UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich Informatik

ist

Modelltransformationen mit Eclipse

Eine Fallstudie auf Basis der J2EE

Diplomarbeit
zur Erlangung des Grades eines Diplom-Informatikers

vorgelegt von
Ramy Hardan

Betreuer:	Prof. Dr. Jürgen Ebert, Institut für Softwaretechnik, Fachbereich Informatik
	Dr. Andreas Winter, Institut für Softwaretechnik, Fachbereich Informatik
Erstgutachter:	Prof. Dr. Jürgen Ebert, Institut für Softwaretechnik, Fachbereich Informatik
Zweitgutachter:	Dr. Andreas Winter, Institut für Softwaretechnik, Fachbereich Informatik

Koblenz, im September 2006

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Lahnstein, 28.09.2006

_____

Ramy Hardan

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
ATL	Atlas Transformation Language
BMP	Bean Managed Persistence
CMP	Container Managed Persistence
CORBA	Common Object Request Broker Architecture
DOM	Document Object Model
EIS	Enterprise Information System
EJB	Enterprise JavaBeans
EMF	Eclipse Modeling Framework
EMOF	Essential Meta Object Facility
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IBM	International Business Machines
IIOP	Internet Inter Orb Protocol
J2EE	Java 2 Platform, Enterprise Edition
J2SE	Java 2 Platform, Standard Edition
JAAS	Java Authentication and Authorization Service
JACC	Java Authorization Contract for Containers
JAF	Java Activation Framework
Java EE	Java Platform, Enterprise Edition
JAXP	Java API for XML Parsing
JAXR	Java API for XML Registries
JAX-RPC	Java API for XML-based Remote Procedure Call
JCA	J2EE Connector Architecture
JDBC	Java Database Connectivity
JDK	J2SE Development Kit
JDT	Java Development Tooling
JET	Java Emitter Templates
JMS	Java Message Service
JMX	Java Management Extensions
JNDI	Java Naming and Directory Interface
JPA	Java Persistence API

JRE	Java Runtime Environment
JRMP	Java Remote Method Protocol
JSF	JavaServer Faces
JSP	JavaServer Pages
JTA	Java Transaction API
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
MDA	Model Driven Architecture
MDSD	Model Driven Software Development
MOF	Meta Object Facility
OCL	Object Constraint Language
ORM	Object Relational Mapping
PAM	Pluggable Authentication Module
PIM	Platform Independent Model
PSM	Platform Specific Model
RCP	Rich Client Platform
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAAJ	SOAP with Attachments API for Java
SAX	Simple API for XML
SOAP	Simple Object Access Protocol
SPI	Service Provider Interface
SQL	Structured Query Language
TLD	Tag Library Descriptor
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XMI	Extensible Markup Language Metadata Interchange
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformation

1 Einleitung

In der vorliegenden Arbeit wird eine Software auf Basis der J2EE modellgetrieben entwickelt. Dieses einführende Kapitel klärt Motivation und Zielsetzung, führt in grundlegende Konzepte modellgetriebener Softwareentwicklung ein und gibt einen Überblick über das geplante Vorgehen.

1.1 Motivation

Modellgetriebene Softwareentwicklung (MDSD, Model Driven Software Development) strebt nach dem Ideal, Softwaresysteme vollständig und automatisch aus Modellen zu generieren. Die Modelle sollen dabei in einer dem Modellierungszweck und der Abstraktionsebene angemessenen Modellierungssprache formuliert und, ggf. in mehreren Schritten, in eine auf einer definierten Zielplattform lauffähige Software transformiert werden. Modelle dienen demnach nicht nur der Dokumentation, sondern unmittelbar der Konstruktion der Software. Die Zielplattform besteht aus nicht generierten, wiederverwendbaren Laufzeitkomponenten wie Betriebsystem, Middleware, technischer und fachlicher Frameworks.

Erfüllt sich die Vision modellgetriebener Softwareentwicklung, sind sämtliche Belange einer Software getrennt voneinander und redundanzfrei in adäquaten Modellen beschrieben. Die Integration der Modelle bis zur Übersetzung in lauffähigen Code erfolgt automatisch durch Modelltransformationen. Durch die Trennung der Belange und die Automatisierung werden optimale Wartbarkeit und maximale Produktivität erreicht.

1.2 Zentrale Konzepte modellgetriebener Softwareentwicklung

Die Kernelemente modellgetriebener Softwareentwicklung sind Modell, Metamodell, Modelltransformation und Zielplattform. Jeder MDSD-Ansatz muss das Zusammenspiel dieser Bausteine festlegen.

Folgender Modellbegriff liegt dieser Arbeit zugrunde ([Wint00], S. 104): „Ein Modell ist ein zielgerichtetes Abbild eines Systems, das zum einen ähnliche Beobachtungen und Aussagen ermöglicht wie dieses System und zum anderen diese Realität durch Abstraktion auf die jeweils problembezogen relevanten Aspekte vereinfacht.“ Aus dieser Definition folgt, dass verschiedene Aspekte eines (Software-) Systems unterschiedliche Modelle erfordern, die in unterschiedlichen, den Problemen und Abstraktionsgraden angepassten Modellierungssprachen formuliert sind.

Die (syntaktische) Definition von Modellierungssprachen erfolgt über Metamodelle[1]. In Anlehnung an [Wint00] definiert [Falk05]: „Ein Metamodell ist die Beschreibung einer Modellierung, die die verwendeten Modellierungskonzepte, d.h. die zur Modellierung verwendeten Sprachmittel, unabhängig von ihrer konkreten Notation, durch ihre abstrakte Syntax, verdeutlicht.“ Um Metamodelle zu beschreiben, ist ebenfalls eine Modellierungssprache nötig, die wiederum über ein Metamodell definiert wird usw. Um nicht endlos viele Metaebenen zu erhalten, bedarf es eines Meta-Metamodells (zur Beschreibung der Modellierungssprache für Metamodelle), das mit seinen eigenen Sprachmit-

[1] Ein Metamodell wird in dieser Arbeit im Sinne eines Metaschemas, also einer Beschreibung der abstrakten Syntax einer Modellierungssprache, verstanden. Andere „Metaisierungsprinzipien“ ([Stra98]) bleiben unberücksichtigt.

teln spezifiziert werden kann. So kann genau eine Sprache sämtliche Metamodelle definieren, die wiederum die unterschiedlichen Modellierungssprachen beschreiben.

Erst das gemeinsame Meta-Metamodell ermöglicht es den Modelltransformationen, in unterschiedlichen Modellierungssprachen formulierte Modelle zusammenzuführen, schrittweise bis auf die Ebene der Zielplattform zu übersetzen und so eine lauffähige Anwendung zu erzeugen. Modelltransformationen sind der Kern der MDSD. Erst sie ermöglichen es, Modelle unmittelbar zur Konstruktion von Software heranzuziehen. Während Metamodelle die (abstrakte) Syntax der Modelle festlegen, bestimmen die Modelltransformationen durch die Abbildung auf die Zielplattform deren Semantik.

Die Zielplattform bildet der Endpunkt aller Modelltransformationsschritte und ist das Subsystem, das die zur Laufzeit der Software benötigten Funktionen und Dienste über definierte Schnittstellen bereitstellt. Die Abstraktionsebene dieser Funktionen und Dienste bestimmt die „Tiefe“ der Modelltransformationen. Für die meisten Zielplattformen liegt das Transformationsergebnis als Programmcode vor, der ebenfalls als Modell verstanden werden kann.

1.2.1 Standardisierungsbemühungen

Eine konkrete Architektur zur Verwirklichung modellgetriebener Softwareentwicklung hat die Object Management Group[2] (OMG) mit der Model Driven Architecture (MDA) vorgeschlagen ([OMG03a]). Plattformunabhängige Modelle (Platform Independent Model, PIM) werden in plattformabhängige transformiert (Platform Specific Model, PSM). Ein PIM spezifiziert einen Teil des Systems unabhängig von technischen Belangen der Plattform. Das durch Modelltransformationen mit Plattformspezifika angereicherte PSM enthält genug Information, um z. B. durch Codegenerierung in die Zielplattform eingebunden zu werden. Die MDA betrachtet die Codegenerierung als einen Schritt, der von den Modelltransformationen getrennten ist.

Als Modellierungssprache war zunächst die Unified Modeling Language (UML, [OMG03b]) vorgesehen, die, insbesondere für die Modellierung der PSM, durch Profile erweitert werden kann. Jüngere Entwicklungen ([OMG05a]) folgen dem flexibleren, oben beschriebenen Ansatz und erlauben neben UML die Definition eigener Modellierungssprachen. Als einheitliches Meta-Metamodell wird Meta Object Facility (MOF, [OMG03c]) vorgeschrieben.

Zwar konnten UML oder MOF auf dem Gebiet der Modellierungssprachen einen festen Platz gewinnen, im Bereich der Modelltransformation tummelt sich jedoch eine Vielzahl von Ansätzen auch außerhalb der andauernden Standardisierungsbemühungen der OMG in Form von Query / Views / Transformations (QVT, [OMG02b]). Eine Übersicht und Klassifikation solcher Ansätze liefern [CzHe03] und [Falk05].

1.3 Zielsetzung

Ziel der Arbeit ist es, anhand eines Fallbeispiels zu untersuchen, wie modellgetriebene Softwareentwicklung mit verfügbaren Mitteln unter dem Dach einer integrierten Entwicklungsumgebung umgesetzt werden kann. Dazu müssen sowohl die zu erstellende Anwendung als auch die Zielplattform ausreichende Komplexität besitzen, damit die verwendeten Werkzeuge und Techniken ihre Einsatzwürdigkeit für reale Projekte beweisen können. Im Zentrum steht dabei die Entwicklung von Modelltransformationen,

[2] http://www.omg.org/

die in unterschiedlichen Modellierungssprachen formulierte Modelle integrieren und auf die Zielplattform abbilden.

1.4 Vorgehensweise

In Kapitel 2 werden die Anforderungen an die zu entwickelnde Anwendung in Form eines Pflichtenheftes definiert. Sie ist als Webanwendung ausgelegt und nutzt eine relationale Datenbank als zentralen Datenspeicher, weshalb sich die in Kapitel 3 beschriebene J2EE als Zielplattform anbietet. Diese erfüllt die geforderten Komplexitätsanforderungen problemlos.

Kapitel 4 führt kurz in die Grundlagen der genutzten Entwicklungsumgebung Eclipse ein und betrachtet das dort verfügbare Eclipse Modeling Framework, das im weiteren Verlauf für die Definition der verschiedenen Metamodelle genutzt wird.

Nachdem Kapitel 5 eine Übersicht über die entworfene Softwarearchitektur der Anwendung gibt, stehen in Kapitel 6 die Transformationssprache Atlas Transformation Language und damit formulierte Modelltransformationen im Vordergrund. Dabei werden Modelle unterschiedlicher Metamodelle schrittweise in Artefakte der Zielplattform überführt. In Kapitel 6 betrifft dies den Bereich der Datenspeicherung in relationalen Datenbanken, in Kapitel 7 erfolgt die Beschreibung der Modelle, Metamodelle und Transformationen, die zur Generierung der für die Verarbeitung von HTTP-Anfragen verantwortlichen Artefakte beitragen.

Kapitel 8 wendet sich zusammenfassend der auf der Zielplattform lauffähigen Anwendung zu, betrachtet die eingesetzten Modelle und zeigt das Zusammenspiel zwischen generierten und nicht generierten Artefakten.

Die Betrachtungen in Kapitel 9 verdichten schließlich die Ergebnisse dieser Arbeit, üben Kritik an den eingesetzten Werkzeugen und schlagen mögliche Weiterentwicklungen vor.

2 Anforderungsdefinition für das Fallbeispiel

Dieses Kapitel enthält die Anforderungen für die im Rahmen der Fallstudie zu erstellende Software, einem webbasierten Spiel, das es einer unbegrenzten Zahl von Nutzern ermöglicht, virtuelle Fußballmannschaften zu managen.

Damit alle Anforderungen weitgehend einheitlich präsentiert werden können, wird folgendes Schema verwendet:

Name:	Der im gesamten Anforderungsdokument eindeutige Name der Anforderung
Art:	Die Beschreibung der Anforderungsart, sofern möglich, bei funktionalen Anforderungen wie **Datenmodell** (z. B. nutzer-a-10) oder **Dialog** (z. B. nutzer-a-20)
Text:	Die Beschreibung der Anforderung selbst. Grundsätzlich werden Formulierungen im Aktiv angestrebt, wodurch deutlich werden soll, wer die Anforderung zu erfüllen hat.
Begründung:	Die Motivation für die Anforderung bzw. das Ziel, das mit der Erfüllung der Anforderung erreicht werden soll. Da die Gründe für die funktionalen Anforderungen hauptsächlich in den willkürlichen Spielregeln liegen, entfällt diese Angabe meist.
Beispiel:	Ein konkretes Szenario zur Vermeidung von möglichen Verständnisschwierigkeiten des Anforderungstextes
Abnahme:	Die Abnahmekriterien für die Validierung der Anforderung. Da wegen des fehlenden Auftraggebers in dieser Fallstudie die Abnahmekriterien willkürlich sind, wurde auf deren Angabe weitgehend verzichtet.
Abhängigkeit:	Die Beziehungen zu anderen Anforderungen. Deren Angabe dient einerseits dem besseren Verständnis, andererseits dem Erkennen von möglichen Inkonsistenzen bei Änderungen.

Nach einer kurzen Einführung in das Spielkonzept werden die funktionalen und nichtfunktionalen Anforderungen getrennt betrachtet.

Die Einteilung der funktionalen Anforderungen erfolgt nach den Spielbereichen **Nutzer und Miniligen**, **Spieler**, **Transfermarkt** sowie **Spielerbewertung und Punktesystem**. Für jeden Bereich wird ggf. ein einführender Text mit Anwendungsfalldiagramm zusätzlich angegeben. Die Anforderungen werden weiter gegliedert: **Prämissen** beschreiben, was das System als gegeben voraussetzen kann. Sie hängen oft mit **Leistungsausgrenzungen** zusammen, die festlegen, welche Anforderungen das System nicht zu erfüllen hat. **Anforderungen** dienen schließlich der Leistungsbeschreibung.

Begriffe aus dem Glossar (Kapitel 2.4) werden bei erstmaligem Auftreten im Text *kursiv* dargestellt.

2.1 Spielkonzept

Im Spiel konkurrieren maximal 12 *Nutzer* innerhalb einer *Miniliga* um die Plätze der *Ligatabelle*. Ziel des Spiels ist es, in einer Saison die meisten Punkte innerhalb der Miniliga zu erhalten. Der Nutzer startet mit einer festen Summe Spielgeld und stellt sich aus aktuellen *Spielern* der ersten Fußball Bundesliga, die er am *Transfermarkt* einkauft, eine virtuelle *Mannschaft* zusammen. Nach jedem realen *Spieltag* erhält jeder Bundesligaspieler, sofern er eingesetzt wurde, eine Bewertung und eine entsprechende Anzahl an Punkten. Der Nutzer erhält die Summe der Punkte über die Spieler seiner Mannschaft. Endet die Saison, hat der Nutzer mit den meisten Punkten gewonnen.

2.2 Funktionale Anforderungen

2.2.1 Nutzer und Miniligen

Die Anwendungsfälle für den Bereich Nutzer und Miniligen zeigt Abb. 2.1.

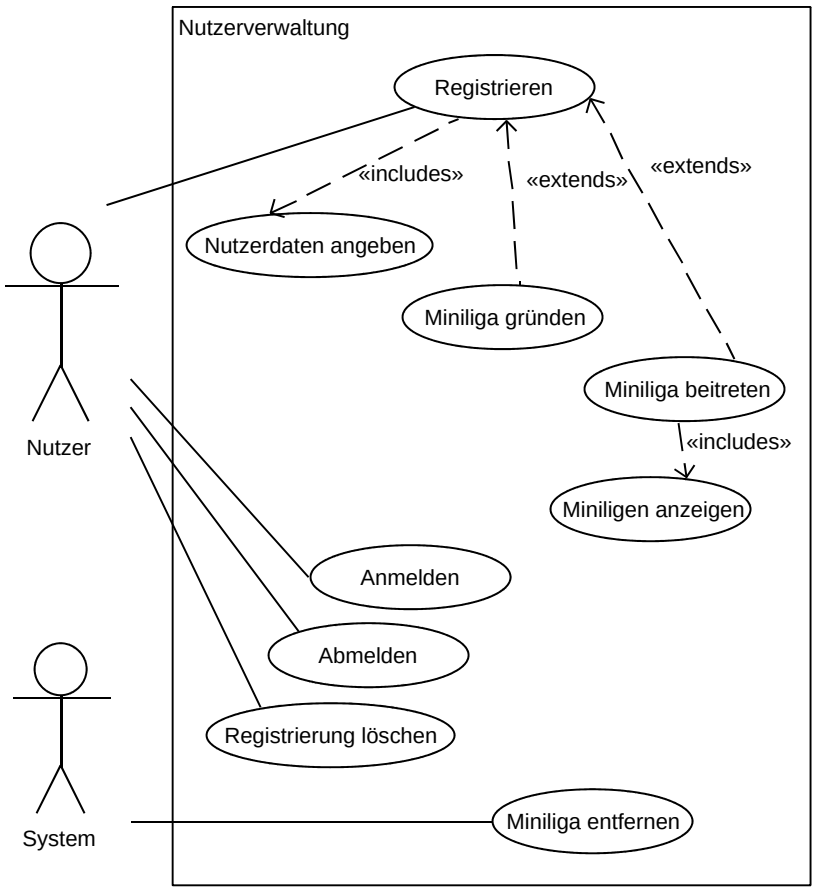


Abb. 2.1 Anwendungsfalldiagramm der Nutzerverwaltung

Anforderungen

Name:	nutzer-a-10
Art:	Datenmodell
Text:	Für jeden Nutzer muss das System den Nutzernamen, das Nutzerpasswort und den aktuellen Spielgeldbetrag speichern. Der Nutzername muss systemweit eindeutig sein.

Name:	nutzer-a-20
Art:	Dialog
Text:	Der Nutzer muss bei Registrierung Nutzernamen und Nutzerpasswort angeben.

Name:	nutzer-a-40
Text:	Der Nutzer muss bei Registrierung entweder eine Miniliga gründen oder einer von einem anderen Nutzer bereits gegründeten Miniliga beitreten.

Name:	nutzer-a-45
Art:	Datenmodell
Text:	Das System muss sicherstellen, dass keine Miniliga mehr als 12 Nutzer enthält.

Name:	nutzer-a-50
Art:	Datenmodell
Text:	Ein Nutzer gehört bis zur Löschung seiner Registrierung zu genau ein und derselben Miniliga. Diese bestimmt der Nutzer durch Gründung oder Beitritt.

Name:	nutzer-a-60
Art:	Datenmodell
Text:	Das System muss für jede Miniliga einen systemweit eindeutigen Miniliganamen speichern.

Name:	nutzer-a-70
Art:	Dialog
Text:	Der Nutzer muss bei Gründung einer Miniliga den Miniliganamen angeben.
Abhängigkeit:	nutzer-a-60

Name:	nutzer-a-80
Text:	Das System muss eine Miniliga entfernen, sobald der letzte in der Miniliga verbliebene Nutzer seine Registrierung löscht.

Name:	nutzer-a-90
Text:	Das System muss alle Spieler aus der Mannschaft des Nutzers für System*offerten* am Transfermarkt der Miniliga verfügbar machen, sobald der Nutzer seine Registrierung löscht (siehe Abschnitt Transfermarkt).

2.2.2 Spieler

Prämissen

Name:	spieler-p-10
Art:	Datenmodell
Text:	Alle zu Saisonbeginn gemeldeten Spieler der ersten Fußball Bundesliga sind zum Spielbeginn im System mit vollständigem Namen und Geburtsdatum gespeichert.

Anforderungen

Name:	spieler-a-10
Art:	Datenmodell
Text:	Für jeden Spieler muss das System die Durchschnittsbewertung, die bisher erreichten Punkte und den aktuellen Marktwert speichern.

Name:	spieler-a-20
Text:	Zu jedem Saisonbeginn muss das System Spielerpunkte und Durchschnittsbewertung auf die Anfangswerte zurücksetzen, d. h. 0 Punkte und keine Bewertung.
Abhängigkeit:	spieler-a-10

Name:	spieler-a-30
Text:	Der Marktwert wird nie zurückgesetzt.
Begründung:	Der unveränderte Marktwert soll die Spielmotivation für den Nutzer über Saisons hinweg erhalten.

Name:	spieler-a-40
Art:	Datenmodell
Text:	Zu Spielbeginn muss das System den Marktwert jedes Spielers auf 800.000 Euro Spielgeld setzen.

Leistungsausgrenzungen

Name:	spieler-l-10
Text:	Das Eingeben der Spielerdaten muss vom System nicht unterstützt werden.
Begründung:	Dies liegt außerhalb des Rahmens der Fallstudie.

Name:	spieler-l-20
Art:	Datenmodell
Text:	Die Position des Spielers (Tor, Abwehr, Mittelfeld, Sturm) und Statistiken aus der Saison (Tore, gelbe Karten, etc.) müssen nicht erfasst sein.

2.2.3 Transfermarkt

Am Transfermarkt können die Nutzer innerhalb einer Miniliga Spieler kaufen und verkaufen, um ihre Mannschaft zu verwalten. Abbildung 2.2 zeigt die Funktionen des Transfermarktes.

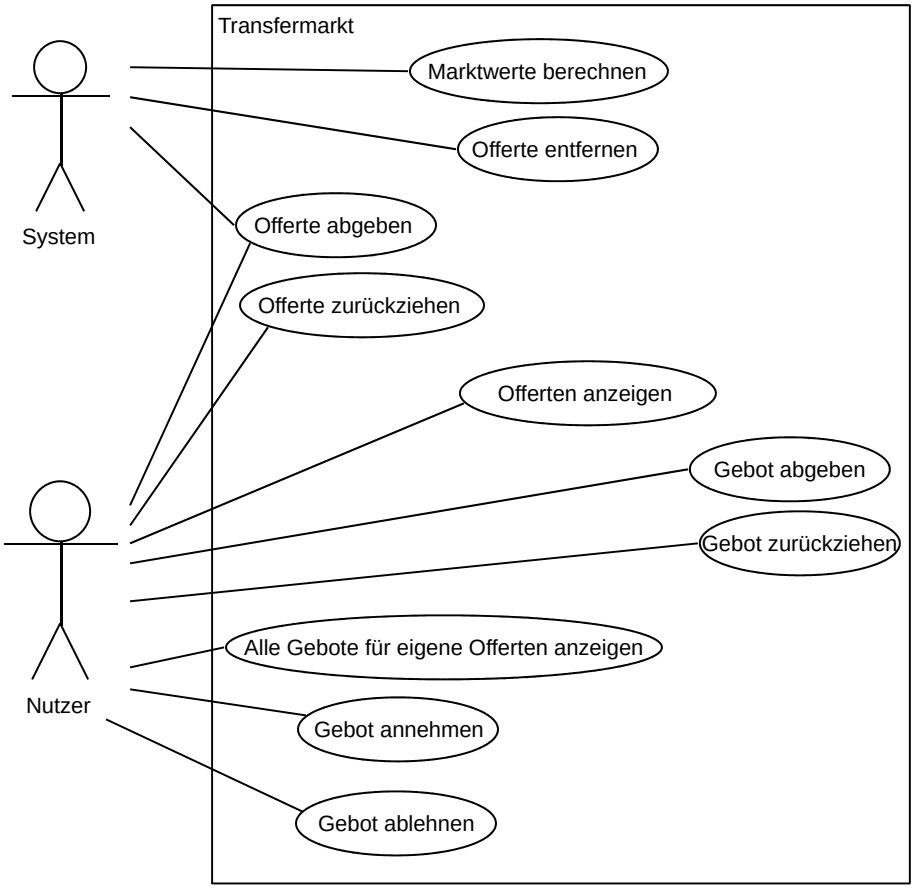


Abb. 2.2 Anwendungsfalldiagramm des Transfermarktes

Anforderungen

Name:	trans-a-10
Art:	Datenmodell
Text:	Zu Spielbeginn startet der Nutzer ohne Spieler in seiner Mannschaft und besitzt einen Spielgeldbetrag in Höhe von 40 Millionen Euro.

Name:	trans-a-20
Text:	Ein Transfermarkt ist begrenzt auf jeweils eine Miniliga, d. h. Offerten und *Gebote* gelten nur innerhalb dieser.

Name:	trans-a-30
Art:	Datenmodell
Text:	Jede Offerte muss einen *Richtwert* (Mindestgebotsbetrag) > 0 besitzen.

Name:	trans-a-40
Text:	Das System muss sicherstellen, dass zu jedem Zeitpunkt höchstens eine Offerte pro Spieler auf einem Transfermarkt existiert.

Name:	trans-a-60
Text:	Das System gibt ausschließlich Offerten für Spieler ab, die in keiner Mannschaft der jeweiligen Miniliga vorhanden sind.

