();
    }
}

```

```

// Helper methods
/**
 * Helper Method "makeUser()"
 */
public CoUserDataIf makeUser() {
    if(m_user == null) {
        m_user = new eu.stardata.business.user.data.CoUserData();
    }
    return m_user;
}

/**
 * Helper Method "fillUser()"
 */
public void fillUser() {
}

/**
 * Helper Method "addUser()"
 */
public void addUser() {
}

/**
 * Helper Method "callUser()"
 */
public CoUserDataIf callUser() {
    // return the data object
    return makeUser();
}

// Methods from columns
@Id
@Column(name = "USER_ID")
@GeneratedValue(strategy = GenerationType.SEQUENCE,
    generator = "SEQ_STORE_USER")
public long getUserId() {
    return makeUser().getUserId();
}

public void setUserId(long arg) {
    makeUser().setUserId(arg);
}

@Column(name = "USERNAME", length = 32)
public String getUsername() {
    return makeUser().getUsername();
}

public void setUsername(String arg) {
    makeUser().setUsername(arg);
}

@Column(name = "PWD", length = 32)
public String getPwd() {
    return makeUser().getPwd();
}

public void setPwd(String arg) {

```

```

        makeUser().setPwd(arg);
    }

    @Column(name = "FORENAME", length = 32)
    public String getForename() {
        return makeUser().getForename();
    }

    public void setForename(String arg) {
        makeUser().setForename(arg);
    }

    @Column(name = "EMAIL", length = 128)
    public String getEmail() {
        return makeUser().getEmail();
    }

    public void setEmail(String arg) {
        makeUser().setEmail(arg);
    }

    @Column(name = "FUNCTION", length = 32)
    public String getFunction() {
        return makeUser().getFunction();
    }

    public void setFunction(String arg) {
        makeUser().setFunction(arg);
    }

    @Column(name = "PRIVILEGE", length = 18)
    public String getPrivilege() {
        return makeUser().getPrivilege();
    }

    public void setPrivilege(String arg) {
        makeUser().setPrivilege(arg);
    }

    // Methods from relations tables
    @ManyToOne(cascade =
{CascadeType.PERSIST, CascadeType.MERGE, CascadeType.REMOVE}, mappedBy =
"user", targetEntity = RoleEntity.class)
    public List<RoleEntity> getRole() {
        return m_role;
    }

    public void setRole(List<RoleEntity> arg) {
        m_role = arg;
    }
}

```

vii. Interceptor

```

package eu.stardata.server.persistence.user;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;
import com.tlgen.common.exception.PersistenceException;

```

```

public class TimeUser {

    private static final long serialVersionUID = 1075704904;

    // Session annotations and fields for interceptor

    /**
     * Default Constructor
     */
    public TimeUser() {
        super();
    }

    // Session class methods
    /**
     * Interceptor method "timeTrace()"
     * @param invocation
     * @return
     * @throws PersistenceException
     */
    @AroundInvoke
    public java.lang.Object timeTrace(InvocationContext invocation)
        throws PersistenceException {
        long start = 0;
        boolean toTime = true;
        try {

            if(toTime)
            {
                start = System.currentTimeMillis();
            }
            return invocation.proceed();

        } catch(Exception ex) {
            throw new PersistenceException(ex.getMessage(), ex, 1);
        }
        finally {
            if(toTime)
            {
                // Example to time
                long ende = System.currentTimeMillis();
                String klasse = invocation.getTarget().toString();
                String methode = invocation.getMethod().getName();
                System.out.println(klasse + ":" + methode + " -> "

+
                (ende - start) + "ms");
            }
        }
    }
}

```

viii. Timerservice

```

package eu.stardata.server.service.monitor;

import javax.annotation.Resource;
import javax.ejb.Stateless;
import javax.ejb.Timeout;
import javax.ejb.Timer;
import javax.ejb.TimerService;

```

```

import eu.stardata.client.bci.monitor.MonitorBci;