Name:	trans-a-70
Text:	Der Nutzer darf nur Offerten für einen Spieler abgeben, der sich in seiner Mannschaft befindet.

Name:	trans-a-80
Art:	Datenmodell
Text:	Ein Nutzer darf maximal 22 Spieler in seiner Mannschaft besitzen.

Name:	trans-a-90
Text:	Ein Nutzer darf höchstens zu so vielen Offerten Gebote abgeben, wie ihm zum Erreichen von 22 Spielern in seiner Mannschaft fehlen.
Beispiel:	Befinden sich 13 Spieler in der Mannschaft, darf der Nutzer zu höchstens 9 Offerten Gebote abgeben.

Name:	trans-a-95
Art:	Dialog
Text:	Die Höhe des Gebotsbetrages ist beschränkt durch den Richtwert der betreffenden Offerte (untere Schranke) und dem aktuellen Spielgeldbetrag des bietenden Nutzers (obere Schranke). Das System muss dies bei Gebotsabgabe sicherstellen.
Begründung:	Der Nutzer darf sich nicht verschulden.

Name:	trans-a-97
Text:	Der Nutzer muss jedes seiner Gebote jederzeit zurückziehen können, sofern der offerierende Nutzer es nicht akzeptiert hat.

Name:	trans-a-100
Text:	Das System muss täglich den Marktwert jedes Spielers berechnen. Der neue Marktwert eines Spielers ergibt sich aus dem arithmetischen Mittel über dem alten Marktwert und allen *Transferbeträgen* des Tages in allen Miniligen.
Beispiel:	Spieler S1 mit Marktwert $M1_{alt}$ = 850.000 Euro wurde an einem Tag in 3 Miniligen L1, L2, L3 mit Transferbeträgen T_{L1} = 800.000 Euro, T_{L2} = 900.000 Euro, T_{L3} = 1.000.000 Euro gehandelt. Somit ist $M1_{neu} = \frac{(M1_{alt} + T_{L1} + T_{L2} + T_{L3})}{4} = 887.500\,Euro$.

Name:	trans-a-110
Text:	Der Richtwert bei Systemofferten muss zum Zeitpunkt der Abgabe dem aktuellen Marktwert entsprechen und für die Dauer der Offerte konstant bleiben.

Name:	trans-a-120
Text:	Das System muss eine Systemofferte genau dann entfernen, sobald sie sich länger als 48 Stunden auf dem Transfermarkt befindet und kein Gebot von einem Nutzer für sie abgegeben wurde.
Begründung:	Für die Nutzer der Miniliga uninteressante Offerten sollen entfernt werden, damit die Übersicht nicht leidet.

Name:	trans-a-130
Text:	Das System muss eine Nutzerofferte genau dann entfernen, sobald sie sich länger als 72 Stunden auf dem Transfermarkt befindet.
Begründung:	Vom offerierenden Nutzer vergessene oder für andere Nutzer der Miniliga uninteressante Offerten sollen entfernt werden, damit die Übersicht nicht leidet.

Name:	trans-a-140
Text:	Der Nutzer darf nur eigene Offerten zurückziehen, muss dies aber jederzeit tun können, insbesondere auch unabhängig von bereits abgegebenen Geboten.

Name:	trans-a-150
Text:	Entfernt das System eine Systemofferte ohne Gebot (trans-a-120), muss das System bei der Marktwertberechnung (trans-a-100) für die entsprechende Miniliga ein Pseudotransferbetrag in Höhe von 85% des Offertenrichtwertes annehmen.
Beispiel:	Systemofferte O1 mit Richtwert = 1.000.000 Euro für Spieler S1 in Miniliga L1 bleibt ohne Gebot und wird entfernt (trans-a-120). Für die Marktwertberechnung (trans-a-100) ergibt sich für diesen Tag in Miniliga L1 ein Transferbetrag T_{L1} = 850.000 Euro.
Abhängigkeit:	trans-a-100, trans-a-120

Name:	trans-a-170
Text:	Das System muss für eine Systemofferte genau dann einen *Transfer* durchführen, sobald sie sich länger als 48 Stunden auf dem Transfermarkt befindet und ein Nutzer mindestens ein Gebot abgegeben hat. Existieren mehrere Gebote, ist für den Transfer das höchste maßgeblich.

Name:	trans-a-180
Text:	Das System muss für eine Nutzerofferte genau dann einen Transfer durchführen, sobald der offerierende Nutzer ein dafür abgegebenes Gebot akzeptiert.

Name:	trans-a-190
Text:	Das System muss mit Durchführung eines Transfers die zugehörige Offerte und alle zugehörigen Gebote entfernen.
Abhängigkeit:	trans-a-170, trans-a-180

Name:	trans-a-200
Text:	Das System muss mit Durchführung eines Transfers den Transferbetrag vom Spielgeldbetrag des bietenden Nutzers subtrahieren und im Falle von Nutzerofferten auf den Spielgeldbetrag des offerierenden Nutzers addieren.
Abhängigkeit:	trans-a-170, trans-a-180

Name:	trans-a-210
Text:	Das System muss mit Durchführung eines Transfers den betreffenden Spieler der Mannschaft des bietenden Nutzers hinzufügen und ihn im Falle einer Nutzerofferte aus der Mannschaft des offerierenden Nutzers entfernen.
Abhängigkeit:	trans-a-170, trans-a-180

Name:	trans-a-220
Text:	Das System muss sicherstellen, dass stets genau 20 Systemofferten pro Transfermarkt existieren. Die betreffenden Spieler muss das System zufällig (gleich verteilt) auswählen.
Begründung:	Der Nutzer soll jederzeit die Möglichkeit haben, Spieler zu kaufen, auch wenn andere Nutzer keine Offerten abgeben.
Abhängigkeit:	trans-a-60

Name:	trans-a-230
Art:	Dialog
Text:	Lässt sich der Nutzer alle Offerten anzeigen, muss das System für jede Offerte Spielername, aktueller Marktwert, Richtwert, Ende der Offerte und Gebot des Nutzers, sofern abgegeben und nicht zurückgezogen, anzeigen.
Begründung:	Der Nutzer soll auf einen Blick eine Gesamtübersicht erhalten.

Name:	trans-a-240
Text:	Der Spielgeldbetrag des Nutzers wird nie zurückgesetzt.

2.2.4 Spielerbewertung und Punktesystem

Nach jedem Bundesligaspieltag erhalten alle Spieler, die mindestens 20 Minuten real im Einsatz waren, eine Bewertung in Form von Schulnoten von einer Fachzeitschrift[3]. Diese Bewertung ist Grundlage für den Spielverlauf.

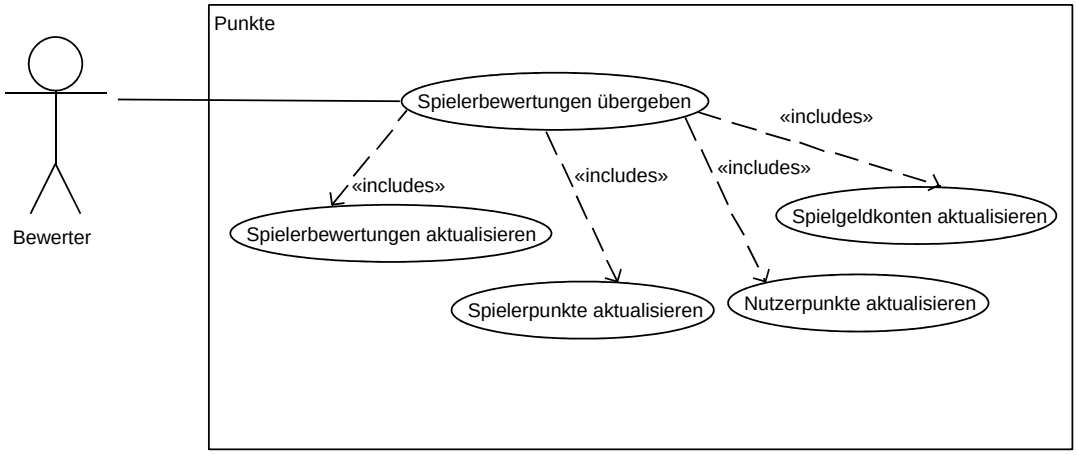


Abb. 2.3 Anwendungsfalldiagramm des Punktebereichs

Prämissen

Name:	punkte-p-10
Text:	Die Schnittstelle aus punkte-a-10 wird nach jedem Spieltag aufgerufen und ihr werden alle Spielerbewertungen des Spieltages übergeben.
Abhängigkeit:	punkte-a-10, punkte-l-10

Anforderungen

Name:	punkte-a-10
Text:	Das System muss eine Schnittstelle für das Einspielen der extern erzeugten Spielerbewertungen bereitstellen.

Name:	punkte-a-20
Text:	Das System muss nach jedem Spieltag zu den bisher erreichten Punkten jedes bewerteten Spielers die Punkte gemäß Abb. 2.4 hinzurechnen. Wurde ein Spieler nicht bewertet, ändert sich sein Punktestand nicht.

[3] Die Zeitschrift „Kicker“ (www.kicker.de) verteilt diese Bewertungen gegen Gebühr über eine Schnittstelle. Im Fallbeispiel werden zufällige aber sinnvolle Werte generiert und eingepflegt.

Note	1	1,5	2	2,5	3	3,5	4	4,5	5	5,5	6
Punkte	12	10	8	6	4	2	0	-2	-4	-6	-8

Abb. 2.4 Note-Punkte-Funktion

Name:	punkte-a-30
Text:	Das System muss nach jedem Spieltag zu den bisher erreichten Punkten jedes Nutzers die Punkte gemäß Abb. 2.4 für alle Spieler hinzurechnen, die sich aktuell in dessen Mannschaft befinden.

Name:	punkte-a-40
Text:	Das System muss nach jedem Spieltag die hinzugerechneten Punkte aus punkte-a-30 mit 10.000 multiplizieren und dem aktuellen Spielgeldbetrag des Nutzers hinzurechnen.
Abhängigkeit:	punkte-a-30

Name:	punkte-a-50
Text:	Das System muss nach jedem Spieltag die Durchschnittsnote jedes Spielers neu berechnen.

Das Szenario aus Abb. 2.5 zeigt beispielhaft die Abrechnung nach einem Spieltag für einen Nutzer, der neun Spieler S1-S9 in seiner Mannschaft führt. Nachdem die Spieler benotet wurden, wobei nur S6 nicht zum Einsatz kam, erhält der Nutzer die entsprechende Punktesumme. Diese Summe wird mit 10.000 multipliziert und dem Nutzer als Spielgeldbetrag angerechnet.

	S1	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**Nutzer**
Note	1,5	3	2	4,5	5	-	3	3	2	
Punkte	10	4	8	-2	-4	0	4	4	8	32
Spielgeld										320.000

Abb. 2.5 Punkteabrechnung für Spieler und Nutzer nach einem Spieltag

Leistungsausgrenzungen

Name:	punkte-l-10
Text:	Das System nimmt die Spielerbewertung nicht vor.

2.3 Nichtfunktionale Anforderungen

Im Rahmen der Fallstudie entfallen die Qualitätsanforderungen, die in einem realen Projekt vorhanden wären. Betroffen sind Zuverlässigkeits-, Benutzungsfreundlichkeits-, Effizienz-, Wartbarkeits- und Portabilitätsanforderungen. Ebenso werden Anforderungen an die Betreibbarkeit (z. B. Überwachung, Aktualisierungen, Datensicherung und -wiederherstellung) sowie gesetzliche Anforderungen (z. B. Datenschutz) nicht berücksichtigt.

Da das Ziel dieser Fallstudie nicht der produktive Betrieb des Spiels ist, sondern zur Untersuchung der modellgetriebenen Entwicklungswerkzeuge dient, existieren umgekehrt bereits Vorgaben für einzusetzende Technologie und Architektur, über die im realen Projekt meist später im Software-Lebenslauf entschieden würde. Dadurch werden einige der fehlenden Qualitätsanforderungen bereits impliziert.

2.3.1 Architekturvorgaben

Name:	nf-architektur-10
Text:	Das System muss die mehrschichtige, serverbasierte Architektur der Java 2 Platform, Enterprise Edition (J2EE) nutzen.

Name:	nf-architektur-20
Text:	Als Clients für die Nutzerinteraktion muss das System Webbrowser unterstützen.
Abhängigkeit:	nf-architektur-10

2.3.2 Technologievorgaben

Name:	nf-technologie-10
Text:	Die Benutzeroberfläche der Software muss mit der Hypertext Markup Language (HTML) beschrieben werden und darf Javascript verwenden. Die Client-Server-Kommunikation erfolgt somit über das Hypertext Transfer Protocol (HTTP).
Abnahme:	Die Darstellung muss im Mozilla Firefox 1.5 unter Windows XP korrekt erfolgen.
Abhängigkeit:	nf-architektur-20

Name:	nf-technologie-20
Text:	Das System muss eine relationale Datenbank als zentralen Datenspeicher nutzen.

Name:	nf-technologie-30
Text:	Das System muss die Schnittstelle zur Übergabe der Spielerbewertungen als Webservice implementieren. Die Kommunikation muss mittels Simple Object Access Protocol (SOAP) über HTTP erfolgen.
Abhängigkeit:	punkte-a-10

Name:	nf-technologie-40
Text:	Als Entwicklungsumgebung muss Eclipse ab Version 3.1 eingesetzt werden.

Name:	nf-technologie-50
Text:	Die Entwicklung muss weitgehend modellgetrieben erfolgen

2.4 Glossar

Gebot Ein Gebot ist ein Kaufangebot für einen **Spieler** am **Transfermarkt**, das sich stets auf genau eine **Offerte** bezieht.

Ligatabelle Die Ligatabelle ist die Rangliste der **Mannschaften** innerhalb einer **Miniliga**.

Mannschaft Eine Mannschaft ist die Menge der vom **Nutzer** am **Transfermarkt** gekauften **Spieler**.

Miniliga Eine Miniliga ist eine Zusammenfassung von höchstens 12 **Mannschaften**.

Nutzer Ein Nutzer ist die Person, die Fooma spielt.

Offerte Eine Offerte ist ein Verkaufsangebot am **Transfermarkt** für genau einen **Spieler**.

Richtwert Der Richtwert einer **Offerte** ist der Mindestbetrag für **Gebote**, also der Spielgeldbetrag, den ein bietender **Nutzer** mindestens zahlen muss.

Spieler Ein Spieler ist ein realer Fußballspieler der ersten Fußball Bundesliga.

Spieltag Ein Spieltag ist ein realer Spieltag der ersten Fußball Bundesliga.

Transfer Ein Transfer ist der Übergang eines **Spielers** entweder von der **Mannschaft** eines **Nutzers** oder vom System zur **Mannschaft** eines anderen **Nutzers** innerhalb derselben **Miniliga**.

Transferbetrag Der Transferbetrag ist der Spielgeldbetrag eines angenommenen **Gebotes**.

Transfermarkt Der Transfermarkt ist die Plattform, auf der **Nutzer** innerhalb einer **Miniliga** die **Spieler** kaufen oder zum Verkauf anbieten.

3 Java 2 Platform, Enterprise Edition

Die von Sun Microsystems[4] initiierte Java 2 Platform, Enterprise Edition[5] (J2EE) koordiniert eine Menge von Spezifikationen zur Definition einer Standardarchitektur und -plattform, mit der mehrschichtige, serverbasierte Geschäftsanwendungen entwickelt, eingesetzt und verwaltet werden können. Sie legt die Schnittstellen und Verträge fest, über die Plattformanbieter Dienste wie Datenbankzugriff, Webzugriff, Sicherheit, Transaktionen, Nachrichtenaustausch usw. erbringen bzw. über die Anwendungsentwickler diese Dienste nutzen.

Neben der Plattformspezifikation ([Sun03a]) enthält die J2EE eine Referenzimplementierung sowie einen Kompatibilitätstest, der es Plattformanbietern ermöglicht, ihre Applikationsserver auf Konformität mit der Spezifikation zu prüfen.

Abbildung 3.1 zeigt die wichtigsten Elemente der mehrschichtigen Architektur. Charakteristisch ist das Container-Konzept. Ein Container ist die Laufzeitumgebung für dazu passende Komponenten. Er verwaltet diese und stellt ihnen seine Dienste zur Verfügung, wofür er selbst zahlreiche Basisdienste kapselt, integriert und nutzt. Diese Basisdienste sind z. T. schon in der Java 2 Platform, Standard Edition (J2SE), dem Java-Kern, enthalten.

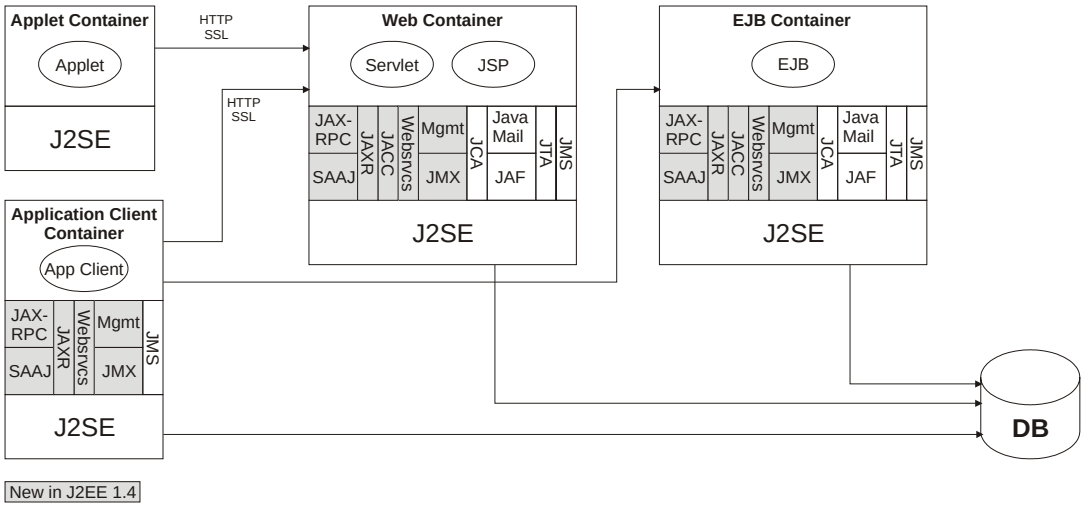


Abb. 3.1 J2EE Container Architektur (nach [Sun03a], S. 6)

Zwar nennt die J2EE-Spezifikation vier Container-Typen, im Mittelpunkt stehen jedoch die serverseitig eingesetzten Web- und Enterprise JavaBeans-Container (EJB-Container). Der Application Client Container stellt zwar Basisdienste zur Verfügung, schränkt aber die sie verwendenden Komponenten in keiner Weise ein und beeinflusst auch nicht deren Lebenszyklus. Diese sind also beliebige Java-Programme. Ein Webbrowser mit Java Plug-In, einer Kombination, die es bereits seit der ersten[6] Java-

[4] http://www.sun.com/

[5] Hier wird Version 1.4 betrachtet

[6] Das Applet-Konzept wurde bereits mit Oak, wie Java vor seiner offiziellen Veröffentlichung 1995 hieß, umgesetzt (siehe auch: http://java.com/en/javahistory/timeline.jsp).

Version erlaubt, Java-Programme (Applets) innerhalb des Browsers auf dem Client auszuführen, bildet den Applet Container. Da dieser lediglich den HTTP-Zugriff auf den Server ermöglichen und somit im J2EE-Einsatz keine zusätzlichen Dienste anbieten muss, bedarf er keiner weiteren Betrachtung.

Vor der Beschreibung der Web- und EJB-Container soll kurz auf das Fundament J2SE und die darin enthaltenen bzw. darauf aufbauenden Basisdienste eingegangen werden.

3.1 Java 2 Platform, Standard Edition

Die J2SE wird in zwei Varianten angeboten. Die Java Runtime Environment (JRE) beinhaltet die Java Virtual Machine (JVM) für das jeweilige Betriebssystem sowie die Klassenbibliothek zur Erbringung der Basisdienste und ist die fundamentale Laufzeitumgebung für alle[7] Java-Programme. Das J2SE Development Kit (JDK) liefert darüber hinaus Compiler, Werkzeuge und API-Dokumentation.

Viele der in der J2SE enthaltenen Basisdienste[8] erhalten im Rahmen der J2EE besonderes Gewicht, wovon einige im folgenden kurz beschrieben werden.

3.1.1 Java Database Connectivity

Eine einheitliche API für den Zugriff auf relationale Datenbanken zu schaffen war Ziel der Entwicklung von Java Database Connectivity (JDBC). Schon früh erfuhr JDBC eine breite Unterstützung seitens der Datenbankanbieter, und so stehen heute für fast alle marktrelevanten relationalen Datenbanken JDBC-Treiber zur Verfügung[9].

3.1.2 Java Naming and Directory Interface

Java Naming and Directory Interface (JNDI) bietet eine API für den Zugriff auf Verzeichnisdienste. Eine Implementierung für den Zugriff über Lightweight Directory Access Protocol (LDAP) ist bereits enthalten.

3.1.3 Java API for XML Parsing

Für den Umgang mit XML-Dokumenten existieren Standards wie Document Object Model (DOM) und Simple API for XML (SAX) für das Parsen sowie Extensible Stylesheet Language Transformation (XSLT) für die Transformation. Java API for XML Parsing (JAXP) ermöglicht, verschiedene Parser[10] und Transformatoren[11] auf einheitliche Weise einzusetzen.

3.1.4 Java Authentication and Authorization Service

Java Authentication and Authorization Service (JAAS) implementiert das Pluggable Authentication Module (PAM) Konzept in Java und erweitert es um benutzer-, grup-

[7] Die J2ME (Java 2 Platform, Micro Edition) beschneidet JVM und Klassenbibliothek für den Einsatz auf leistungsschwachen Geräten (PDA, Mobiltelefon usw.). Sie wird hier aber nicht berücksichtigt.

[8] Viele Dienste (JDBC, JNDI, JAXP, …) begannen als optionale Zusatzbibliotheken und wurden erst später in den Kern aufgenommen. Die Ausführungen hier beziehen sich auf J2SE 1.4.

[9] siehe auch: http://java.sun.com/products/jdbc/industry.html

[10] z. B. Xerces (http://xerces.apache.org/xerces-j/), Crimson (http://xml.apache.org/crimson/)

[11] z. B. Xalan (http://xml.apache.org/xalan-j/), Saxon (http://www.saxonica.com/)

pen- oder rollenbasierte Autorisierung. PAM erlaubt es, Anwendungen, Authentisierungsdienste zu nutzen, ohne die dazu eingesetzten Methoden (Benutzername und Passwort, Zertifikate, …) oder Implementierungen (Abfrage von Datenbanken, Verzeichnisdiensten, Dateien, Smartcards, …) unterstützen zu müssen. Letztere werden als austauschbare Module eingeklinkt, die in jeder PAM-kompatiblen Anwendung einsetzbar sind.

3.1.5 Remote Method Invocation

Die Java-Variante von Remote Procedure Call (RPC) nennt sich Remote Method Invocation (RMI) und ermöglicht den Aufruf von Methoden eines Java-Objekts, das im Adressraum einer anderen JVM und ggf. eines anderen Hosts vorliegt. Neben dem Java Remote Method Protocol (JRMP), das auf die Kommunikation zwischen Java-Objekten beschränkt ist, ermöglicht RMI-IIOP (Internet Inter Orb Protocol) auch die Kommunikation mit entfernten CORBA-Objekten (Common Object Request Broker Architecture ([OMG04])).

3.2 Basisdienste außerhalb der J2SE

Ein J2EE-Applikationsserver bietet Basisdienste, die nicht Bestandteil der Standardedition sind. Einige davon sollen hier ebenfalls kurz skizziert werden. Eine vollständige Auflistung bietet Abb. 3.2.

3.2.1 J2EE Connector Architecture

Was mit JDBC für relationale Datenbanken erreicht wurde, nämlich eine produktunabhängige Zugriffsmöglichkeit zu schaffen, soll mit der J2EE Connector Architecture (JCA) auch bei beliebigen Back-End-Systemen, im J2EE-Jargon Enterprise Information Systems[12] (EIS) genannt, gelingen[13]. Wie bei JDBC wird einerseits eine API für den Zugriff auf ein beliebiges EIS definiert, andererseits ein Service Provider Interface (SPI) festgelegt, das für ein konkretes EIS von einem entsprechenden Treiber implementiert werden muss. JCA spezifiziert die Verträge zwischen J2EE-Applikationsserver und EIS bzgl. Verbindungs-, Transaktions-, Sicherheits- und Threadmanagement.

3.2.2 Java APIs for XML-based Remote Procedure Call

Die Unterstützung von Webservices als integraler Bestandteil der J2EE war ein prägendes Merkmal der Version 1.4. Java APIs for XML-based Remote Procedure Call (JAX-RPC) ermöglicht sowohl den Zugriff auf als auch die Bereitstellung von Webservices mit Java.