@Stateless(name = MonitorBci.JNDI_NAME, mappedName = "efp/"+
    MonitorBci.JNDI_NAME+"/remote")
public class MonitorSessionBean implements MonitorBci
{
    @Resource
    TimerService timerService;

    public static final String KOMMAND = "HSP";

    /**
     * start the monitor
     */
    public void startMonitor()
    {
        System.out.println(" ----- START MONITOR -----
");
        timerService.createTimer(1000, 1000, KOMMAND);
    }

    /**
     * stop the monitor
     */
    public void stopMonitor()
    {
        System.out.println(" ----- STOP MONITOR -----
");
        for(Object obj : timerService.getTimers())
        {
            Timer timer = (Timer) obj;
            String bez = (String) timer.getInfo();
            if (bez.equals(KOMMAND))
            {
                timer.cancel();
            }
        }
    }

    /**
     * time out
     */
    @Timeout
    public void timeout(Timer timer)
    {
        String bez = (String) timer.getInfo();
        if (bez.equals(KOMMAND))
        {
            System.out.println("Call Time :" +
Runtime.getRuntime().freeMemory());
        }
    }
}

```

ix. Webservice

x. Message Driven Bean

```

package eu.stardata.server.message.chat;

import javax.annotation.Resource;
import javax.jms.Message;
import javax.jms.JMSEException;
import javax.jms.ConnectionFactory;
import javax.jms.ObjectMessage;
import javax.ejb.ActivationConfigProperty;
import javax.jms.BytesMessage;
import javax.jms.Connection;
import com.tlgen.common.ejb.message.BaseMessageBean;
import eu.stardata.client.test.CallBackServerMessage;
import javax.jms.MessageProducer;
import javax.jms.TextMessage;
import javax.jms.Session;
import javax.jms.Topic;
import java.io.Serializable;
import javax.jms.MessageListener;
import javax.ejb.MessageDriven;

@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destination",
        propertyValue = "queue/queueA"),
    @ActivationConfigProperty(propertyName = "messageSelector",
        propertyValue = "MessageFormat = 'TestMessage'"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
        propertyValue = "Auto-acknowledge"),
    @ActivationConfigProperty(propertyName = "subscriptionDurability",
        propertyValue = "NonDurable")})
public class TestMessageBean extends BaseMessageBean
    implements MessageListener {

    private static final long serialVersionUID = 73342617;

    // Manager class data fields
    @Resource(mappedName = "ConnectionFactory")
    public ConnectionFactory m_factory = null;

    @Resource(mappedName = "topic/topicA")
    public Topic m_connect = null;

    // Message class methods
    /**
     * Message method "onMessage()"
     * @param message
     * @return
     */
    public void onMessage(Message message) {
        try {
            CallBackServerMessage callBack =
                new CallBackServerMessage();
            Object backObject = callBack.callBackChat(message);
            // send back to the client the result from the call back class
            if(m_factory != null && m_connect != null) {
                Connection connect = m_factory.createConnection();
                Session session = connect.createSession(false,
                    Session.AUTO_ACKNOWLEDGE);

```

```
MessageProducer sender = session.  
    createProducer(m_connect);  
// make message receive object  
if(backObject instanceof String) {  
    TextMessage textMessage = session.  
        createTextMessage();  
    textMessage.setText((String)backObject);  
    sender.send(textMessage);  
}  
else if(backObject instanceof byte[]) {  
    BytesMessage bytesMessage = session.  
        createBytesMessage();  
    bytesMessage.writeBytes((byte[])backObject);  
    sender.send(bytesMessage);  
}  
else {  
    ObjectMessage objMessage = session.  
        createObjectMessage();  
  
    objMessage.setObject((Serializable)backObject);  
    sender.send(objMessage);  
}  
connect.close();  
}  
catch(JMSEException ex) {  
    ex.printStackTrace();  
}  
}  
}
```

6. Literaturhinweis

1. "JSR 317: Java™ Persistence API, Version 2.0", EJB3 Persistence Specification
1. "JSR 318: Enterprise JavaBeans™, Version 3.1" EJB3 Specification