3.2.3 Weitere Basisdienste

Weitere Dienste, die ein J2EE-Applikationsserver bereitstellt, sind u. a. JavaMail für das Senden und Empfangen von Emails, Java Message Service (JMS) für die Kopplung von Systemen mittels Nachrichten über Punkt-zu-Punkt- oder Publish-Subscribe-

[12] Datenbanken werden häufig ebenfalls unter dem Begriff EIS zusammengefasst. So kann JCA als Verallgemeinerung von JDBC betrachtet werden.

[13] Es existieren bereits zahlreiche JCA-Treiber für gängige Systeme wie SAP R/3 (siehe auch: http://java.sun.com/j2ee/connector/products.html)

Verbindungen wählbarer Dienstgüte sowie Java Transaction API (JTA) zur Steuerung von (verteilten) Transaktionen.

3.3 Übersicht der Basisdienste und Container

Abbildung 3.2 zeigt, welche Basisdienste, die nicht bereits in der J2SE enthalten sind, die jeweiligen Container eines J2EE-konformen Applikationsservers anbieten müssen. Die containerspezifischen Dienste sind hier nicht enthalten.

Container / **Basisdienst**	App Client	Applet	Web	EJB
JMS 1.1	J	N	J	J
JTA 1.0	J	N	J	J
JavaMail 1.3	J	N	J	J
JAF 1.0	J	N	J	J
JAXP 1.2[14]	J	N	J	J
JCA 1.5	N	N	J	J
Web Services 1.1	J	N	J	J
JAX-RPC 1.1	J	N	J	J
SAAJ 1.2	J	N	J	J
JAXR 1.0	J	N	J	J
J2EE Management 1.0	J	N	J	J
JMX 1.2	J	N	J	J
JACC 1.0	N	N	J	J

Abb. 3.2 Angebot an Basisdiensten der verschiedenen J2EE-Container (nach [Sun03], S. 86f)

3.4 Enterprise JavaBeans[15] Container

Der EJB-Container steht im Zentrum der J2EE-Architektur. Er bietet dem EJB-Entwickler (Enterprise Bean Provider ([Sun03a], S. 36)) ein Programmiermodell, das es erlaubt, von Sicherheits-, Verteilungs-, Transaktions- und Persistenzaspekten zu abstrahieren. Diese werden getrennt in Deskriptoren deklariert. Deskriptoren sind XML-Dateien, die jeder EJB (letztendlich eine Java-Klasse, die bestimmte Schnittstellen implementiert) zur Seite gestellt werden.

Anders als der Web-Container, der seinen Komponenten größtenteils nur die APIs der Basisdienste aus Abb. 3.2 und der J2SE bereitstellen muss, kapselt und integriert der

[14] In der J2SE ist lediglich JAXP 1.1 enthalten. Version 1.2 verlangt zusätzlich von Parsern die Fähigkeit, XML-Dokumente gegen XMLSchema validieren zu können.

[15] [Sun03a]

EJB-Container diese zu höherwertigen Diensten. So kommen EJB-Entwickler ggf. mit keiner dieser APIs in Berührung, müssen dafür aber auch Einschränkungen bei der Programmierung hinnehmen, um keine Verträge mit dem Container zu brechen (s. [Sun03a], S. 562 ff).

Die Spezifikation ([Sun03a]) unterscheidet drei Typen von Enterprise JavaBeans, die jeweils typische Anforderungen bei der Entwicklung von Geschäftsanwendungen abdecken sollen.

- Eine **Entity Bean** stellt eine persistente Entität dar, deren Zustand in einer relationalen Datenbank gespeichert wird. Die von Entity Beans angebotene Schnittstelle besteht i. d. R. lediglich aus Zugriffsmethoden für die Attribute sowie Methoden zum Erzeugen, Abfragen und Löschen. Soll der Container für die Persistenz sorgen, müssen die Informationen für die Abbildung des Zustandes der Entity Bean auf Tabellen in der Datenbank im zugehörigen Deskriptor abgelegt sein. Diesen Fall nennt man Container Managed Persistence (CMP). Nimmt jedoch die Entity Bean dies selbst mittels JDBC vor, so spricht man von Bean Managed Persistence (BMP). Clients einer Entity Bean können Application Clients, Servlets oder andere EJBs sein (vgl. Abb. 3.1).
- Methoden einer **Session Bean** implementieren die Geschäftslogik. Dazu ruft sie ggf. weitere Enterprise JavaBeans auf (Session oder Entity Beans) oder nutzt direkt JDBC oder JCA, um auf das Back-End zuzugreifen. Eine Session Bean ist immer transient und kann zustandslos (**Stateless Session Bean**) oder zustandsbehaftet (**Stateful Session Bean**) sein. Methoden einer Stateless Session Bean können als Endpunkte für Webservices deklariert werden. Somit können als Client, neben Application Clients, Servlets und anderen EJBs, beliebige Webservice Clients fungieren. Von der tatsächlichen Zugriffsmethode kann der Entwickler bei der Programmierung abstrahieren.
- Eine **Message Driven Bean** wird bei Eintreffen einer Nachricht (über JMS) ausgeführt und ist immer zustandslos (aus Sicht des Client) und transient. Sie dient der asynchronen Kommunikation und kann nicht direkt aufgerufen werden.

Allen EJB-Typen ist gemein, dass jeder Zugriff ausschließlich über den EJB-Container erfolgt. Nur so kann Verteilungstransparenz aus Sicht des Clients, Transaktionssteuerung und Einhaltung von Sicherheitsrichtlinien vom Container gewährleistet werden. Dies hat zur Folge, dass EJB-Aufrufe im Vergleich zu gewöhnlichen Methodenaufrufen sehr „teuer“ sind. Auch die vielen mit einer einzelnen EJB verbundenen Artefakte (mehrere Schnittstellen, eine implementierende Klasse, vom EJB-Container generierte Klassen, ggf. mehrere Deskriptoren) erhöhen sowohl den Aufwand für die Laufzeitumgebung als auch für den Entwickler. Da die EJB-Architektur ihre Vorteile anscheinend in weit weniger Anwendungsfällen ausspielen kann als erwartet, ist ihr Gebrauch stets abzuwägen (vgl. [JoHo04]).

3.5 Web Container

Der Web-Container ist mit einem Webserver integriert und beherbergt als Laufzeitkomponenten Servlets ([Sun03b]), an die er die angenommenen HTTP-Anfragen leitet und die von den Servlets erzeugten Antworten an den Client zurücksendet. Für den Servlet-Entwickler stellt der Web-Container eine API[16] bereit, deren wichtigste Elemente folgende Basisklassen und Schnittstellen sind:

- `HttpServlet` ist die abstrakte Basisklasse, die der Entwickler ableitet, um HTTP-Anfragen verarbeiten zu können.
- Die `HttpServletRequest`-Schnittstelle bietet eine objektorientierte Sicht auf eine HTTP-Anfrage. Über sie lassen sich Anfrageparameter abfragen, die z. B. durch ein HTML-Formular übergeben wurden.
- Analog kapselt ein `HttpServletResponse` die Antwort des Webservers. Über diese Schnittstelle kann der Entwickler hauptsächlich Einfluss auf den HTTP-Header nehmen und insbesondere den MIME-Typ (Multipurpose Internet Mail Extensions) der Ausgabe, z. B. `text/html` setzen.
- Über die `HttpSession` Schnittstelle ermöglicht der Web-Container die Verwaltung von auf dem Server gespeicherten Sitzungsdaten. Dadurch erweitert er das zustandslose HTTP um das Konzept der Sitzung.

Das Sequenzdiagramm in Abb. 3.3 veranschaulicht den typischen Ablauf bei der Verarbeitung einer HTTP-Anfrage. Das Servlet empfängt die Anfrage gekapselt als `HttpServletRequest` Objekt. Aus diesem entnimmt es die Anfrage- und Sitzungsparameter, die es zur Verarbeitung braucht, und generiert die Ausgabe (meist HTML), indem es den MIME-Typ der Ausgabe festlegt und in den `PrintWriter` des `HttpServletResponse` schreibt.

[16] Die Servlet API teilt sich in einen protokollunabhängigen und einen HTTP-spezifischen Bereich, sie ist also grundsätzlich nicht auf HTTP beschränkt, findet dort aber die weitaus häufigste Anwendung.

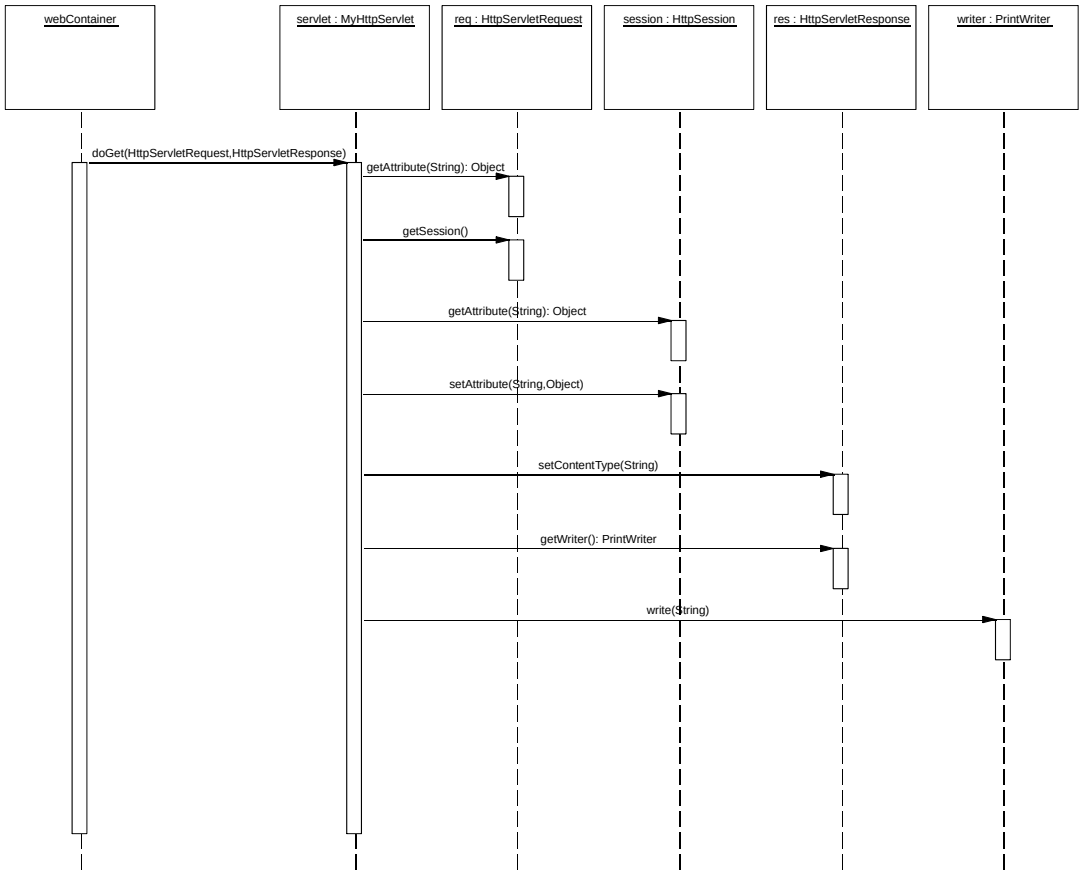


Abb. 3.3 Typische Verarbeitung einer HTTP-Anfrage durch ein Servlet

Auch zu Servlets gehört ein XML-Deskriptor, in dem u. a. festgelegt wird, für welches URL-Muster ein Servlet zuständig ist, damit der Web-Container die Anfragen zuordnen kann. Ebenfalls über URL-Muster teilt man dem Web-Container mit, welche Bereiche eine Authentisierung benötigen und wie diese erfolgen soll. Zur Wahl stehen HTTP Basic Authentication (Benutzername und Passwort werden unverschlüsselt übertragen), HTTP Digest Authentication (Benutzername und Passwort werden als Hash[17] übertragen), HTTPS Client Authentication (Zertifikate) und formularbasierte Authentisierung (Benutzername und Passwort werden über ein HTML-Formular abgefragt). Eine weitere Anwendung der URL-Muster im Deskriptor erlaubt es, den Zugriff auf die entsprechenden URLs nur bestimmten Benutzerrollen zu erlauben.

Als weiteres Konzept bietet der Web-Container *ServletFilter* an. Diese können verkettet und vor ein Servlet gestellt werden. Jede für dieses Servlet bestimmte HTTP-Anfrage durchläuft so die Filterkette vor Erreichen des Servlet. Analog wird die HTTP-Antwort des Servlet in umgekehrter Reihenfolge durch die Filter geleitet. Ein ServletFilter kann eine Anfrage auch selbst beantworten, ohne sie an den nächsten Filter bzw. das Servlet

[17] Es wird ein einfaches Challenge-Response-Verfahren eingesetzt. Der Server sendet einen Zufallswert an den Client, der dort zusammen mit Benutzername und Passwort einer Hashfunktion übergeben wird. Das Ergebnis wird an den Server übermittelt, der ebenfalls einen Hashwert mit dem bei ihm gespeicherten Passwort, dem Benutzernamen und dem Zufallswert berechnet. Stimmen beide Hashwerte überein, ist der Nutzer authentisiert (siehe auch: RFC 2617).

weiterzureichen. Häufig werden Aspekte wie Kompression, Caching oder Logging in Filtern gekapselt.

3.5.1 JavaServer Pages

Da bei Servlets HTML[18]-Code in den Java-Quellen eingebettet ist, ist dessen Wartung sehr aufwendig. Dieses Problem sollten JavaServer Pages (JSP, [Sun03c]) in den Griff bekommen, die dem entgegengesetzten Weg folgen und Javacode in HTML einbetten. Zur Laufzeit unterscheiden sie sich nicht von Servlets, da sie vor ihrem Einsatz in ein Servlet übersetzt werden[19]. Den nun schlecht zu wartenden Javacode sollen jetzt *Tag Extensions* ersetzen, eingebetteter Javacode wird aber weiter unterstützt. Tag Extensions ermöglichen dem Entwickler, eigene Tags zu deklarieren und deren Semantik mit einem *TagHandler* in Java zu implementieren. Die TagHandler unterliegen der Verwaltung des Web-Containers.

Das Beispiel aus Abb. 3.4, das den Inhalt eines Warenkorbs als HTML-Seite ausgibt, soll Syntax und grundlegende Konzepte von JSP verdeutlichen.

In Zeile 1 bestimmt eine *JSP-Direktive* den MIME-Typ der erzeugten Ausgabe. Im Servlet, das aus dieser JSP resultiert, wird diese Direktive in einen entsprechenden Aufruf von `setContentType()` des `HttpServletResponse` übersetzt (vgl. Abb. 3.3).

In Zeile 2 wird eine *Tag Library* deklariert. In einer Tag Library werden mehrere Tag Extensions zusammengefasst. Diese Information sowie die Zuordnung von Tag Extension zu implementierendem TagHandler ist wie üblich in einem XML-Deskriptor (Tag Library Descriptor, TLD) untergebracht. Um Konflikte zwischen gleichnamigen Tags unterschiedlicher Bibliotheken zu vermeiden, wird jede Tag Library mit einer URI (Uniform Resource Identifier) verknüpft und in der JSP über ein Präfix referenziert.

Zeile 3 deklariert ein Objekt `cart` vom Typ `de.hardan.jspexample.Cart`, das dem Sitzungsspeicher (mittels `HttpSession.getAttribute()`) entnommen wird.

In Zeile 7 kommt schließlich eine Tag Extension zum Einsatz. Das Präfix `c` verweist auf die URI in Zeile 2 und identifiziert so die Tag Library, die das Tag `forEach` enthält.

```
 1 <%@ page contentType="text/html; charset=ISO-8859-1" %>
 2 <%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
 3 <jsp:useBean id="cart" type="de.hardan.jspexample.Cart" scope="session"/>
 4 <html>
 5   <head><title>Cart Items</title></head>
 6   <body>
 7     <c:forEach var="item" items="cart.products">
 8       <c:out value="item" />
 9       <br>
10     </c:forEach>
11   </body>
12 </html>
```

Abb. 3.4 Beispiel einer JSP mit Tag Extensions

[18] Servlets können beliebige Ausgaben erzeugen, während JSP nur in Verbindung mit HTML/XHTML und XML sinnvoll einzusetzen sind.

[19] Die meisten Web-Container erlauben auch den Einsatz von JSP zur Laufzeit und übersetzen sie selbst in Servlets vor dem ersten Zugriff. JSP können dann als weitere Laufzeitkomponente im Web-Container betrachtet werden.

3.6 Neuerungen der J2EE-Folgeversion

Die kommende Version 5 der Java Platform, Enterprise Edition (Java EE[20]) enthält weitgehende Änderungen und Ergänzungen, von denen einige bereits zur Umsetzung des Fallbeispiels dienen.

3.6.1 Java Persistence API

Im Kern der Java EE bricht die neue, bereits verabschiedete EJB-Spezifikation 3.0, was Entity Beans betrifft, mit dem Container-Konzept. Entity Beans dienen dem Speichern von Objekten in relationalen Datenbanken, eine Aufgabe, die bisher der EJB-Container übernahm. Nun erledigt dies ein über die Java Persistence API (JPA, [Sun06c]) anzusprechender „Persistence-Provider“ ([Sun06d], S. 33). Die JPA ist angetreten, einen einheitlichen Standard zur Persistenz von Java-Objekten zu schaffen und so Java Data Objects (JDO), EJB-CMP und etliche Frameworks abzulösen. Sie ist also nicht auf den Einsatz innerhalb der Java EE beschränkt ([Sun06c], S. 15).

3.6.2 JavaServer Faces

Für die Entwicklung dynamischer Weboberflächen enthalten die Servlet- und JSP-Technologien viele Lücken, die zahlreiche Frameworks zu füllen versucht haben. JavaServer Faces (JSF) strebt auch hier nach Vereinheitlichung und unterstützt fehlende Konzepte wie das Binden von HTML-Formularfeldern an Objektattribute, die Typkonvertierung und Validierung von Benutzereingaben oder die Verwaltung des Zustandes der Benutzeroberfläche über HTTP-Anfragen hinweg. Außerdem bietet es dem Entwickler eine erweiterbare Menge wiederverwendbarer und konfigurierbarer Komponenten für den Bau von Benutzeroberflächen, wie es schon aus dem Desktopbereich bekannt ist. JSF baut dabei auf Servlets auf und lässt sich in JSP integrieren.

3.6.3 Konsequenzen für die Implementierung des Fallbeispiels

Mit dem Herauslösen der Entity Beans aus dem EJB-Container und der Übertragung der Verantwortung für die Persistenzbelange an die JPA zeigt sich, dass zumindest für diesen Bereich das Container-Konzept unzulänglich ist. Die Entscheidung, Entity Beans dem EJB-Container zu entziehen, trägt lediglich der Entwicklung Rechnung, dass viele Projekte Entity Beans (nach EJB 2.1) oder den EJB-Container als Ganzes zugunsten individueller Lösungen umgehen, insbesondere, wenn nicht alle Dienste des EJB-Containers benötigt werden. Auch für das Fallbeispiel sind die Einschränkungen und die hohe Komplexität bei der Entwicklung von Entity Beans bisheriger Prägung ungerechtfertigt, weshalb hierfür die JPA aus der neuen EJB-Spezifikation zum Einsatz kommt.

Im Falle von JSF handelt es sich nicht um eine Abkehr von bestehenden Konzepten, sondern um eine Erweiterung. Die Servlet-Architektur des Web-Containers ist in java-basierten Webanwendungen allgegenwärtig, auch JSP fand weite Verbreitung. Dennoch bieten beide lediglich Basisdienste. Für die Entwicklung komplexer Systeme notwendige Funktionen werden jedoch meist als Ergänzungen und nicht als Ersatz entworfen, so

[20] Name und Abkürzung der Folgeversionen der Java 2 Platform, Enterprise Edition (J2EE). Analog heißt die Java 2 Platform, Standard Edition (J2SE) nun Java SE. Siehe auch http://java.sun.com/javaee/index.jsp

auch JSF. Die oben erwähnten und in Kapitel 7 näher beschriebenen Funktionen unterstützen ebenfalls die Implementierung des Fallbeispiels.

4 Eclipse und das Eclipse Modeling Framework

Das Eclipse-Projekt[21] wurde von IBM ins Leben gerufen, um als Integrationsplattform für die Vielzahl der dort intern eingesetzten Entwicklungswerkzeuge zu dienen. Eclipse fußt auf einem Plug-In-Konzept, über das sämtliche Funktionalität bereitgestellt wird. Plug-Ins können Basisdienste (selbst Plug-Ins) nutzen, erweitern und an definierten Punkten selbst genutzt und erweitert werden. Basisdienste helfen u. a. beim Bau konsistenter Benutzeroberflächen, bieten ein Workspace-Konzept für die Verwaltung aller von Werkzeugen benötigter Artefakte und liefern Frameworks für den Bau von Editoren, Compilern oder Debuggern ([DFK+05], S. 3 ff).

Ab November 2001 erfolgte die Weiterentwicklung als Open-Source-Projekt unter der Leitung des eigens gegründeten Eclipse-Konsortiums, dem neben IBM sieben weitere Unternehmen angehörten ([DFFK05], S. 1). Nicht nur das Konsortium wuchs auf über einhundert[22] Mitglieder, auch das Projekt entwickelte sich rasant und wurde nicht nur um zahlreiche Plug-Ins erweitert, sondern im Kern weiter modularisiert. So ist dessen Einsatz nicht mehr auf die Erstellung integrierter Entwicklungsumgebungen beschränkt, sondern bietet sich für die Entwicklung beliebiger Desktop-Anwendungen an. Dazu ist Eclipse in die Rich Client Platform (RCP) für allgemeine Anwendungen und die Workbench IDE für Entwicklungsumgebungen getrennt worden, wobei die IDE auf der RCP aufsetzt.

Eines der prominentesten Plug-Ins ist das Java Development Tooling (JDT), eine Java-Entwicklungsumgebung, die auch als Beweis für die Tauglichkeit der Plug-In-Architektur und Basisdienste für komplexe Aufgaben dient[23]. Im Bereich der Modellierung bietet Eclipse mit dem Eclipse Modeling Framework (EMF) eine Möglichkeit, u. a. Metamodelle zu spezifizieren und darauf aufbauend Modelleditoren zu generieren, welche die Einhaltung der im entsprechenden Metamodell definierten Syntaxregeln gewährleisten. Alle erzeugten Metamodelle besitzen mit Ecore ein gemeinsames Meta-Metamodell, das einer Untermenge der MOF ([OMG02a]), EMOF (Essential MOF), entspricht. Somit stellt EMF mit Ecore die nötige Integrationsebene, um Instanzen verschiedener Metamodelle ineinander zu überführen. Durch die Verwandtschaft zu MOF lehnt es sich außerdem an die MDA an und ermöglicht so die Einbindung dort vorhandener oder entstehender Konzepte.

Die Erweiterbarkeit von Eclipse durch die Plug-In-Architektur, die ausgereiften Entwicklungswerkzeuge des JDT für Java-Anwendungen und die Modellierungsfunktionen von EMF bieten ein leistungsstarkes, integriertes Fundament für die Umsetzung des Fallbeispiels. Jedoch erhält man von Eclipse kaum Unterstützung bei der Modelltransformation, dem zentralen Mechanismus in der modellgetriebenen Entwicklung. Dies leistet das in Kapitel 6 vorgestellte Plug-In, das eine Transformationssprache bereitstellt und EMF-Modelle verarbeitet.

Dieses Kapitel untersucht die allgemeinen Grundlagen von EMF weitgehend unabhängig von dessen im Fallbeispiel geforderten Einsatz zur Metamodellierung. Bezüge dazu werden, wenn möglich, hergestellt.

---

[21] http://www.eclipse.org/

[22] http://www.eclipse.org/membership/

[23] Eclipse ist selbst eine Java-Anwendung und dient als seine eigene Entwicklungsumgebung

4.1 Das EMF-Metamodell Ecore

EMOF bietet als Untermenge von MOF 2.0 nur die wesentlichen Elemente zur Modellierung objektorientierter Systeme ([OMG03c], S. 32) und weicht, bis auf Unterschiede in der Namensgebung, kaum von Ecore ab (vgl. [Ecli05a]). Dies ist darauf zurückzuführen, dass Ecore als Implementierung der MOF 1.4 begann, sich dann im Einsatz mit EMF weiterentwickelte und schließlich selbst zur Spezifikation von EMOF beitrug ([MeSt05], S. 14).

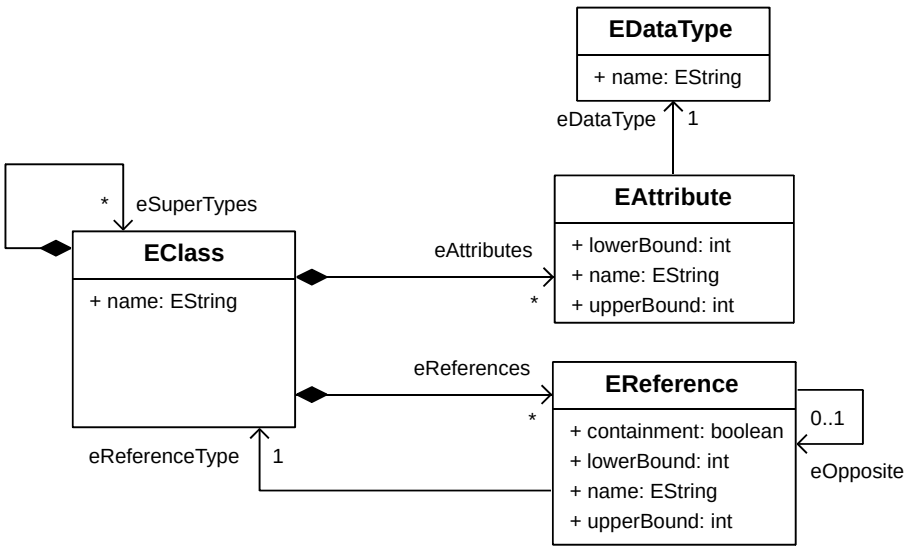


Abb. 4.1 Vereinfachtes Ecore-Metamodell (nach [BaGr05], S. 12)

In Abb. 4.1 sind Kernelemente von Ecore dargestellt. Trotz der Vereinfachung werden zusätzlich zu den Bezeichnern einige der Unterschiede zu MOF deutlich, die auch die oben skizzierte Evolution widerspiegeln. So entspricht `EReference` nicht der `Association` in MOF 1.4, welche ohne Entsprechung bleibt, sondern der `AssociationEnd`. Für eine bidirektionale Beziehung werden also zwei `EReference`-Objekte benötigt, die das jeweils andere als `eOpposite` referenzieren (vgl. [BaGr05], S. 36; [OMG02a], S. 3-50 ff.). Dieser Ansatz wird in EMOF wieder aufgegriffen. Dort tritt das Element `Property` an die Stelle der `EReference` und es besitzt ebenfalls eine Referenz `opposite` zu sich selbst (vgl. [OMG03b], S. 97). EMOF lässt jedoch als Typen für `Property` sowohl Datentypen (primitive Datentypen und Aufzählungstypen) als auch Klassen zu und benötigt deshalb nicht die Unterscheidung zwischen Attributen (Datentypen) und Referenzen (Klassen als Typ), wie sie in Ecore zu finden ist (`EAttribute`, `EReference`). Abbildung 4.2a und 4.2b vervollständigen die Ecore-Beschreibung.

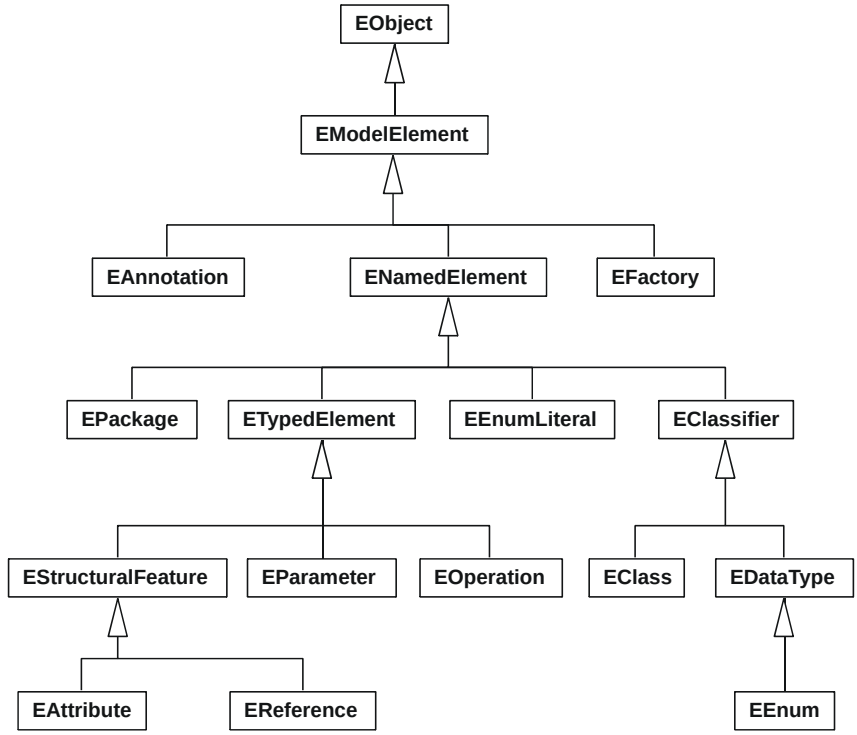


Abb. 4.2a Ecore Metamodell (nur Hierarchie, nach [Ecli05b])

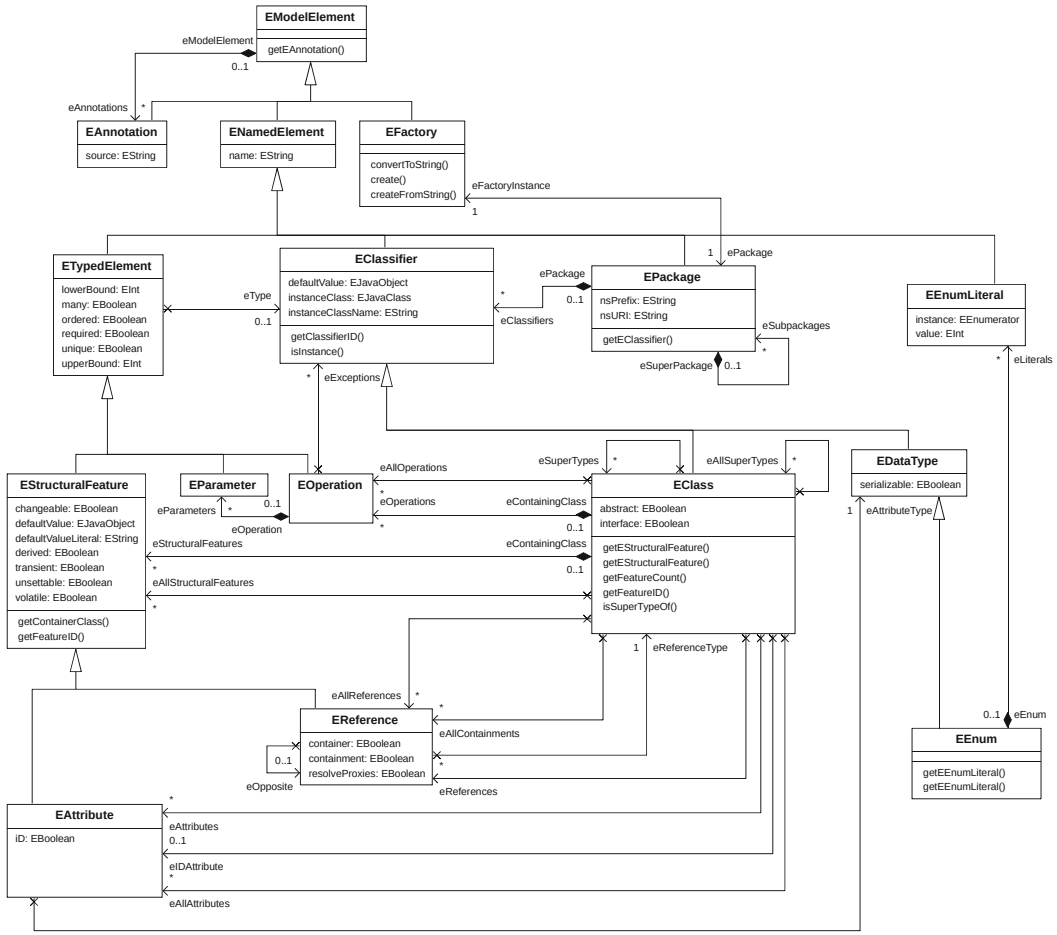


Abb. 4.2b Ecore Metamodell (nach [Ecli05b])

4.2 Definition der Ecore-Modelle

EMF zeigt sich vielseitig in den Möglichkeiten, Ecore-Modelle zu beschreiben. Das XML Metadata Interface (XMI) stellt dabei das kanonische Speicherformat für Ecore-Modelle dar (Abb. 4.3). EMF kann außerdem in XMI kodierte EMOF-Modelle lesen und schreiben ([Ecli05a]).

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    nsURI="http:///org/hardan/playground/emf/ecore/library.ecore"
    nsPrefix="org.hardan.playground.emf.ecore"
    name="library">
  <eClassifiers xsi:type="ecore:EClass" name="Book">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="title"
       eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="pages"
       eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
  </eClassifiers>
</ecore:EPackage>
```

Abb. 4.3 XMI-Serialisierung eines Ecore-Modells

Der mitgelieferte Ecore-Editor wurde selbst mit EMF generiert. Modelleditoren, die die Einhaltung der in einem Metamodell spezifizierten Strukturen und Einschränkungen sicherstellen, sind ein Ergebnis der Codeerzeugung mit EMF. So wurde Ecore nachträglich mit EMF definiert, es ist also sein eigenes Metamodell. Der in Abb. 4.4 dargestellte Editor ist dann aus dieser Definition generiert worden.

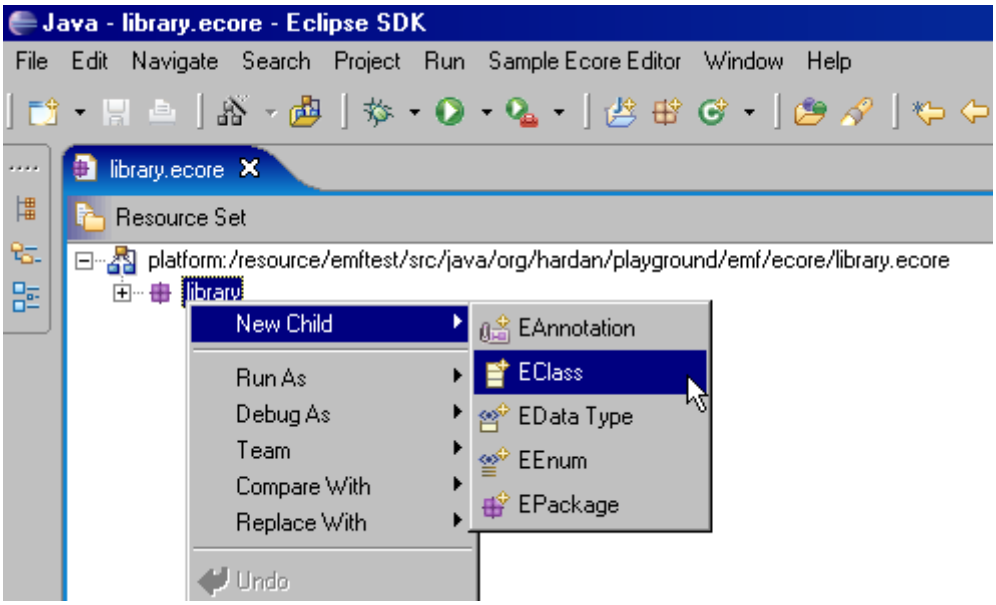


Abb. 4.4 Der Ecore-Editor des EMF

Eine weitere Beschreibungsmöglichkeit steht mit annotierten Java-Schnittstellen zur Verfügung. Die in Abb. 4.4 dargestellte XMI-Datei ließe sich auf diese Weise wie in Abb. 4.5 formulieren. Kapitel 6 beschreibt die neu in den Java-Sprachstandard aufgenommenen Annotationen detailliert.

```
/**
 * @model
 */
public interface Book {
    /**
     * @model
     */
    String getTitle();

    /**
     * @model
     */
    int getPages();
}
```

Abb. 4.5 Definition eines Ecore-Modells mit annotierten Java-Schnittstellen

EMF kann ebenfalls als XMLSchema[24] definierte Modelle in Ecore umwandeln (Abb. 4.6)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http:///org/hardan/playground/emf/ecore/library.ecore"
     xmlns="http:///org/hardan/playground/emf/ecore/library.ecore"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:complexType name="Book">
     <xsd:sequence>
       <xsd:element name="title" type="xsd:string"/>
       <xsd:element name="pages" type="xsd:integer"/>
     </xsd:sequence>
   </xsd:complexType>
</xsd:schema>
```

Abb. 4.6 XMLSchema zur Definition in Abb. 4.3

[24] http://www.w3.org/XML/Schema

4.3 Die Codeerzeugung

Den Ablauf bei der Erzeugung von Code skizziert Abb. 4.7. Dazu nötige zusätzliche Information bezieht EMF aus einem Generatormodell, welches das Ecore-Modell entsprechend anreichert. Mit dessen Hilfe werden die in den Java Emitter Templates definierten Codeschablonen ausgefüllt und die Plug-In-Projekte generiert.

Im Rahmen des Fallbeispiels dient dieser Schritt der Erzeugung von Metamodellklassen und zugehörigen Modelleditoren. Für jedes mit dem Ecore-Editor (Abb. 4.4) erstellte Metamodell generiert EMF einen spezifischen Editor als Eclipse-Plug-In und integriert ihn so in den Entwicklungsprozess. Die Erstellung der Editoren erfolgt also ebenfalls modellgetrieben nach der hier beschriebenen Methode. Diese lässt sich jedoch nicht für die Implementierung der Beispielanwendung verallgemeinern.

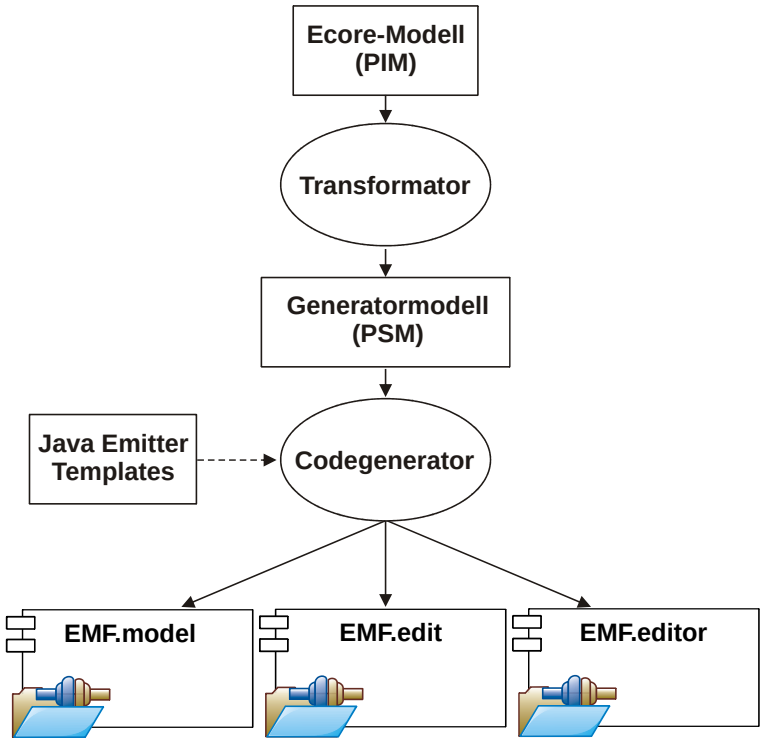


Abb. 4.7 Ablauf der Codeerzeugung

4.3.1 Das Generatormodell

Im Sinne der MDA nimmt ein Ecore-Modell die Rolle des PIM ein und besitzt demnach nicht genügend Information, um direkt Javacode zu erzeugen. So muss der Generator beispielsweise wissen, in welchen Java-Paketen die generierten Klassen abgelegt werden sollen oder welches Attribut einer Klasse als dessen Kennzeichen in Editoren fungieren soll.

Als Eingabe für den Codegenerator besitzt EMF ein eigenes Modell, das, mit Standardwerten initialisiert, beim Erstellen oder Einlesen eines Ecore-Modells miterzeugt wird. Der Transformationsschritt vom Ecore- zum Generatormodell ist fest in EMF eingebettet und bleibt dem Nutzer verborgen. Das erzeugte Generatormodell dient als PSM und stellt jeder Ecore-Metaklasse eine eigene zur Seite, wie Abb. 4.8 veranschaulicht.

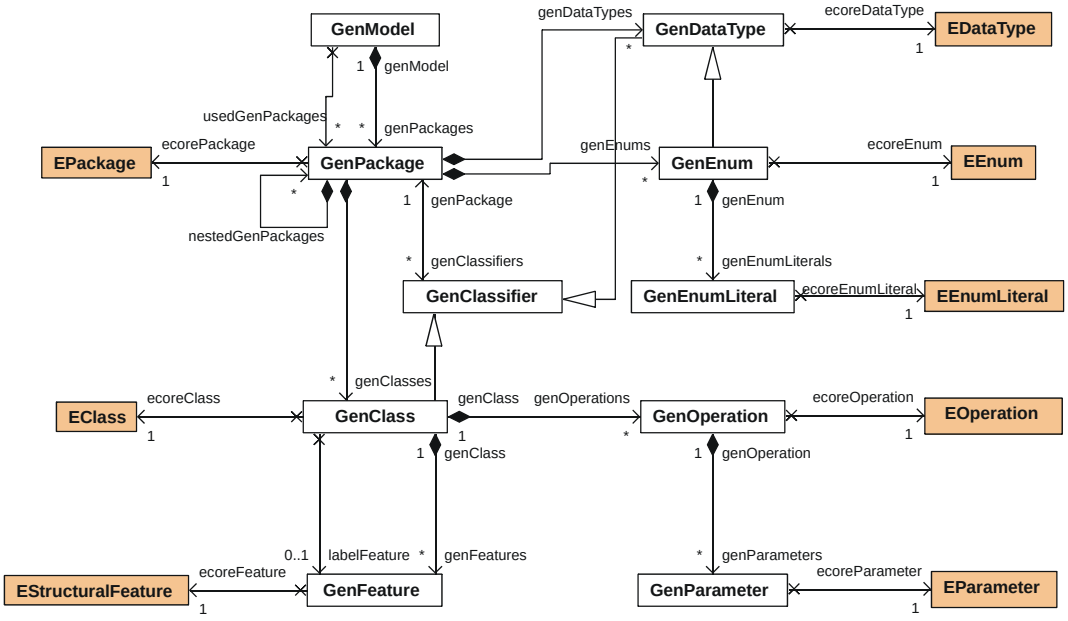


Abb. 4.8 Das Generatormodell und dessen Bezug zu Ecore (nach [BaGr05])

Abbildung 4.9 zeigt das Generatormodell mit allen Attributen, welche die Codeerzeugung steuern. Wie anhand der Datentypen zu sehen ist, wurde es selbst mit EMF modelliert und besitzt somit Ecore als Metamodell. Dies wurde wie schon beim Ecore-Editor dazu genutzt, einen Editor für das Generatormodell zu erzeugen und mit EMF auszuliefern.

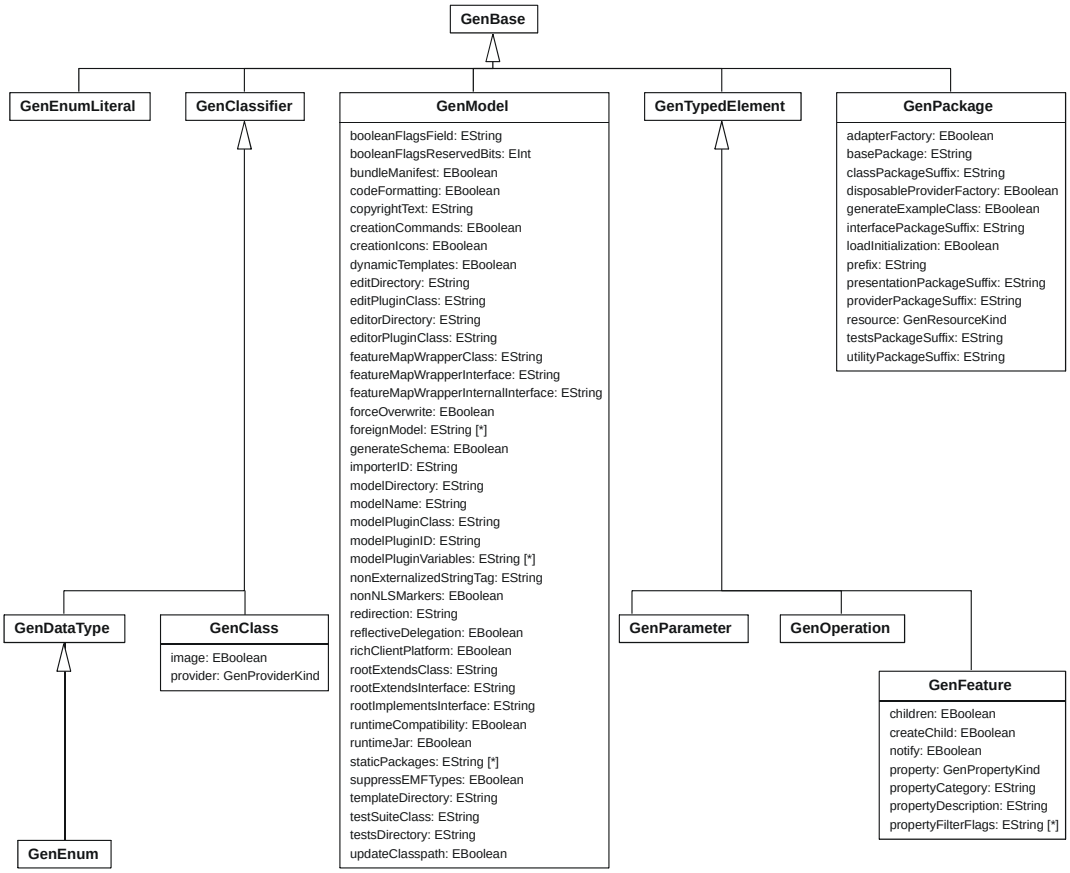


Abb. 4.9 Das Generatormodell mit Attributen

4.3.2 Java Emitter Templates (JET)

Java Emitter Templates sind Teil des EMF und dienen der Erzeugung von Code. Anders als der Name vermuten lässt, geben sie beliebigen Text aus und sind somit nicht auf Javacode beschränkt, unterstützen dessen Erzeugung aber auch nicht in besonderem Maße. JET leiht Syntax, Konzepte und sogar Quellcode von JSP und integriert diese in Eclipse ([BaGr05], S. 81). Zwar wurde JET für den Einsatz im EMF entwickelt, es lässt sich jedoch auch losgelöst davon verwenden.

Abbildung 4.10 verfeinert den Ablauf bei der Codeerzeugung aus Abb. 4.7 und skizziert die Arbeitsweise von JET.

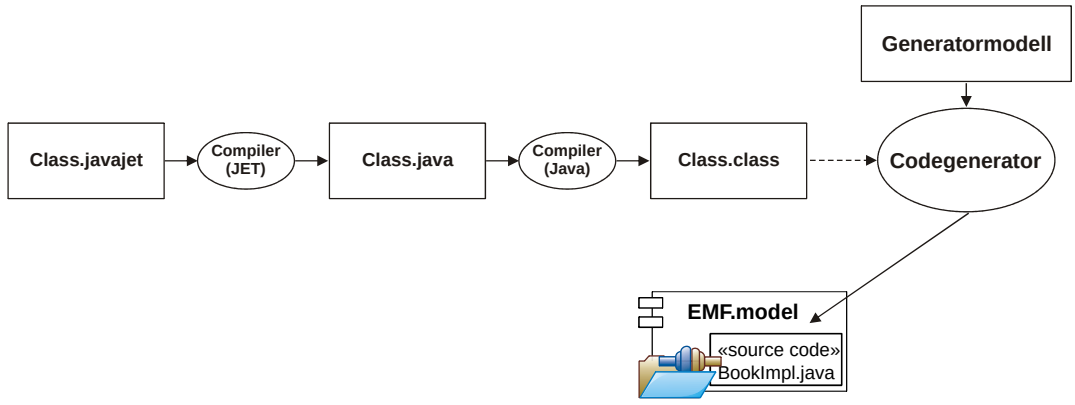


Abb. 4.10 Codegenerierung mit JET

In Abb. 4.12a-c werden die einzelnen Schritte anhand eines Beispiels konkretisiert. Für jede der in einer JSP-ähnlichen Syntax formulierten Vorlagen wird der Quellcode einer entsprechenden Java-Klasse erzeugt und dieser vom Java-Compiler übersetzt. Der Codegenerator übergibt dann dieser Klasse über die Methode

```
public String generate(Object argument)
```

das Generatormodell und erhält den erzeugten Code als String zurück. Es existiert für jedes zu generierende Plug-In-Projekt jeweils eine Menge von Java Emitter Templates[25]. Abbildung 4.11 zeigt die Syntaxelemente eines Java Emitter Template. Alle, bis auf die `Include`-Anweisung, sind in Abb. 4.12a vorhanden.

JET-Anweisung	`<%@ jet package="packageName" class="className" imports="spaceSeparatedImports" […] %>`	Bestimmt hauptsächlich Klassenname und Paket für die JET-Kompilierung
Include-Anweisung	`<%@ include file="urlSpec" %>`	Fügt Text aus angegebener Datei an Stelle der Include-Anweisung ein
Scriptlet	`<% javaCode %>`	Schließt beliebigen Javacode ein und steuert so die Ausgabe
Ausdruck	`<%= javaExpression %>`	Gibt das Ergebnis der Auswertung aus (Ergebnistyp muss zu einer StringBuffer.append() Methode passen)

Abb. 4.11 JET Syntaxelemente

[25] z. B. für EMF.model zu finden unter:
http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.emf/plugins/org.eclipse.emf.codegen.ecore/templates/model/?cvsroot=Tools_Project

```
<%@ jet package="org.eclipse.emf.codegen.ecore.templates.model"
imports="java.util.* org.eclipse.emf.codegen.ecore.genmodel.*"
class="Class" %>

<%GenClass genClass = (GenClass)argument;%>
[…]
public<%if (genClass.isAbstract()) {%> abstract<%}%> class
<%=genClass.getClassName()%><%=genClass.getClassExtends()%><%=genClass.getClassImple
ments()%>
{
[…]
} //<%=genClass.getClassName()%>
```

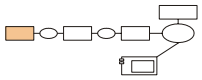


Abb. 4.12a Java Emitter Template zur Erzeugung einer Java-Klasse (Ausschnitt aus Class.javajet[26])

```
public class Class {
[…]
  protected final String TEXT_14 = NL + NL + "public";
  protected final String TEXT_15 = " abstract";
  protected final String TEXT_16 = " class ";
  protected final String TEXT_17 = NL + "{";
[…]
  protected final String TEXT_826 = NL + "} //";
[…]

  public String generate(Object argument)
  {
    StringBuffer stringBuffer = new StringBuffer();
    GenClass genClass = (GenClass)argument;
[…]
    stringBuffer.append(TEXT_14);
    if (genClass.isAbstract()) {
      stringBuffer.append(TEXT_15);
    }
    stringBuffer.append(TEXT_16);
    stringBuffer.append(genClass.getClassName());
    stringBuffer.append(genClass.getClassExtends());
    stringBuffer.append(genClass.getClassImplements());
    stringBuffer.append(TEXT_17);
[…]
    stringBuffer.append(TEXT_826);
    stringBuffer.append(genClass.getClassName());
[…]
    return stringBuffer.toString();
  }
}
```

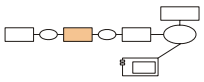


Abb. 4.12b Java Emitter Template aus Abb. 4.11a nach der JET-Kompilierung (Ausschnitt aus Class.java[27])

```
public class BookImpl implements Book
{
[…]
} //BookImpl
```

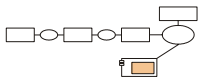


Abb. 4.12c Die generierte Java-Klasse (Ausschnitt aus BookImpl.java)

[26] http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.emf/plugins/org.eclipse.emf.codegen.ecore/templates/model/Class.javajet?rev=1.21&cvsroot=Tools_Project&content-type=text/vnd.viewcvs-markup

[27] http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.emf/plugins/org.eclipse.emf.codegen.ecore/src/org/eclipse/emf/codegen/ecore/templates/model/Class.java?rev=1.21&cvsroot=Tools_Project&content-type=text/vnd.viewcvs-markup

4.3.3 Die generierten Plug-In-Projekte

Ergebnisse der JET-Transformationen sind verschiedene Plug-In-Projekte, die aufeinander aufbauen.

Das Fundament bildet **EMF.model**. Dies ist die Java-Implementierung des mit Ecore definierten Modells und basiert seinerseits auf dem Laufzeitrahmenwerk von EMF. Für jede Modellklasse werden eine Schnittstelle mit Zugriffsmethoden für die Attribute nach JavaBeans-Konvention und eine implementierende Klasse generiert. Die Implementierung überwacht die im Modell beschriebenen Einschränkungen (z. B. Kardinalitäten oder bidirektionale Beziehungen bei `EReference`) und klinkt sich als Sender in den Benachrichtigungsmechanismus von EMF ein, um ggf. angemeldete Beobachter über Modelländerung zu informieren (Abb. 4.13).

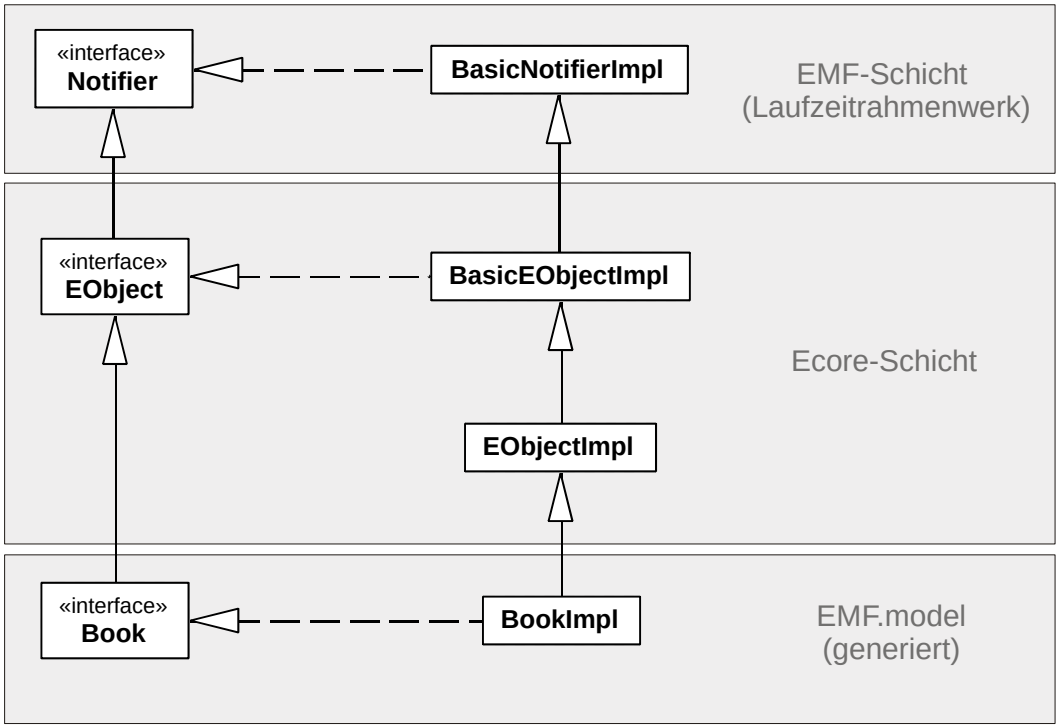


Abb. 4.13 Modellimplementierung (nach [BaGr05], S. 91)

Zusätzlich werden sowohl eine Fabrik zum Erstellen von Modellobjekten zur Laufzeit als auch eine weitere Klasse, die Metadaten des Modells wie Anzahl der Attribute jeder Modellklasse bereitstellt, erzeugt.

Neben der Manipulation der Modellobjekte über den generierten Code ermöglicht die Schnittstelle `EObject` Veränderungen auch mit Hilfe einer generischen API (Abb. 4.14).

```
public interface EObject {
    Object eGet(EStructuralFeature feature);
    void eSet(EStructuralFeature feature, Object newValue);
    boolean eIsSet(EStructuralFeature feature);
    void eUnset(EStructuralFeature feature);
}
```

Abb. 4.14 Generische Schnittstelle von EObject

EMF.edit und **EMF.editor** erweitern EMF.model um Konzepte zur Bearbeitung der Modelle in Editoren. EMF.edit beinhaltet den oberflächenunabhängigen Teil, während EMF.editor darauf aufbauend einen Modelleditor in die Oberfläche von Eclipse integriert.

4.4 Erweiterung des generierten Codes

Der Generator annotiert jedes generierte Codefragment (Klassen, Attribute, Methoden, etc.) mit `@generated`. Da mit Ecore kein Verhalten, sondern lediglich Struktur modelliert werden kann, müssen die generierten Klassen ggf. erweitert werden. Ecore besitzt zwar das Element `EOperation`, es dient aber nur der Deklaration von Methoden in den Schnittstellen. In der Implementierung wird eine `UnimplementedException` geworfen. Um eine Implementierung zu ersetzen und vor dem Überschreiben durch den Generator zu schützen, muss die entsprechende Annotation in `@generated not` umgewandelt werden.

Für die Umsetzung des Fallbeispiels wurde auf die manuelle Anreicherung der Metamodellklassen um Verhaltensaspekte verzichtet. Diese sind vollständig in den Modelltransformationen gekapselt.

5 Architektur des Fallbeispiels

JSF erlaubt es, den beliebten, aus der Entwicklung von Benutzeroberflächen für Desktopanwendungen bekannten Model-View-Controller-Ansatz ([Reen79]) auch für Webanwendungen zu nutzen. Auch beim Entwurf des Fallbeispiels kommt er zum Einsatz: Für jeden Anwendungsfall, z. B. beim Benutzerlogin, nimmt ein Controller die über JSF eintreffende Anfrage entgegen. Ebenfalls befüllt JSF ein Model-Objekt, das als Ausgangspunkt für den Zugriff auf das Model dient und sämtliche Objekte referenziert, die die zur Verarbeitung benötigten Daten (aus der Anfrage) aufnehmen. Der Controller leitet das Model-Objekt an die passende Methode eines Service-Objekts weiter, in der dann die eigentliche Verarbeitung (ggf. mit Zugriff auf die Datenbank mittels JPA) stattfindet. Zusätzlich zum Model-Objekt erhält die Service-Methode ein Objekt, das ihr erlaubt, Fehlermeldungen and den Aufrufer zurückzusenden oder die Nutzersitzung zu beenden.

In Abb. 5.1 zeigt ein Sequenzdiagramm die Interaktionen der verschiedenen Architekturelemente für den Login-Vorgang. Der Lebenszyklus aller dort enthaltenen Objekte, `callerContext` ausgenommen, wird von JSF gesteuert. Der `callerContext` ist das zuvor erwähnte Hilfsobjekt zur Rückgabe von Fehlermeldungen etc. an den Aufrufer. Es kapselt lediglich vorhandene JSF-Funktionalität über eine unabhängige Schnittstelle, damit die Service-Klassen keinen JSF-spezifischen Code enthalten müssen.

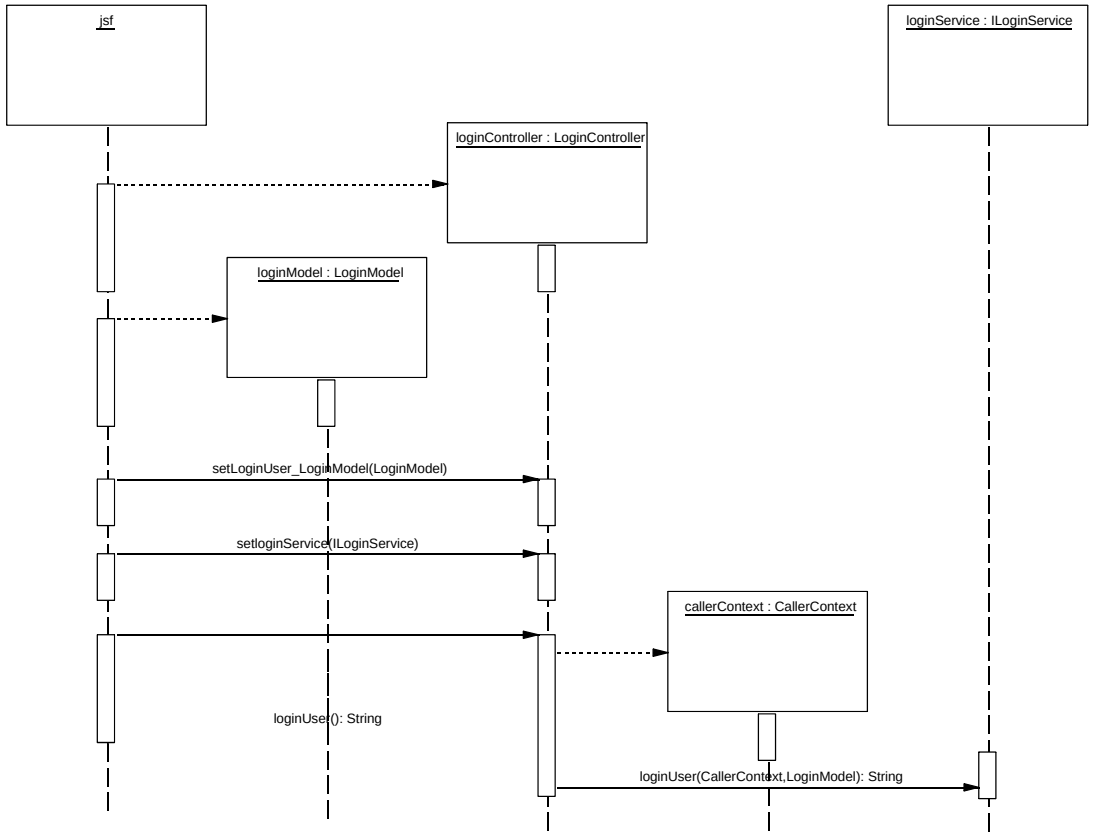


Abb. 5.1 Sequenzdiagramm zur Anfrageverarbeitung im Fallbesipiel

Für jede Anfrage erzeugt JSF zunächst neue Model- und Controller-Objekte und verknüpft sie miteinander, sodass der Controller eine Referenz auf das Model besitzt. Die für die Verarbeitung der Anfrage benötigten Objekte werden ebenso mit dem Model-Objekt (beim Login gemäß Abb. 5.2) verknüpft. Danach ruft JSF die für die Anfrage zuständige Methode im Controller auf. Diese delegiert den Aufruf an das passende Service-Objekt und übergibt dabei Model und neu erzeugten `callerContext`. Die Service-Objekte sind zustandslos, weshalb nur jeweils eine Instanz für die Lebensdauer der Anwendung existiert (Singleton, [GoF95]).

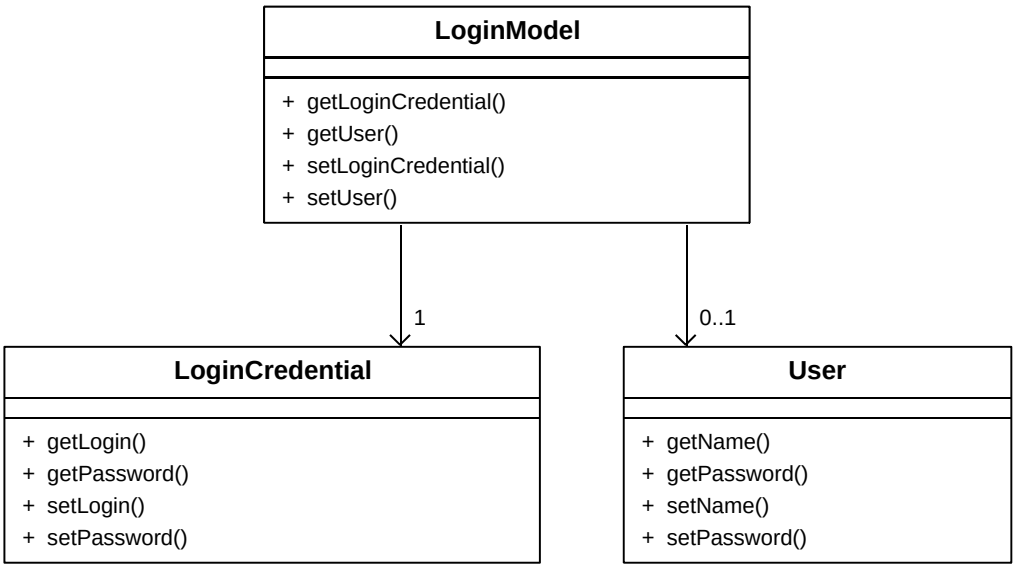


Abb. 5.2 Diagramm der für das Login benötigten Klassen

Die mit dem Model verknüpften Objekte haben unterschiedliche, von JSF kontrollierte Lebensdauern. So existiert z. B. das `LoginCredential`-Objekt, das Benutzername und Passwort aus dem Login-Formular aufnimmt, nur für die Dauer der Anfrage. Das `User`-Objekt bleibt jedoch bis zum Ende der Nutzersitzung bestehen.

Model-, Controller- und die mit dem Model verbundenen Klassen, wie die Entity Bean `User`, werden aus Modellen generiert. Das gleiche gilt für JSF-Konfigurationsdateien, die diese Klassen verdrahten und deren Lebenszyklen steuern. Die Service-Klassen als Kern der Anwendung müssen manuell erstellt werden, ebenso die Benutzeroberfläche.

Die Wahl der Modelltransformationssprache fiel wegen der Analysen in [Falk05], der Integration mit Eclipse und der Unterstützung für EMF auf die Atlas Transformation Language (ATL), deren Funktionsweise und Einsatz anhand der Generierung von Entity Beans im nächsten Kapitel beschrieben wird.

6 Generierung von Entity Beans mit ATL

Die Java Platform, Enterprise Edition (Java EE) sieht für die Speicherung von Objekten in relationalen Datenbanken Entity Beans vor. Die in der kommenden Version 5 der Java EE enthaltene Spezifikation der Enterprise JavaBeans (EJB) 3.0 vereinfacht den Umgang mit dem Persistenzaspekt und unterstützt erstmals Vererbung ([Sun06b], S. 9). Da Entity Beans nicht mehr vom EJB-Container verwaltet werden, kann sie der Entwickler wie gewöhnliche Java-Klassen behandeln.

Neben den bereits früher verwendeten XML-basierten Deployment-Deskriptoren können alternativ Java-Annotationen die nötigen Zusatzinformationen für die Abbildung der Klassen auf Datenbanktabellen aufnehmen ([Sun06b], S. 14). Dies hat den Vorteil, dass keine getrennten Artefakte mehr für die Definition einer Entity Bean verwaltet werden müssen, die deklarative Beschreibung der Zusatzinformationen aber gewahrt bleibt.

Die im Rahmen der Fallstudie entwickelten ATL-Transformationen nutzen diese Neuerungen und erzeugen aus einer Ecore-Beschreibung des Klassenmodells sowie einem Modell mit Persistenzinformationen ein vereinigtes Java-Modell, das die Entity Beans vollständig beschreibt. Aus dem so entstandenen Modell erzeugt eine letzte Transformation schließlich Javacode (Abb. 6.1).

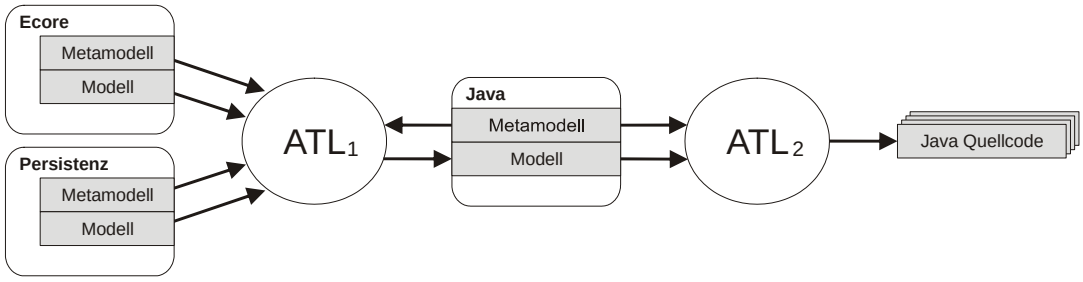


Abb. 6.1 Übersicht der Transformationen zur Erzeugung von Entity Beans

Abbildung 6.2 zeigt einführend Beispiele für Ausgangspunkt und Ziel dieser Transformationskette. Die im Fallbeispiel genutzte Java-Klasse `Miniliga` wird aus Ecore- und Persistenzmodell generiert.

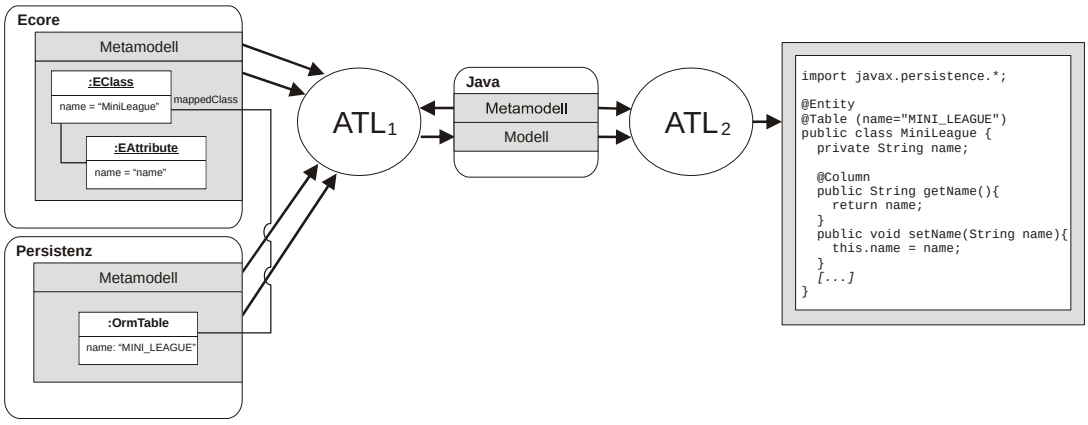


Abb. 6.2 Einführendes Beispiel zur Generierung von Entity Beans

Die Modelltransformationssprache ATL ist als Eclipse-Plugin implementiert und arbeitet direkt auf den mit EMF erzeugten XMI-Dateien der Modelle und Metamodelle. Eine Codegenerierung der Metamodell- und Modellklassen durch EMF ist nicht notwendig, da ATL diese selbst dynamisch aus den entsprechenden XMI-Dateien erzeugt. Als Meta-Metamodell dient ausschließlich Ecore (vgl. Kapitel 4).

Nach einer Einführung in das neue Java-Sprachkonzept der Annotationen folgt ein Abschnitt über die grundlegenden Möglichkeiten, Objekte und deren Beziehungen auf Datenbanktabellen abzubilden. Die Umsetzung dieser Möglichkeiten durch die Java Persistence API (JPA) der EJB-Spzeifikation 3.0 wird in diesem Zuge ebenfalls betrachtet. Nachfolgend stehen die in der Transformationskette genutzten Metamodelle im Mittelpunkt, bevor die Funktionsweise von ATL und die Transformationsschritte zur Erzeugung der Entity Beans beleuchtet werden.

6.1 Java-Annotationen

Neuer Teil des Java Sprachstandards in der Version 1.5 sind Annotationen, die Klassen, Schnittstellen, Attributen und Methoden anheim gestellt werden können. Annotationen enthalten durch Annotationstypen (syntaktisch) definierte Zusatzinformationen („Metadaten“), wie im Falle von Entity Beans Persistenzinformationen. Abbildung 6.3a und 6.3b zeigen eine annotierte Methode und den zugehörigen Annotationstyp ([Sun06c], S. 163). Die Annotation gibt die Eigenschaften der Datenbanktabellenspalte an, die den Wert des Attributs `isbn` speichert.

```
public class Book {

  @Column(name="ISBN", unique=true, length=13)
  private String isbn;
 […]
}
```

Abb. 6.3a Beispiel eines annotierten Attributs

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Column {
  String name() default "";
  boolean unique() default false;
  boolean nullable() default true;
  boolean insertable() default true;
  boolean updatable() default true;
  String columnDefinition() default "";      // SQL fragment for DDL generation
  String table() default "";                 // columns may be stored in a
                                             // different table
  int length() default 255;                  // String length
  int precision() default 0;                 // decimal precision
  int scale() default 0;                     // decimal scale
}
```

Abb. 6.3b Annotationstyp zur Annotation aus Abb. 3a

Annotationstypen werden ähnlich wie Schnittstellen deklariert, jedoch können zu den angegebenen Annotationselementen zusätzlich Vorgabewerte definiert werden. Die Typen von Annotationselementen sind beschränkt auf primitive Datentypen, `String`, `Class`, `enum`, Annotationstypen und Arrays dieser Typen. Wie in Abb. 6.3b zu sehen ist, können Annotationstypen ebenfalls annotiert werden.

Bei der Definition von Annotationstypen gibt die vordefinierte Annotation `@Target` an, auf welche Java-Elemente Annotationen dieses Typs anwendbar sind[28]. Die Annotation `@Retention` bestimmt den Gültigkeitsbereich der Annotationen des betreffenden Typs. `SOURCE` bedeutet, dass der Compiler die Annotationen dieses Typs nicht in den von ihm erzeugten Bytecode aufnehmen muss. Enthält `@Retention`[29] den Wert `CLASS`, so muss der Compiler sie einbinden, jedoch braucht die virtuelle Maschine den Zugriff auf die Annotationen zur Laufzeit nicht zu unterstützen. Bei `RUNTIME` muss auch letzteres möglich sein.

[28] siehe auch http://java.sun.com/j2se/1.5.0/docs/api/java/lang/annotation/ElementType.html

[29] siehe auch http://java.sun.com/j2se/1.5.0/docs/api/java/lang/annotation/RetentionPolicy.html

6.2 Abbildung von Klassen auf Datenbanktabellen

Die EJB-Spezifikation [Sun06c] erlaubt weitgehende Kontrolle über die Abbildung von Klassen auf Tabellen einer Datenbank. Dabei gilt es festzulegen, wie Objektidentität, Attribute, Vererbungshierarchien und Beziehungen zu behandeln sind.

Zur Abbildung i. w. S. gehört auch die Lösung des Problems der Konflikterkennung bei Objektspeicherung, sobald eine optimistische Sperrstrategie eingesetzt wird.

Die Annotation `@Entity` markiert eine gewöhnliche Klasse als Entity Bean (Abb. 6.4), d. h., der Persistence Provider erkennt daran, dass die Objekte dieser Klasse dauerhaft in einer relationalen Datenbank gespeichert werden sollen. Der Persistence Provider stellt die Laufzeitumgebung bereit, die den Persistenzaspekt umsetzt. Vor Version 3.0 der EJB-Spezifikation war dies im EJB-Container integriert. Die Auslagerung ermöglicht nun die Verwendung von Entity Beans auch außerhalb der Java EE.

```
@Entity
public class Book {
 […]
}
```

Abb. 6.4 Definition einer Entity Bean

6.2.1 Objektidentität

Da das relationale Modell keine Identität der Tupel unabhängig von deren Werten kennt, muss die Speicherung des Objektzustands auch einen Objektidentifikator als Primärschlüssel der Tabelle umfassen. Entity Beans besitzen also stets ein mit `@Id` annotiertes Attribut, das diesen Identifikator aufnimmt (Abb. 6.5).

```
@Entity
public class Book {

  @Id
  private Long id;
 […]
}
```

Abb. 6.5 Die Objekt-ID

6.2.2 Attribute

Meist wird eine Klasse auf genau eine Tabelle abgebildet und jedes Attribut der Klasse entspricht genau einer Spalte dieser Tabelle. Jedoch lassen sich Attribute auch von der Speicherung ausschließen (transiente Attribute), erscheinen also nicht in einer Tabelle. Ebenso können Attribute auf Spalten verschiedener Tabellen verteilt werden, oder ein Attribut nimmt mehrere Spalten ein. All diese Varianten unterstützt [Sun06c] mit entsprechenden Annotationen.

Im einfachsten Fall, ein Attribut pro Spalte, ist oft keine Annotation zur Speicherung nötig, wenn einer der folgenden Attributtypen vorliegt ([Sun06c], S. 20f):

- primitiver Datentyp (oder entsprechende Objekthülle)
- `java.lang.String`
- `java.math.BigDecimal`
- `java.math.BigInteger`
- `java.util.Date`
- `java.util.Calendar`
- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`
- `byte[]` (oder `Byte[]`)
- `char[]` (oder `Character[]`)
- `enum`
- jeder die Schnittstelle `Serializable` implementierende Typ

Da diese Typen eine Entsprechung in SQL (Structured Query Language) besitzen bzw. kanonisch abgebildet werden können, bedarf es keiner weiteren Persistenzinformation. Der Spaltenname ist dann mit dem Attributnamen identisch. Für weitergehende Einstellungen dient die bereits aus Abb. 6.3a und 6.3b bekannte `@Column`-Annotation.

Die Annotation `@Transient` veranlasst den Persistence Provider das betreffende Attribut zu ignorieren. In Abb. 6.6 wird die Gesamtsumme eines Warenkorbs nicht in der Datenbank abgelegt, da sie nach späterem Laden stets neu über den (gespeicherten) Warenkorbinhalt errechnet werden kann.

```
@Entity
public class ShoppingCart {

  @Transient
  private java.math.BigDecimal total;
 […]
}
```

Abb. 6.6 Ein nicht-persistentes Attribut

Durch die Angabe mehrerer Tabellen zu einer Klasse lassen sich Attribute beliebig verteilen. In Abb. 6.7 wird das Gehalt eines Angestellten in einer separaten Tabelle untergebracht. Den Namen der Haupttabelle legt die `@Table`-Annotation fest, die generell entfallen kann, sofern der Tabellenname mit dem Klassennamen übereinstimmen soll. Die Annotation `@SecondaryTable` ([Sun06c], S. 164) gibt zusätzliche Tabellen für die Aufnahme von Attributen einer Klasse an. Das Element `table` der Annotation `@Column` (vgl. Abb. 6.3b) gibt für jedes Attribut die Tabelle an, in der es abgelegt wird. In der Datenbank enthält die Zusatztabelle außerdem eine Spalte mit dem Fremdschlüssel zur Haupttabelle, deren Name der `@Id`-Spalte der Haupttabelle entspricht.

```
@Entity
@Table(name="EMPL")
@SecondaryTable(name="EMP_SALARY")
public class Employee {

  @Id
  @Column(name="EMP_ID")
  private Long id;

  private String ssn;

  @Column(table="EMP_SALARY")
  private java.math.BigDecimal salary;
 […]
}
```

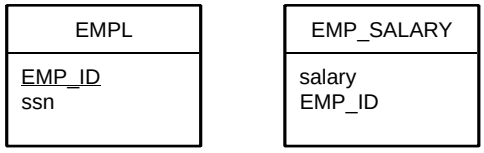


Abb. 6.7 Mehrere Tabellen pro Klasse

Falls der Typ eines Attributs keinem der oben aufgeführten Typen entspricht und selbst keine Entity Bean ist, muss er als `@Embeddable` ([Sun06c], S. 196) annotiert werden, damit eine Speicherung möglich ist. Voraussetzung dafür ist, dass dieser Typ aus Attributen mit den o. a. abbildbaren Typen zusammengesetzt ist (Abb. 6.8). Besitzt eine Entity Bean ein Attribut eines solchen Typs, so werden dessen Attribute in die Tabelle der Entity Bean eingebettet. Ein Attribut aus der Entity Bean wird also auf mehrere Tabellenspalten abgebildet.

```
@Embeddable
public class EmploymentPeriod {
  java.util.Date startDate;
  java.util.Date endDate;
 […]
}
```

Abb. 6.8 Ein Typ, der in eine Entity Bean eingebettet werden kann

6.2.3 Vererbung

Ein weiteres objektorientiertes Konzept, das dem relationalen Modell fehlt, ist die Vererbung. Abbildung 6.9 zeigt die möglichen Strategien zur Abbildung von Vererbungshierarchien. Ein Persistence Provider muss gemäß Spezifikation mindestens die Strategien „Eine Tabelle pro Klasse“ (`SINGLE_TABLE`) und „Eine Tabelle pro Klassenhierarchie“ (`JOINED`) unterstützen ([Sun06c], S. 39).

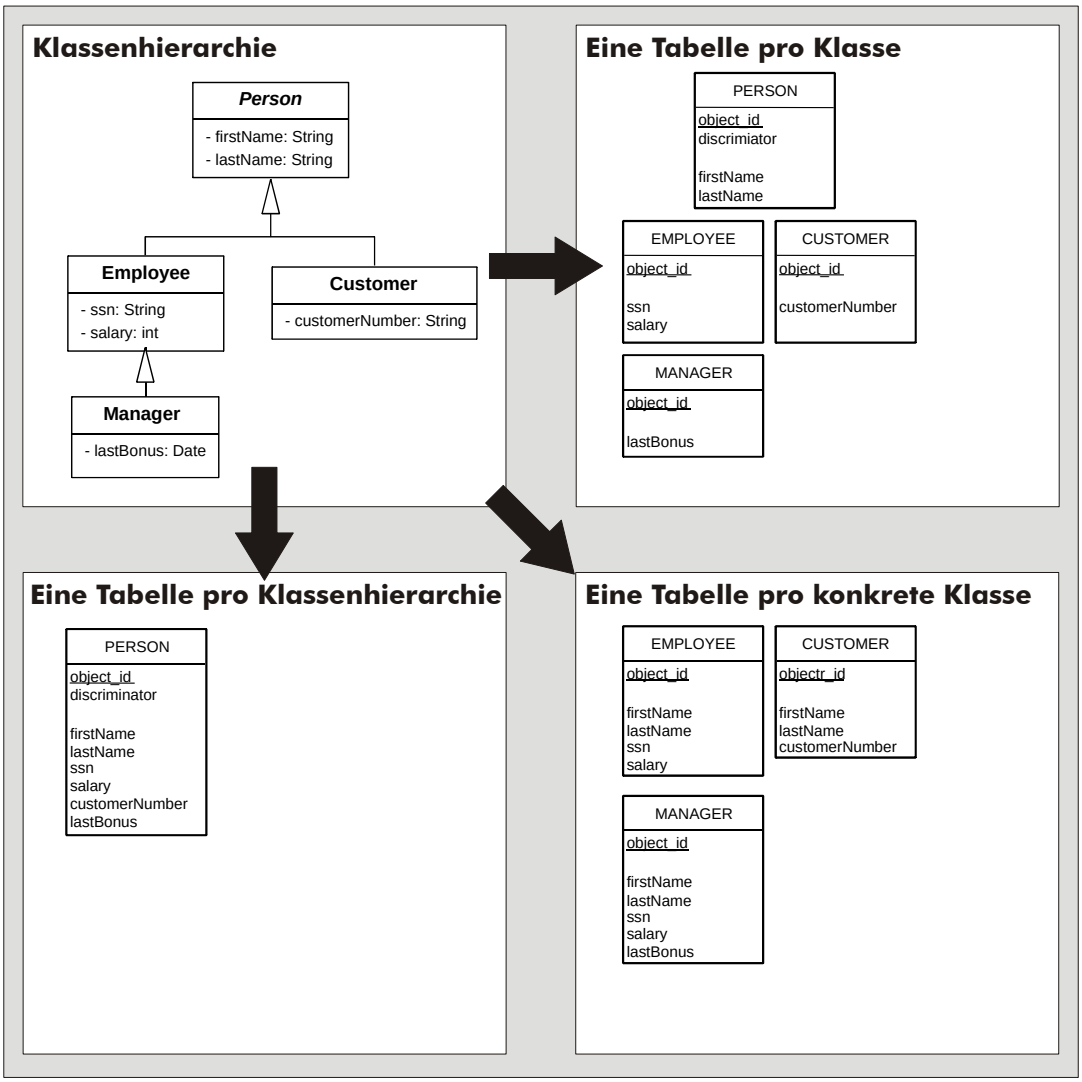


Abb. 6.9 Mögliche Strategien zur Abbildung von Vererbung

Wird **eine Tabelle pro Klassenhierarchie** genutzt (Voreinstellung bei Entity Beans), werden alle in der Klassenhierarchie zu speichernden Attribute in einer Tabelle abgelegt. Neben dem Primärschlüssel für die Objektidentität (s. o.) muss außerdem ein „Diskriminator“ gespeichert werden; ein Wert, der die konkrete Klasse des Objektes identifiziert. Dieser ist nötig, damit beim Laden aus der Datenbank die Klasse des zu erzeugenden Objekts ermittelt werden kann, da sie nicht aus dem Tabellennamen ableitbar ist. Der Diskriminator kann wie gewohnt mittels Annotationen definiert werden (Abb. 6.10), sofern von den Vorgaben abgewichen werden soll. Er wird lediglich in die Tabelle aufgenommen und bleibt auf Java-Seite unsichtbar.

```
@Entity
@Inheritance(strategy=SINGLE_TABLE) //could be omitted since default
@DiscriminatorColumn(name="DISCR", discriminatorType=DiscriminatorType.STRING)
public class Employee {
 […]
}
```

Abb. 6.10 Annotationen für Vererbung bei Entity Beans

Existiert **eine Tabelle pro Klasse** in der Datenbank, ist kein Diskriminator nötig, da die Klasse durch die Tabelle eindeutig bestimmt ist. Jede Tabelle enthält nur die Attribute der zugehörigen Entity Bean. Wird ein Objekt einer Subklasse geladen, muss ein Join über die Tabellen aller Superklassen erfolgen, um sämtliche Attribute zu erreichen. Der Wert für die Objektidentität einer Subklassentabelle dient dabei als Fremdschlüssel für die Superklassentabelle.

Als dritte Strategie besteht die Möglichkeit, **eine Tabelle pro konkrete Klasse** anzulegen. Jede Tabelle enthält so die Attribute der konkreten Klasse und die der Superklassen.

6.2.4 Beziehungen

Zur Speicherung von Objektbeziehungen in relationalen Datenbanken dienen Fremdschlüssel. Welche Tabellen die Fremdschlüssel beherbergen, ist von der Kardinalität der Beziehung abhängig. Abbildung 6.11 zeigt die Datenbanktabellen, die sich aus den Objektbeziehungen je nach Kardinalität ergeben. Im Falle von M:N wird eine zusätzliche Tabelle nötig, und die 1:1-Beziehung ist ein Spezialfall von 1:N, bei dem die Werte der Fremdschlüsselspalte eindeutig sein müssen (`UNIQUE`-Constraint).

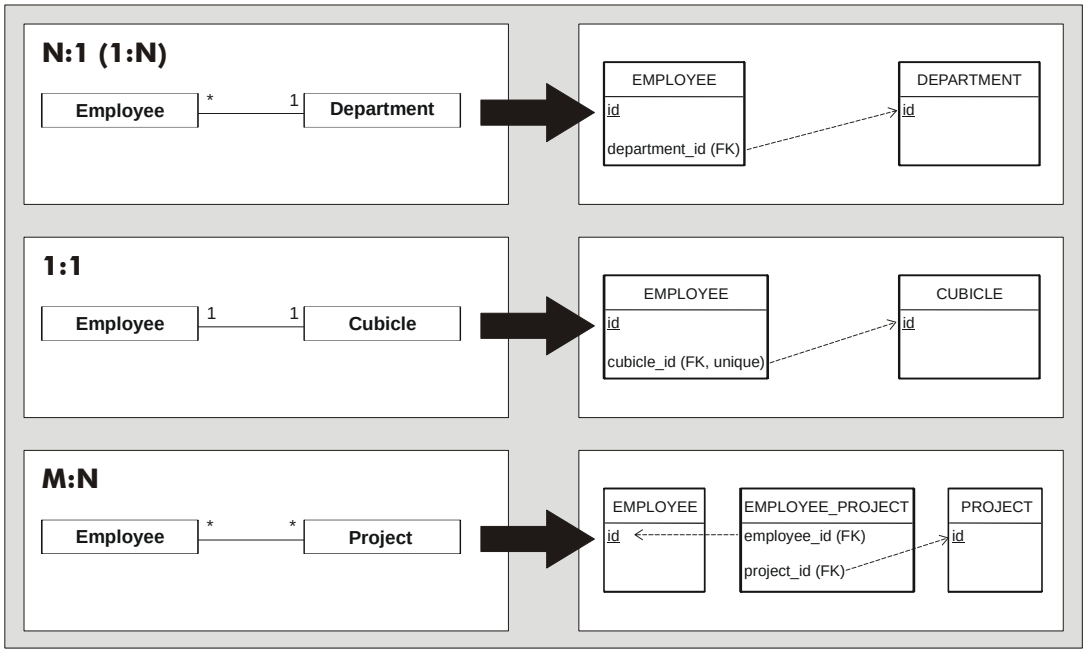


Abb. 6.11 Abbildung von Objektbeziehungen auf Datenbanktabellen

Die Kardinalität einer Beziehung reicht jedoch nicht aus, um eine eindeutige Abbildung auf Tabellen zu ermöglichen. Bei 1:1-Beziehungen muss feststehen, welche Tabelle die Fremdschlüsselspalte enthalten soll. Ebenfalls muss der Name der zusätzlichen Join-Tabelle im Falle von M:N eindeutig sein (`EMPLOYEE_PROJECT` oder `PROJECT_EMPLOYEE`). Dazu wird eine Seite der Beziehung als die Beziehung besitzende Seite ausgezeichnet. Im obigen Beispiel ist dies stets `Employee`.

Weitere Informationen sind nötig, um uni- und bidirektionale Beziehungen unterscheiden zu können. Die bei Entity Beans hierfür genutzten Annotationen sind beispielhaft in Abb. 6.12 dargestellt (vgl. [Sun06c], S. 25 ff).

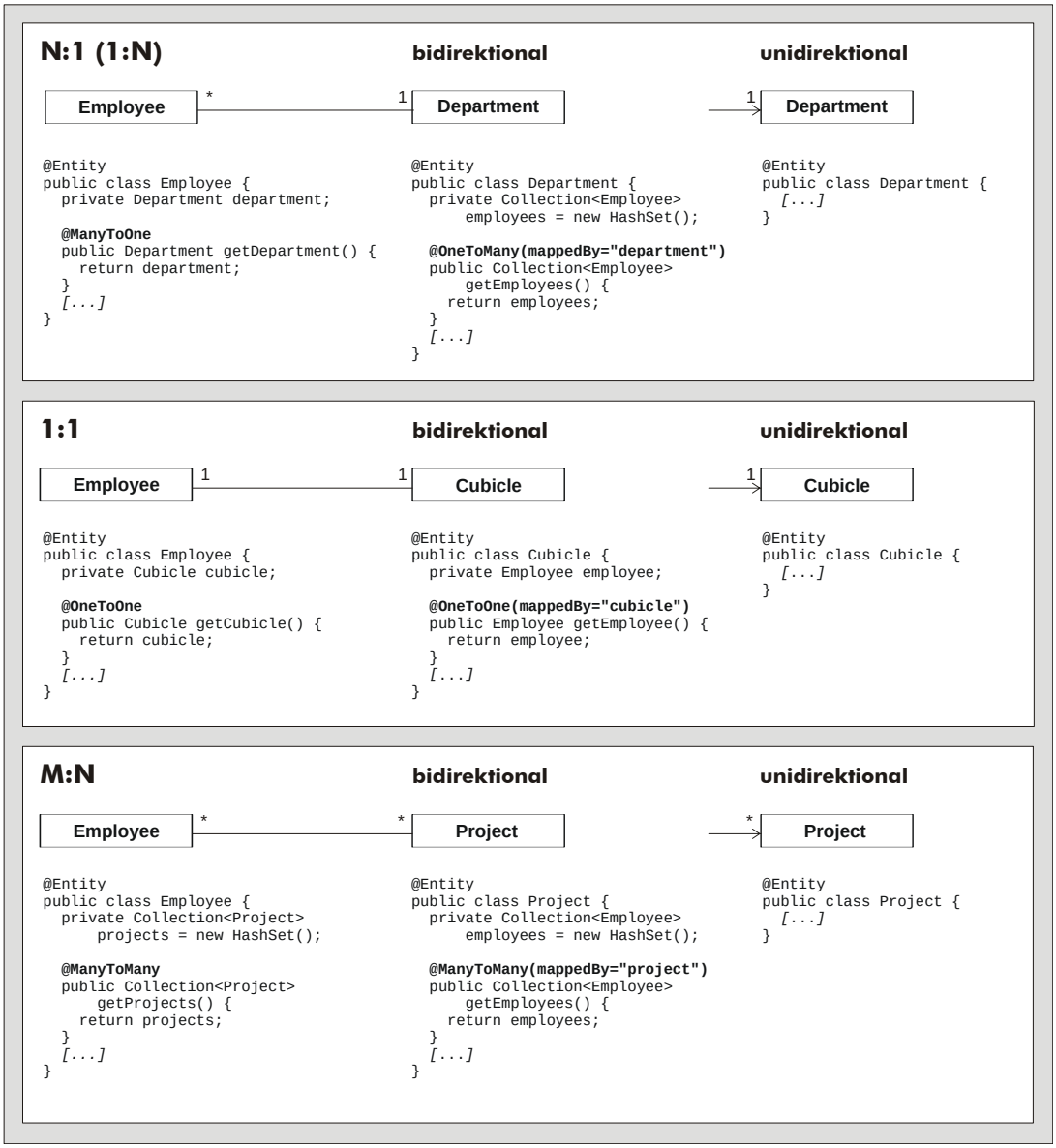


Abb. 6.12 Annotationen für Beziehungen zwischen Entity Beans

Bei bidirektionalen Beziehungen definiert das Annotationselement `mappedBy` auf der nicht-besitzenden Seite, welche Referenz das andere Ende der Beziehung auf der besitzenden Seite ausmacht.

6.2.5 Versionen

Vor allem bei Web-Anwendungen, einem der Haupteinsatzfelder der Java EE, verbietet sich eine pessimistische Sperrung von Entity Beans, sobald eine Transaktion mehrere HTTP-Anfragen überspannt. Da HTTP keinen Verbindungszustand kennt und zwischen zwei Anfragen viele Minuten vergehen können, wäre die Entity Bean sonst für die Verwendung in den zwischenzeitlich eintreffenden Anfragen anderer (tausender) Clients gesperrt.

Um dies zu vermeiden, wird bei derart langen Transaktionen die Isolation zugunsten der Skalierbarkeit geopfert. Um weiterhin Konsistenz gewährleisten zu können, müssen die nun möglichen Konflikte beim Speichern einer Entity Bean erkannt und aufgelöst werden.

Jede Entity Bean, bzw. deren Tabelle, enthält zu diesem Zweck ein zusätzliches numerisches Versionsattribut, das bei jeder Speicherung inkrementiert wird. Anhand der Version kann der Persistence-Provider nun entscheiden, ob zwischen Laden und Speichern ein zweiter Prozess dieselbe Entity Bean bereits verändert und gespeichert hat. In diesem Fall ist die Versionsnummer in der Datenbank höher als die des durch den ersten Prozess zu speichernden Objekts, und die Transaktion bricht ab. Abbildung 6.13 zeigt die Annotation zur Definition des Versionsattributs.

```
@Entity
public class Employee {

  @Version
  private int version;
 […]
}
```

Abb. 6.13 Versionsattribut zur Erkennung von Speicherungskonflikten

Zur Konflikterkennung unterstützt [Sun06c] auch den Einsatz von Zeitstempeln. Wegen der prinzipiellen Kollisionsmöglichkeiten und dem zusätzlichen Aufwand zur Synchronisation in Clusterumgebungen wird diese Variante als unterlegen angesehen und nicht weiter ausgeführt.

6.3 Metamodelle

Die Metamodelle der an den Transformationen beteiligten Modelle wurden, soweit nicht vorhanden, mit dem Eclipse Modeling Framework (EMF) erstellt. Zur Erstellung des Persistenzmodells wurde aus dem entsprechenden Metamodell mittels EMF ein Editor generiert (vgl. Kapitel 4.3). Für die Klassenbeschreibung dient Ecore als Metamodell.

6.3.1 Persistenz

Die Persistenzinformationen werden gemäß dem Metamodell aus Abb. 6.14 festgehalten. Für jede zu persistierende Klasse (`EClass`-Instanz) ist eine diese Klasse referenzierende `OrmTable`[30]-Instanz zu erstellen. Auf diese Weise kann die Transformation die `EClass`-Instanz als Entity Bean identifizieren und entsprechende Annotationen erstellen. Sofern keine `OrmColumn`-Instanzen für eine `OrmTable` angegeben werden, geht die Transformation davon aus, dass jedes Attribut der referenzierten Klasse mit den in der EJB-Spezifikation [Sun06c] aufgeführten Vorgaben abgebildet werden soll. Sollen andere Werte Anwendung finden, gibt eine `OrmColumn`-Instanz diese an und verweist auf das betreffende Attribut. Ein `TransientColumn` unterbindet die Persistierung eines Attributs.

Wie OrmColumn ermöglicht OrmRelationship das Überschreiben von Spezifikationsvorgaben zur Abbildung von Beziehungen. Alle nötigen Informationen lassen sich sonst bereits aus dem Ecore-Modell ableiten.

Der `FetchType` dient der Optimierung und bestimmt den Zeitpunkt, zu dem ein Attribut oder eine referenzierte Entity Bean aus der Datenbank geladen wird. Dabei zwingt der Standardwert `EAGER` den Persistence-Provider, das Attribut beim Laden und Erzeugen des Objekts mitzuladen; der Wert `LAZY` dient hingegen als Hinweis, dass das Attribut auch später (bei Zugriff) geladen werden kann. ([Sun06c], S. 179). Beispielsweise ließe sich so das ständige Laden eines in der Datenbank als Binary Large Object (BLOB) gespeicherten Fotos eines Person-Objekts verhindern, wenn es die Anwendung nur bei Bedarf anzeigen soll.

[30] ORM = Object Relational Mapping

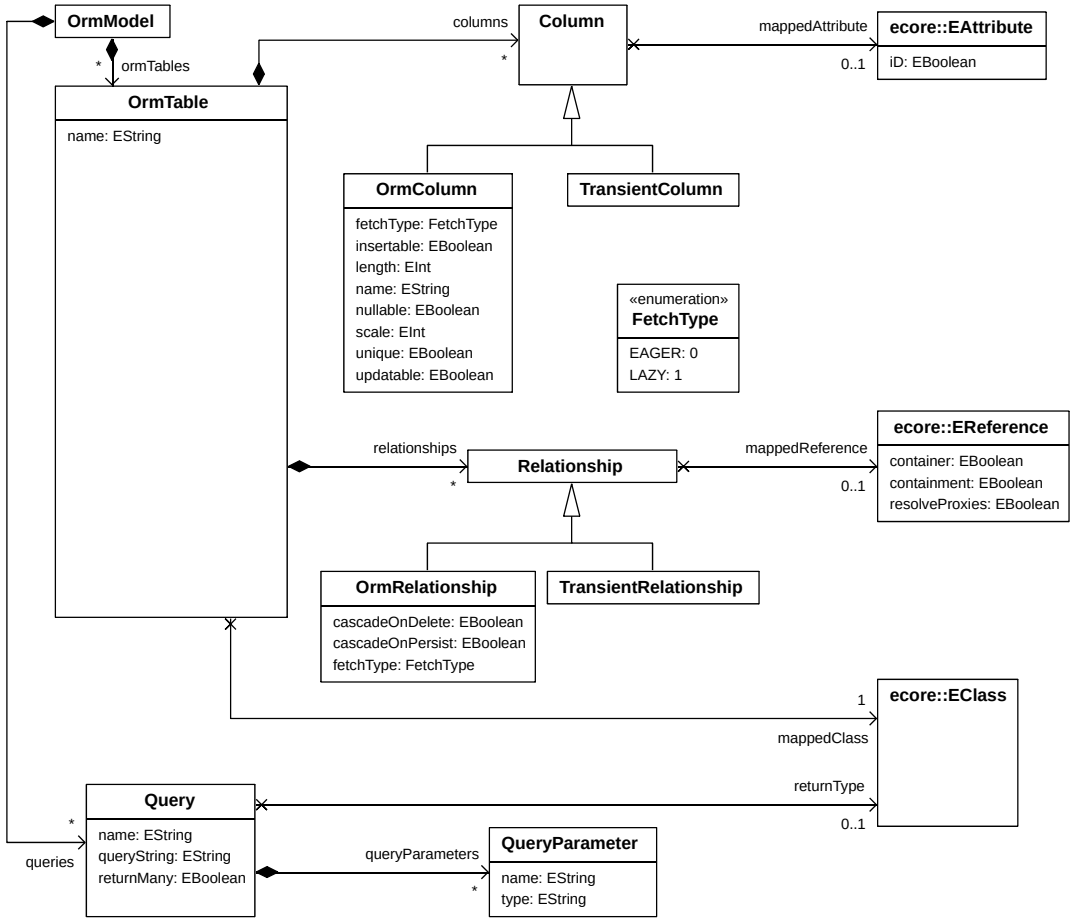


Abb. 6.14 Metamodell für Persistenzinformationen

6.3.2 Java

Das Java-Metmodell weist eine ähnliche Struktur wie Ecore auf, besitzt jedoch bis auf Datentypen keine Abhängigkeiten zu Ecore (Abb. 6.16). Der Hauptunterschied liegt in der Behandlung von Annotationen. Abbildung 6.15 zeigt den betreffenden Ausschnitt des Java-Metamodells. Alle annotierbaren Elemente, also `JavaClass` (schließt auch Annotationstypen ein), `Field` und `Method`, erben von `Annotatable`. Eine Annotation kann mehrere `AnnotationElements` enthalten, die weiter in `BasicAnnotationElement` und `AnnotationAnnotationElement` unterschieden werden.

Instanzen von `BasicAnnotationElement` besitzen als Typ entweder einen primitiven Datentyp, `String`, `Class`, `enum` oder ein Array eines dieser Typen. Eine Multiplizität größer 1 des Attributs `values` besagt, dass ein Array vorliegt.

Annotationselemente können als Typ jedoch auch Annotationstypen und Arrays davon besitzen, deren Werte sind also wieder Annotationen. Für diesen Fall existiert das `AnnotationAnnotationElement`, das kein Attribut `values`, sondern eine entsprechende Beziehung zu `Annotation` besitzt. Auch hier entscheidet die Multiplizität, ob ein Array vorliegt.

Die den Typ des `AnnotationElement` bestimmende Beziehung zu `JavaClassifier` fehlt aus Gründen der Übersichtlichkeit in Abb. 6.15.

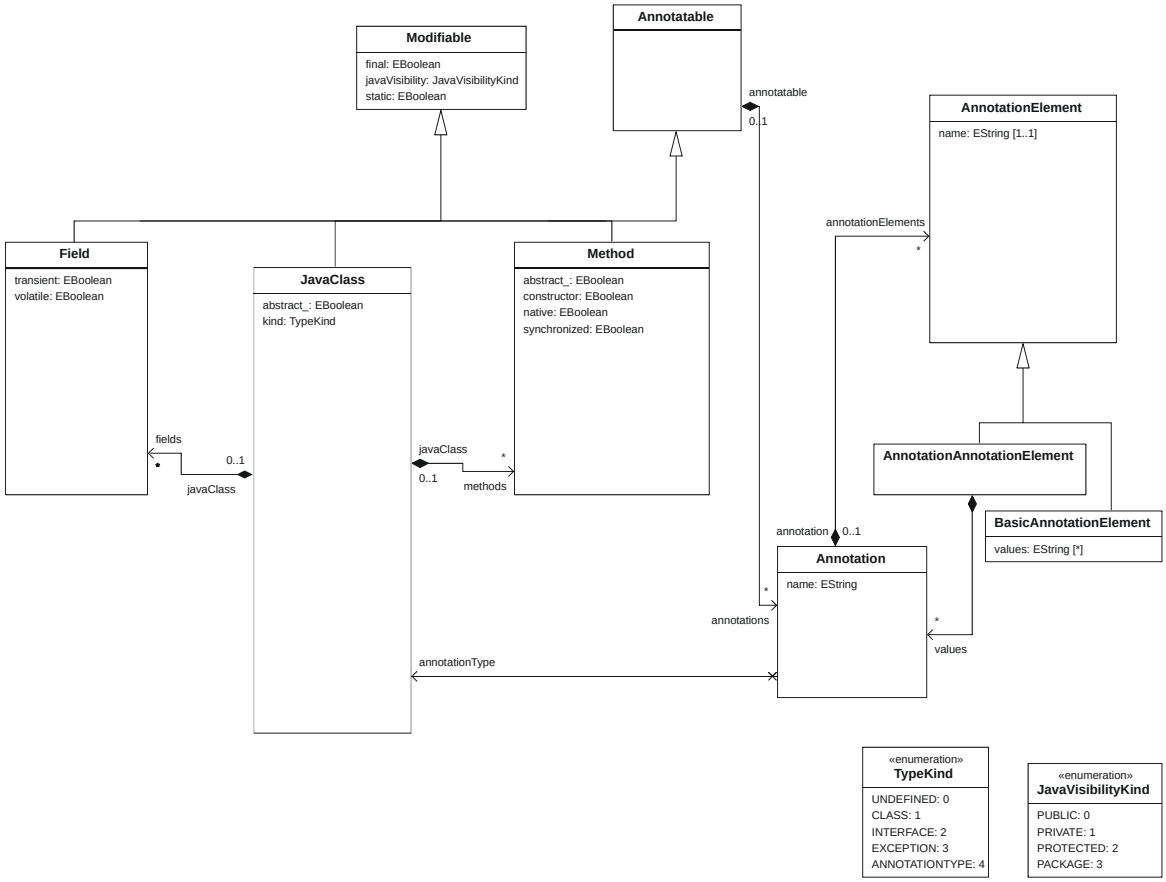


Abb. 6.15 Annotationen im Java-Metamodell (Ausschnitt)

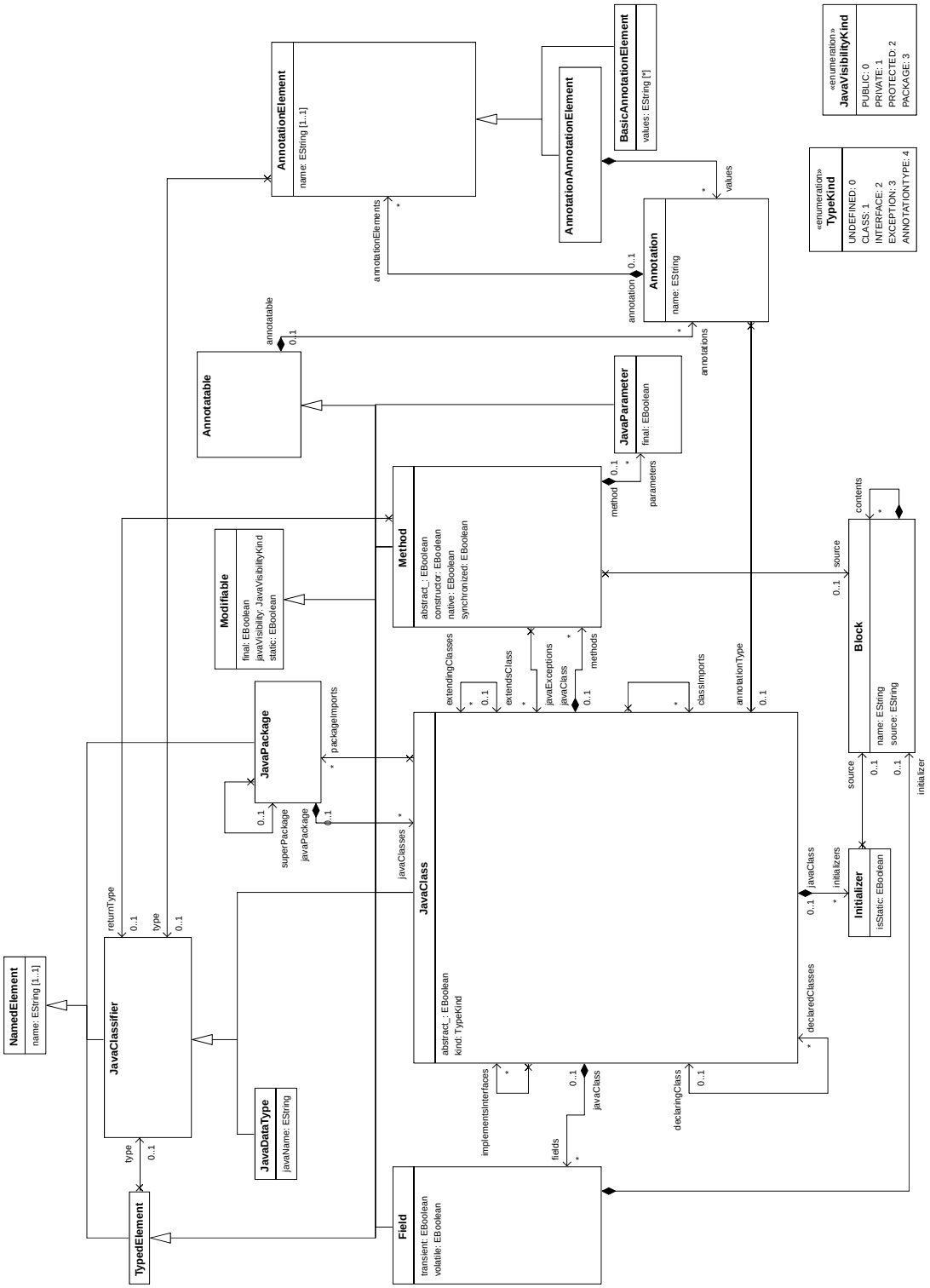


Abb. 6.16 Java-Metamodell

6.4 ATL-Transformationen

Die Erzeugung der Entity Beans erfolgt in zwei Schritten (vgl. Abb. 6.1). Zunächst werden Klassen- und Persistenzmodell von einer ersten ATL-Transformation in einem Java-Modell zusammengeführt. Im zweiten Schritt erzeugt eine weitere ATL-Transformation aus dem Java-Modell entsprechende Quellcode-Dateien.

Vor der Beschreibung der Transformationen zur Generierung der Entity Beans soll zunächst die grundsätzliche Funktionsweise von ATL anhand vereinfachter Modelle und Metamodelle herausgearbeitet werden.

6.4.1 Grundzüge einer ATL-Transformation

In Abb. 6.17 ist eine Modelltransformation dargestellt, die die grundsätzliche Anwendung von ATL zeigen soll. Beteiligt an der ATL-Transformation sind ein Quell- und ein Zielmodell mit unterschiedlichen Metamodellen. Das Metamodell der Quelle ist ein Ausschnitt aus Ecore, das des Zielmodells ein Teil des Java-Metamodells aus Abb. 6.16. Ergebnis der Transformation ist ein Java-Modell, das alle `EPackage`- und `EClass`-Instanzen aus dem Quellmodell als `JavaPackage`- bzw. `JavaClass`-Instanzen übernimmt, `EDataType`-Instanzen jedoch nicht.

Module fassen in ATL eine Menge von Transformationsregeln zusammen. Zeilen 1 und 2 definieren ein Modul `ecore2java`, dessen Regeln aus einem Ecore-Modell ein Java-Modell erzeugen. Die beteiligten Modelle werden bei Aufruf an das Modul übergeben und an die jeweiligen Variablen `IN` und `OUT` gebunden. Die dazugehörigen Metamodelle `JavaMM` und `Ecore` werden ebenfalls als in XMI kodierte Ecore-Instanzen übergeben.

Kern einer ATL-Transformation bilden die deklarativen Transformationsregeln[31]. Sie bestehen aus einem Quellmuster (Schlüsselwort `from`) und einem Zielmuster (Schlüsselwort `to`). Eine Regel findet für alle Elemente aus dem Quellmodell Anwendung, auf die das Quellmuster passt. Das Quellmuster ist durch einen Quelltyp (Klasse des Quellmetamodells) und ein optionales Prädikat, ein boolscher OCL-Ausdruck (Object Constraint Language), bestimmt. Das Zielmuster besteht aus einer Menge von Zieltypen (Klassen des Zielmetamodells), die jeweils eine Menge von Zuweisungen besitzen. In einer Zuweisung wird einem Strukturmerkmal des Zieltyps (also einer Instanz der Meta-Metaklassen `EAttribute` oder `EReference`) der Wert eines OCL-Ausdrucks über dem Quellmodell zugewiesen.

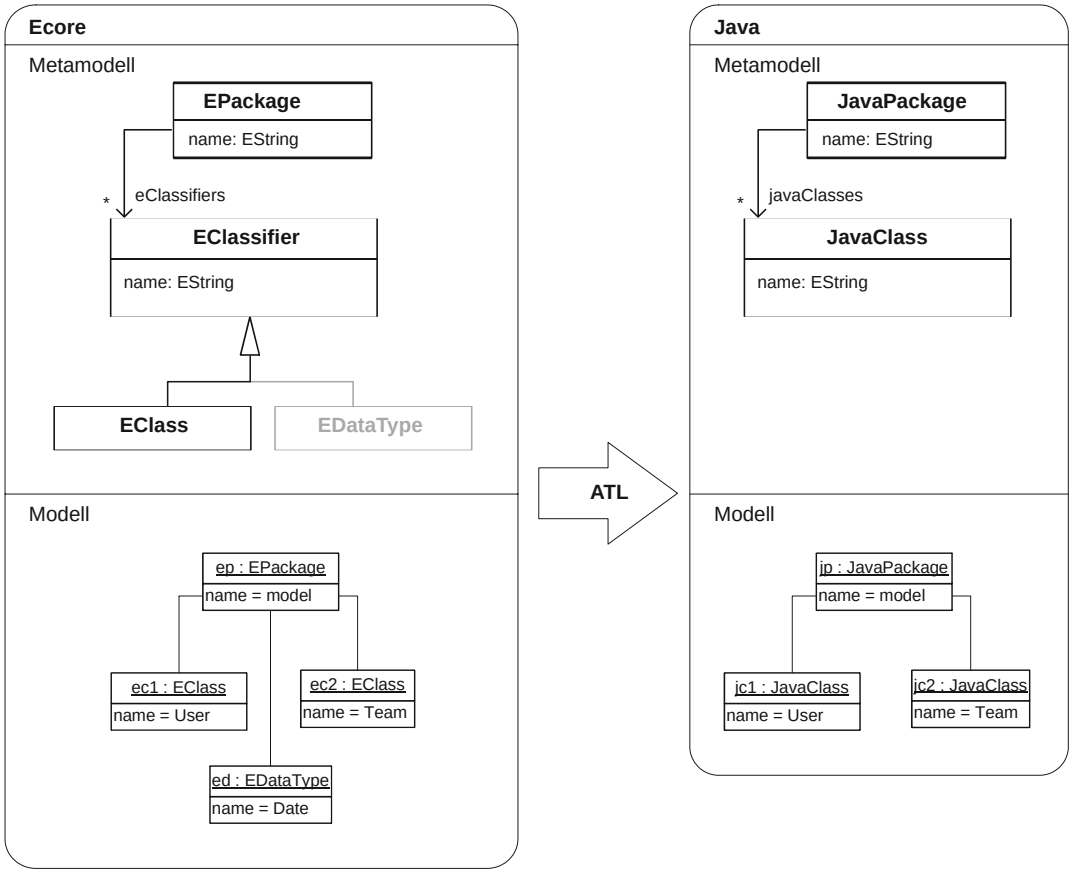
Das Quellmuster aus Zeile 17 passt zu jeder `EClass`-Instanz, da kein weiteres Prädikat angegeben ist. Das Zielmuster ab Zeile 19 besteht aus nur einem Zieltyp (`java!JavaClass`) und wird für jede `EClass`-Instanz, gebunden an die Variable `ec` (Zeile 17), aus dem Quellmuster angewandt. Es entsteht also für jede `EClass`-Instanz eine `JavaClass`-Instanz im Zielmodell. In der einzigen Zuweisung dieses Zieltyps bestimmt der OCL-Ausdruck `ic.name` den Wert des `JavaClass`-Attributs `name` (Zeile 20). Diese Zuweisung ist möglich, da das Ergebnis des OCL-Ausdrucks und das `JavaClass`-Attribut in diesem Fall denselben Typ besitzen[32]. Bei der Zuweisung aus

[31] ATL erlaubt auch den imperativen Regelaufruf, was aber hier nicht betrachtet werden soll.

[32] Das Java-Metamodell leiht den Datentyp EString aus Ecore, um das Beispiel schlank zu halten. Wollte man dort keine Ecore-Elemente auf Metaebene verwenden, müsste zusätzlich ein Datentyp String defi-

Zeile 10 ist dies nicht möglich, da das Ergebnis des OCL-Ausdrucks eine Menge von `EClass`-Instanzen ist, die Referenz `javaClasses` jedoch eine Menge von `JavaClass`-Instanzen erwartet. In einem solchen Fall wird eine Regel mit passenden Quell- und Zielmustern angewandt. Hier ist dies die Regel `Class` ab Zeile 15, die für die `EClass`-Instanzen des OCL-Ausdrucks `JavaClass`-Instanzen liefert.

niert werden (Meta-Metaklasse EDataType), dessen Wert für das Attribut instanceClassName java.lang.String lautet, so wie dies auch bei dem Datentyp EString in Ecore der Fall ist. Damit wäre gewährleistet, dass einem Attribut vom Datentyp String im Zielmodell (Java) ein Wert vom Datentyp EString im Quellmodell (Ecore) zugewiesen werden kann, da beide Datentypen von java.lang.String repräsentiert werden.




```
01 module ecore2java;
02 create OUT : JavaMM from IN : Ecore;
03
04 rule Package {
05     from
06             ep : Ecore!EPackage
07     to
08             jp : java!JavaPackage (
09                    name <- ip.name,
10                    javaClasses <- ip.eClassifiers->select(cl |
11                                   cl.oclIsKindOf(Ecore!EClass))
12             )
13 }
14
15 rule Class {
16     from
17             ec : Ecore!EClass
18     to
19             jc : java!JavaClass (
20                    name <- ic.name
21             )
22 }
```

Abb. 6.17 Beispiel einer ATL-Transformation

Interner Ablauf einer ATL-Transformation

Charakteristisch für ATL-Transformationen ist, dass sie in zwei Phasen ablaufen. In der ersten Phase werden „Trace-Links“ erzeugt, die in der zweiten Phase für die Zuweisungen genutzt werden ([JoKu05]). Ein Trace-Link ist ein 3-Tupel aus Regel, einem zum Quellmuster der Regel passenden Quellmodellelement und der Menge der für diese Regel und dieses Quellelement erzeugten Zielmodellelemente. Die Zielmodellelemente

werden zunächst nur erzeugt, ohne die Zuweisungen auszuführen. Für jede Regel werden in der ersten Phase alle Trace-Links erzeugt, wie Abb. 6.18 anhand des Beispiels aus Abb. 6.17 verdeutlicht. Da hier immer nur ein Zieltyp pro Zielmuster existiert, enthält die Menge der Zielmodellelemente jeweils nur ein Element.

```
traceLink1 = (Package, ep:EPackage, {jp:JavaPackage})
traceLink2 = (Class, ec1:EClass, {jc1:JavaClass})
traceLink3 = (Class, ec2:EClass, {jc2:JavaClass})
```

Abb. 6.18 Trace-Links des Beispiels aus Abb. 6.17 als Ergebnis der ersten Transformationsphase

In Phase 2 werden dann für jedes Zielmodellelement eines jeden Trace-Links die Zuweisungen ausgeführt. Die Lösung für das bereits skizzierte Problem bei inkompatiblen Zuweisungen aus Abb. 6.17, Zeile 10 wird nun deutlicher. Der OCL-Ausdruck liefert die Menge `{ec1, ec2}`. Für ec1 wird nun der TraceLink gesucht, der `ec1` als Quellmodellelement und eine `JavaClass`-Instanz in der Menge der Zielmodellelemente besitzt (`traceLink2`). Ist der Trace-Link identifiziert, können die entsprechenden Zuweisungen der dort verzeichneten Regel (`Class`) ausgeführt werden. Für `ec2` wird analog verfahren.

Da bei ATL auf Quellmodelle nur lesend (OCL ist frei von Seiteneffekten [OMG05b], S. 21) und auf Zielmodelle nur schreibend (OCL-Ausdrücke betreffen nur Quellmodelle) zugegriffen wird, ist diese Aufspaltung in zwei Phasen möglich und die Anwendungsreihenfolge der Regeln beliebig.

6.4.2 ATL-Transformation zur Generierung der Entity Beans

Der Einsatz von ATL im Rahmen des Fallbeispiels soll nun anhand der Erzeugung des Java-Modells, das die Entity Beans beschreibt, konkretisiert werden. Zur Wahrung der Übersichtlichkeit entfallen die Transformationsregeln zur Behandlung von Beziehungen.

```
module orm; create OUT : java from IN : Ecore, IN2: orm;
```

Die Moduldefinition zu Beginn verlangt zwei Eingabemodelle (`IN`, `IN2`) mit zugehörigen Metamodellen (`Ecore`, `orm`). Diese werden bei Aufruf der Transformation übergeben. `IN` bindet das in Ecore beschriebene Klassenmodell und `IN2` das gemäß Persistenz-Metamodell (vgl. Abb. 6.14) definierte Persistenzmodell. Modelle als auch die passenden Metamodelle werden als in XMI kodierte Dateien übergeben. Ebenso verhält es sich mit Modell und Metamodell (`java`) der Ausgabe, nur wird die an `OUT` gebundene Datei bei jeder Transformation überschrieben.

Die weiteren Ausführungen folgen der Ecore-Struktur. Sie beginnen außen mit `EPackage` und gehen über `EClass` zu `EAttribute`.

Transformation der Pakete

Bei der Transformation von `EPackage`- zu `JavaPackage`-Instanzen wird im Zielmodell zusätzlich das Paket[33] `javax.persistence` angelegt. Es enthält die Implementierung der Annotationstypen aus [Sun06c], und jede Entity Bean muss es importieren ([Sun06c], S. 163). Außerdem werden die Datentypen für das ID-, Diskriminator- und

[33] Es handelt sich dabei formal um zwei Instanzen von JavaPackage, wobei persistence in javax enthalten ist (Referenz superPackage)

Versionsattribut erzeugt (vgl. Kapitel 6.2), da diese Attribute spezifisch für Entity Beans und somit nicht im Klassenmodell definiert sind. Abbildung 6.19 zeigt die entsprechende Regel.

```
rule Package {
        from
                ip : Ecore!EPackage
        to
                op : java!JavaPackage (
                        name <- ip.name,
                        superPackage <- ip.eSuperPackage,
                        javaClasses <- ip.eClassifiers
                ),
                persistencePackage : java!JavaPackage (
                        name <- 'persistence',
                        superPackage <- javaxPackage
                ),
                javaxPackage : java!JavaPackage (
                        name <- 'javax'
                ),
                idType : java!JavaDataType (
                        name <- 'Orm_IdType',
                        javaName <- 'long'
                ),
                discrType : java!JavaDataType (
                        name <- 'Orm_DiscrType',
                        javaName <- 'java.lang.String'
                )
                versionType : java!JavaDataType (
                        name <- 'Orm_VersionType',
                        javaName <- 'int'
                ),
[…]
}
```

Abb. 6.19 ATL-Regel zur Erzeugung der Pakete und Datentypen im Java-Modell (Ausschnitt)

Das Zielmuster (der `to`-Teil der Regel) enthält sechs Zieltypen, für jedes zu erstellende Element im Zielmodell einen. Da das Quellmuster (`from`-Bereich) auf jede `EPackage`-Instanz passt, werden für jede `EPackage`-Instanz die sechs beschriebenen Elemente im Java-Modell erzeugt.

Transformation der Klassen

Existieren mehrere Zieltypen pro Zielmuster, d. h. für ein Element des Quellmodells werden mehrere Zielmodellelemente erstellt, erweist sich Navigation in ATL als problematisch. Konkret äußert sich dies u. a. bei der Erzeugung der Java-Klassen für die Entity Beans (Abb. 6.20). Wie bereits erwähnt, muss jede dieser Klassen das Paket `javax.persistence` importieren (genauer das Paket `persistence`, welches `javax` als `superPackage` besitzt), das durch den zweiten Zieltyp aus Abb. 6.19 erzeugt wurde. Ein OCL-Ausdruck auf der rechten Seite einer entsprechenden Zuweisung `packageImports <- …` kann jedoch nur über dem Quellmodell ausgewertet werden und liefert als Ergebnis lediglich das `EPackage`-Objekt, für das die Regel aus Abb. 6.19 jedoch sechs Zieltypen aufweist. ATL kann jetzt nicht ohne weiteres entscheiden, welcher der Zieltypen zugewiesen werden soll und wählt stets den ersten und in diesem Fall falschen.

Die Lösung birgt eine wenig intuitive, im Kontext des Moduls definierte, Hilfsfunktion `thisModule.resolveTemp()`, die zwei Argumente erwartet ([Atla06], S. 16). Das erste ist der OCL-Ausdruck, der das Quellelement liefert. Das zweite ist ein String, der dem Variablenbezeichner des Zieltyps, welcher zu der zum Quellelement passenden Regel gehört, entspricht. Im konkreten Fall werden also der OCL-Ausdruck

`ic.ePackage` und der String `'persistencePackage'` übergeben, wobei die Variable `ic` den Quelltyp `EClass` bindet (Abb. 6.20, erster Zieltyp, zweite Zuweisung). Somit kann nach erfolgter Navigation zu `EPackage` anhand des zweiten Arguments das Paket `persistence` ausgewählt und zugewiesen werden.

```
rule Class {
        from
                ic : Ecore!EClass
        to
                oc : java!JavaClass (
                        name <- ic.name,
                        packageImports <- thisModule.resolveTemp( ic.ePackage,
'persistencePackage' ),
                        extendsClass <- ic.eSuperTypes->select(e | e.interface =
false)->first(),
                        implementsInterfaces <- ic.eSuperTypes->select(e | e.interface
= true),
                        fields <- ic.eStructuralFeatures->select(sf |
sf.oclIsKindOf(Ecore!EAttribute))
                ),

                idField : java!Field (
                        name <- 'id',
                        javaClass <- ic,
                        type <- thisModule.resolveTemp( ic.ePackage, 'idType' ),
                        javaVisibility <- #PRIVATE
                ),
                idGetter : java!Method (
                        name <- 'getId',
                        returnType <- thisModule.resolveTemp(ic.ePackage, 'idType'),
                        javaClass <- ic,
                        annotations <- idAnnotation,
                        source <- idGetterImpl
                ),
                idAnnotation : java!Annotation (
                        name <- 'Id'
                ),
                idGetterImpl : java!Block (
                        source <- '{return id;}';
                ),
[…]
}
```

Abb. 6.20 ATL-Regel zur Erzeugung der Java-Klasse einer Entity Bean (Ausschnitt)

Die übrigen Zieltypen in Abb. 6.20 generieren das ID-Attribut, zugehörige Zugriffsmethoden (hier nur die `get`-Methode) und eine Annotation, die dieses Attribut als Objektidentifikator kennzeichnet. Die Annotation haftet hier an der `get`-Zugriffsmethode statt am Attribut selbst. Beide Varianten sind laut [Sun06c] möglich und gleichbedeutend.

Ähnlich wird das Versionsattribut behandelt. Für die Erzeugung der klassenbezogenen Annotationen wird eine Transformation, ausgehend vom Persistenzmodell, ausgeführt (Abb. 6.21).

```
rule EntityAnnotation {
	from
		it : orm!OrmTable
	to
		oa : java!Annotation (
			name <- 'Entity',
			annotatable <- it.realMappedClass()
		),
		tableAnnotation : java!Annotation (
			name <- 'Table',
			annotatable <- it.realMappedClass(),
			annotationElements <- tableName
		),
		tableName : java!BasicAnnotationElement (
			name <- 'name',
			values <- '"' + if it.name.oclIsUndefined() then
it.mappedClass.name else it.name endif + '"'
		)
}
```

Abb. 6.21 ATL-Regel zur Erzeugung der klassenbezogenen Annotationen für Entity Beans

Bei Intermodellreferenzen offenbart ATL eine weitere Schwäche, die mit eigenen Hilfsfunktionen umgangen werden muss. Die Regel aus Abb. 6.21 erzeugt für jede `OrmTable`-Instanz im Persistenzmodell eine `Entity`- und eine `Table`-Annotation. Die Metaklasse `OrmTable` enthält die Referenz `mappedClass`, die auf die zu annotierende Klasse aus dem Ecore-Modell verweist. Konkret stellt EMF bei der Modellierung diese Beziehung mit der URI (Uniform Resource Identifier) der Ecore-Klasse her und speichert sie in der XMI-Datei des Persistenzmodells. Um die Annotationen an die korrekte Klasse zu binden, sollte also in der ATL-Transformation die Zuweisung `annotatable <- it.mappedClass` innerhalb des Zieltyps genügen.

Leider erzeugt ATL sowohl für die `EClass`-Instanz aus dem Klassenmodell als auch für die referenzierte `EClass`-Instanz aus dem Persistenzmodell jeweils ein Objekt, obwohl beide identisch sind. Demnach lieferte die Zuweisung `annotatable <- it.mappedClass` das für das Persistenzmodell erzeugte und an den „Namensraum“ `IN2` gebundene `EClass`-Objekt. Doch dafür schalten die Regeln nicht, die für das Ecore-Metamodell bestimmt sind, da diese offenbar ausschließlich Objekte aus dem Namensraum `IN` akzeptieren.

Zur Lösung dieses Problems wurde die Hilfsfunktion `realMappedClass()` im Kontext von `OrmTable` definiert, die jede `EClass`-Instanz aus dem Klassenmodell (`IN`) mit der durch `mappedClass` referenzierten `EClass`-Instanz (`IN2`) auf Gleichheit prüft und das so gefundene Objekt aus `IN` zurückliefert. Der Test auf Gleichheit wurde ebenfalls mit Hilfsfunktionen wie folgt umgesetzt (Der Test auf Gleichheit von `EAttribute`-Instanzen wird später bei deren Transformation benötigt):

- Zwei `EAttribute`-Instanzen sind gleich gdw. sie den gleichen Namen besitzen und die sie enthaltenden `EClass`-Instanzen gleich sind.
- Zwei `EClass`-Instanzen sind gleich gdw. sie den gleichen Namen besitzen und die sie enthaltenden `EPackage`-Instanzen gleich sind.

- Zwei `EPackage`-Instanzen sind gleich gdw. sie den gleichen Namen besitzen und die sie enthaltenden `EPackage`-Instanzen gleich sind.

Abbildung 6.22 zeigt die oben beschriebene Arbeitsweise der Hilfsfunktion `realMappedClass()`. Sie wählt aus allen `EClass`-Instanzen aus `IN` diejenige aus, die der durch `mappedClass` referenzierten aus `IN2` nach obiger Definition gleicht.

```
helper context orm!OrmTable def: realMappedClass() : Ecore!EClass =
       (Ecore!EClass.allInstances()->select(e | e.equals(self.mappedClass)))-
>first();
```

Abb. 6.22 ATL-Hilfsfunktion zum Auffinden der abzubildenden Ecore-Klasse

Transformation der Attribute

Bei der Transformation der Attribute tritt die Problematik der Intermodellreferenzen ebenfalls auf und wird ähnlich gelöst (Abb. 6.23). Hier wird jedoch die Gleichheit ausgehend vom Ecore-Modell geprüft. Dies ist nötig, da für ein `EAttribute` entschieden werden muss, ob es persistent ist oder nicht. Soll ein `EAttribute` nicht persistent sein, so muss ein `TransientColumn`-Objekt aus dem Persistenzmodell es referenzieren. Die Überprüfung erfolgt in der ersten Zuweisung des Zieltyps `fieldAnnotation`.

```
rule Attribute {
        from
                ia : Ecore!EAttribute
        to
                of : java!Field (
                        name <- ia.name,
                        type <- ia.eAttributeType,
                        javaVisibility <- #PRIVATE
                ),
                getter : java!Method (
                        name <- 'get' + ia.name.firstToUpper(),
                        returnType <- ia.eAttributeType,
                        javaClass <- ia.eContainingClass,
                        source <- getterImpl,
                        annotations <- fieldAnnotation
                ),
                fieldAnnotation : java!Annotation(
                        name <- if orm!TransientColumn.allInstances()->select(e |
ia.equals( e.mappedAttribute ))->size() > 0 then 'Transient' else 'Column' endif
                ),
                getterImpl : java!Block (
                        source <- ia.name.javaGetterImpl()
                ),
[…]
}
```

Abb. 6.23 ATL-Regel zur Erzeugung der Attribute einer Entity Bean (Ausschnitt)

6.4.3 Ergebnis der ersten Transformation

Das Ergebnis der ersten Transformation fasst Abb. 6.24 zusammen. Um besonders das Java-Modell in dieser Darstellung nicht zu überladen, wurden die ID- und Versionsattribute sowie deren Zugriffsmethoden und Annotationen nicht aufgenommen. Ebenso fehlen die Datentypen. Objektbeziehungen wurden nur benannt, wenn Missverständnisse auftreten könnten.

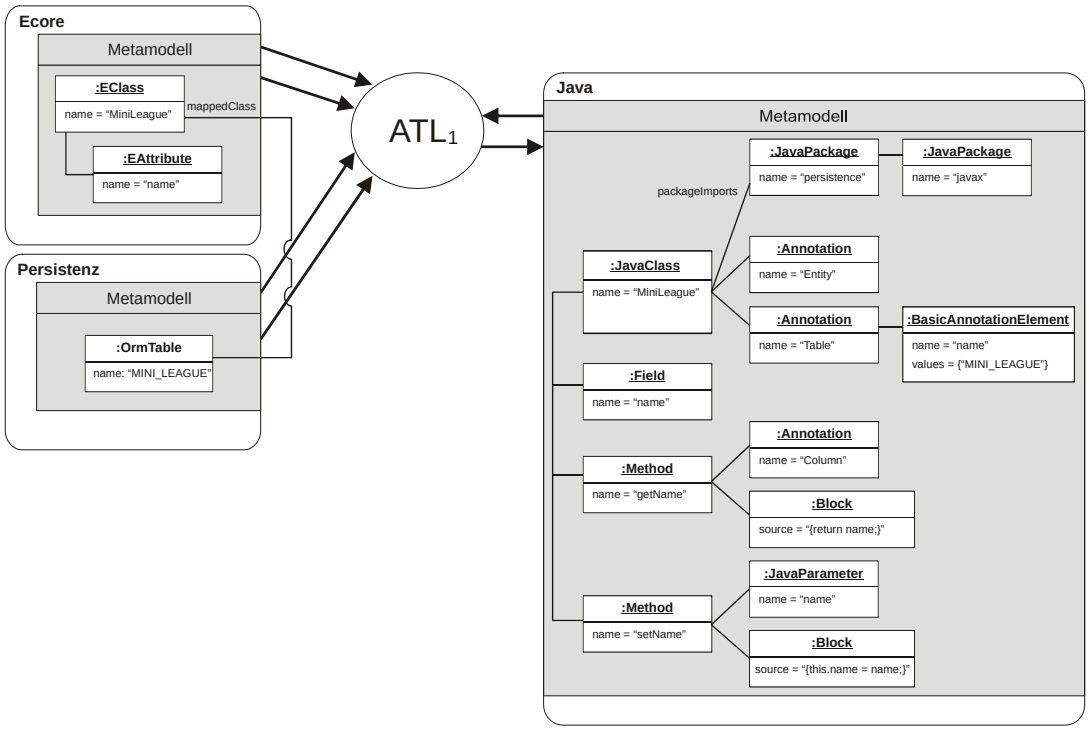


Abb. 6.24 Aus Klassen- und Persistenzmodell generiertes Java-Modell (Ausschnitt)

Ausgehend vom Java-Modell kann nun eine weitere ATL-Transformation entsprechenden Java-Quellcode generieren. Diese Transformation ist nicht auf die Erzeugung von Code für Entity Beans beschränkt, sondern kann für beliebige Java-Modelle wiederverwendet werden.

6.4.4 ATL-Transformation zur Generierung von Java-Code

ATL bietet das Konzept der Query, um aus Eingabemodellen den Wert eines primitiven Datentyps zu erzeugen, in diesem Fall einen String ([Atla06], S. 12). Die Definition der Query für die Codeerzeugung lautet wie folgt:

```
query Java2Code = Java!JavaClass.allInstances()->collect(e | e.toString().writeTo(
       'c:/temp/atlout/' + e.javaPackage.toString().replaceAll('.', '/') + '/' + e.name
       + '.java'));
```

Für alle `JavaClass`-Instanzen wird die Hilfsfunktion `toString()` aufgerufen, die den Quellcode der Klasse als String zurückliefert. Die im Kontext von String definierte Hilfsfunktion `writeTo()` erwartet einen Pfad zur Ausgabe des Strings in eine Datei[34].

[34] Die String-Funktionen sind in einer ATL-Bibliothek strings zusammengefasst, die jedoch nicht dokumentiert ist. Die hier genutzten Funktionen wurden in den Transformationsbeispielen unter http://www.eclipse.org/gmt/atl/atlTransformations/ entdeckt.

Die `toString()`-Funktion von `JavaClass` nutzt wiederum `toString()`-Funktionen von `JavaPackage`, `Annotation`, `Method`, `Field` etc., um deren String-Repräsentation und die eigene mittels Operator + zu verknüpfen. Diese Vorgehensweise pflanzt sich für die weiteren Elemente entsprechend fort.

Als Beispiel soll die Codegenerierung für Annotationen in Abb. 6.25 dienen. Die Funktionsdefinition in Zeile 1 gibt den Namen (`toString`) und Rückgabetyp (`String`) an. Zeile 2 setzt vor den Annotationsnamen das Klammeraffensymbol und dahinter ein Leerzeichen. Falls die Annotation Annotationselemente enthält (`if-then-else`-Operator in Zeile 3 und 6), wird für jedes in Zeile 4 und 5 die `toString()`-Funktion aufgerufen und das Ergebnis in Klammersymbole eingeschlossen. Die Prüfung in Zeile 5 ist nötig, um Kommasymbole nur zwischen Annotationselementen einzufügen.

```
1 helper context Java!Annotation def: toString() : String =
2       '@' + self.name + ' ' +
3       if self.annotationElements->size() > 0 then
4               '(' + self.annotationElements->iterate(e; acc : String = '' | acc +
5                if acc = '' then '' else ', ' endif + e.toString() ) + ')'
6       else '' endif + '\n';
```

Abb. 6.25 ATL-Funktion zur Erzeugung von Code für Annotationen

Für diese grausame Methode der Codegenerierung spendet lediglich der Umstand Trost, dass die ATL-Transformation für beliebige Java-Modelle verwendet werden kann und nur bei seltener Änderung des Java-Sprachstandards angepasst werden muss.

6.4.5 Ergebnis der zweiten Transformation

Den durch die zweite Transformation erzeugten Java-Quellcode zeigt Abb. 6.26, wobei die Codezeilen grau dargestellt sind, die aus den Elementen generiert wurden, welche wegen der Übersichtlichkeit nicht im Java-Modell aufgenommenen wurden.

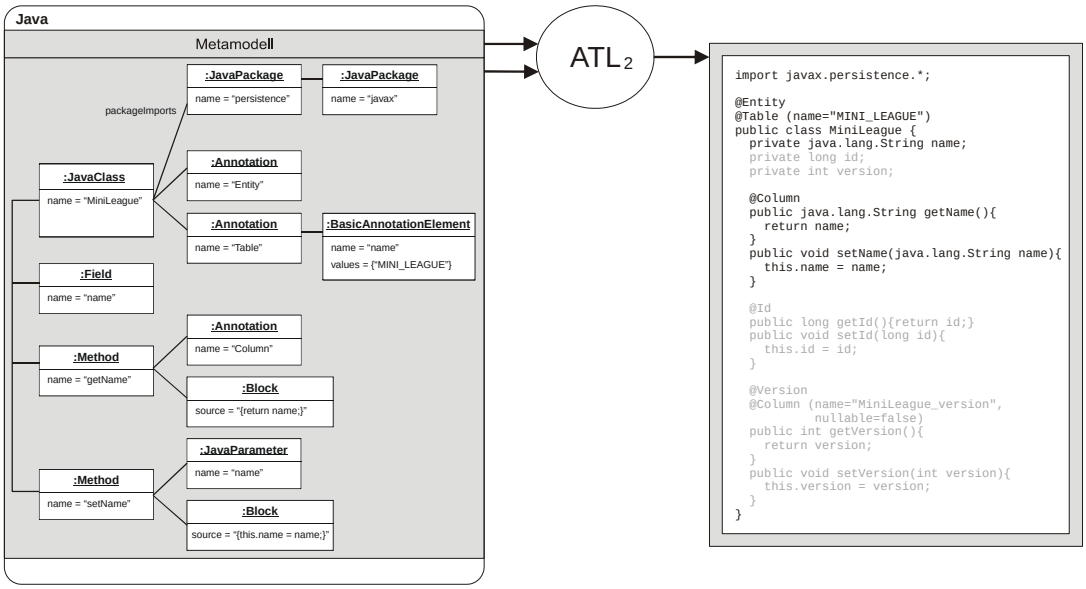


Abb. 6.26 Aus Java-Modell generierter Quellcode

7 Generierung der JavaServer-Faces-Artefakte

Dieses Kapitel beschreibt die Technologie JavaServer Faces (JSF) und die Generierung der Artefakte, die in diesem Zusammenhang im Fallbeispiel zur Erstellung der Benutzeroberfläche und der Verarbeitung der HTTP-Anfragen zum Einsatz kommen. Es sind die in Kapitel 5 bereits skizzierten Model- und Controller-Klassen sowie in XML codierte JSF-Konfigurationsdateien. Die dafür eingesetzte Transformationskette (Abb. 7.1) benötigt neben einem Metamodell für XML ein weiteres Ausgangsmodell und Metamodell, das die Anfragen an das System und die an der Verarbeitung beteiligten Objekte beschreibt. Die Transformation zur Generierung des Java-Quellcodes (ATL_2) ist identisch mit der aus Abb. 6.1.

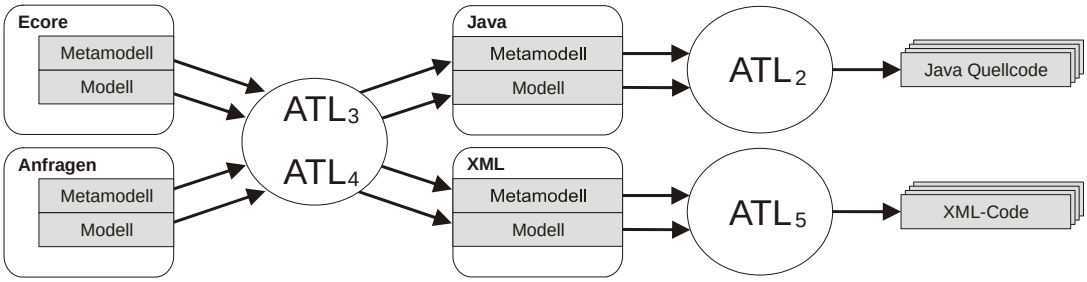


Abb. 7.1 Übersicht der Transformationen zur Erzeugung der JSF-Artefakte

Ein einführendes Beispiel für die Transformation der Ausgangsmodelle in XML zur Konfiguration von JSF gibt Abb. 7.2. Die Modellausschnitte skizzieren das Model für die Login-Operation und wie JSF es verwalten soll. Das Ecore-Modell ist dasselbe, das zur Erzeugung der Entity Beans benutzt wird.

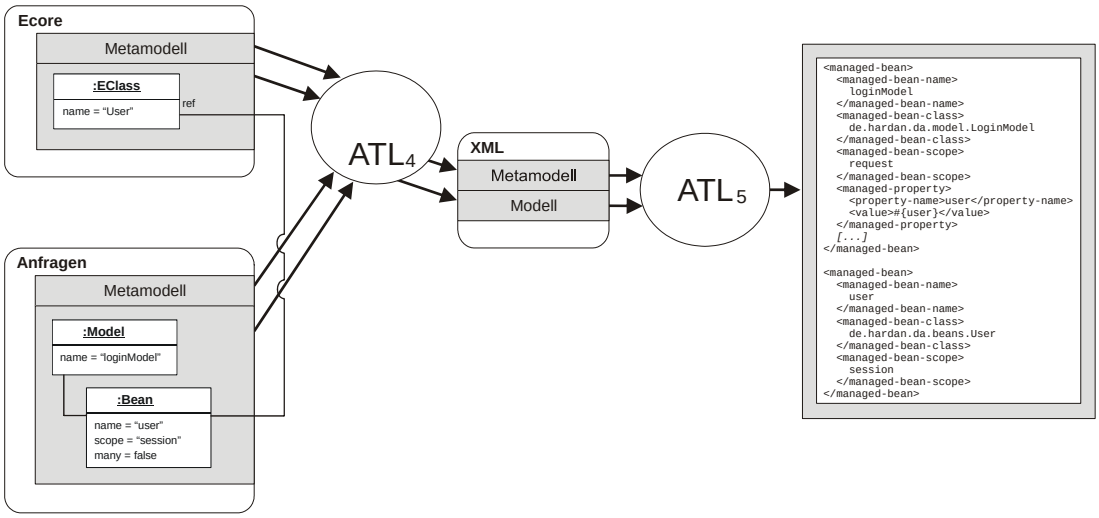


Abb. 7.2 Einführendes Beispiel zur Erzeugung der JSF-Konfiguration

Nach der Einführung in das Einsatzgebiet und die Ziele von JSF wird anhand eines Beispiels die Arbeitsweise von JSF untersucht. Zuletzt werden die ATL-Transformationen und Metamodelle zur Erzeugung der in Kapitel 5 skizzierten Model- und Controller-Klassen sowie JSF-Konfigurationsdateien beschrieben.

7.1 Einführung in Java Server Faces

Seit Einführung der Servlet- und JSP-Technologie zur Erstellung dynamischer Weboberflächen Ende der neunziger Jahre sind zahlreiche Frameworks auf deren Basis entstanden (u. a. Struts[35], WebWork[36], Tapestry[37], Turbine[38]), die häufig benötigte, aber von den Spezifikationen nicht abgedeckte Funktionen bereitstellen. Die häufigsten Ergänzungen betreffen:

- Verwaltung des Zustandes der Oberflächenelemente (z. B. Inhalt eines Eingabefeldes) über mehrere HTTP-Anfragen hinweg
- Binden von HTML-Formularfeldern an Objektattribute. Fragt z. B. ein HTML-Formular personenbezogene Daten wie Name, Geburtsdatum etc. ab, so sollen diese Felder an die entsprechenden Attribute eines Person-Objekts gebunden werden können. Binden heißt hier, dass das Framework Formulareingaben in die korrekten Typen (z. B. Datum) konvertiert und dem gebundenen Attribut zuweist. Ebenso kann umgekehrt ein HTML-Formular mit den gebundenen Attributen eines Objekts befüllt werden.
- Validierung (Plausibilitätsprüfung) von Benutzereingaben
- Anzeige von Fehlern (Konvertierungs-, Validierungs- und allgemeine Anwendungsfehler) in menschenlesbarer Form
- Verwaltung der Navigationsstruktur. Eine feste Verdrahtung von HTML-Seiten über direkte Links soll durch eine flexible und durch die Anwendung zur Laufzeit veränderbare Verknüpfung ersetzt werden.

Die JSF-Spezifikation [Sun06a] strebt an, den Wildwuchs an Frameworks zur Webentwicklung mit Java zu beschneiden und eine standardisierte Lösung zu etablieren. Dazu präsentiert sie sich im Kern zwar unabhängig vom Transportprotokoll HTTP und der Darstellung mit HTML, setzt jedoch dort ihren Schwerpunkt ([Sun06a], S. 1-20).

[35] http://struts.apache.org/

[36] http://www.opensymphony.com/webwork/

[37] http://jakarta.apache.org/tapestry/

[38] http://jakarta.apache.org/turbine/

Zusätzlich zu den oben aufgeführten Funktionen bietet JSF:

- Eine erweiterbare Menge wiederverwendbarer und konfigurierbarer Komponenten für die Benutzeroberfläche. Dies können einfache Komponenten (Eingabefelder, Buttons) sein oder komplexe, aus anderen Komponenten zusammengesetzte (Tabellen, Baumansichten). Die Benutzeroberfläche entsteht durch Komposition.
- Ein typsicheres Ereignismodell. Ereignisse in der Benutzeroberfläche des Clients (z. B. das Klicken auf einen Button) werden auf dem Server behandelt, ohne dass der Entwickler das zugrundeliegende Transportprotokoll berücksichtigen muss.

All diese Aspekte sollen es ermöglichen, bewährte Konzepte bei der Entwicklung von Benutzeroberflächen aus dem Desktopbereich, wie der Model-View-Controller-Ansatz, in die Webentwicklung zu übertragen.

In Kapitel 7.2 werden beispielhaft die Arbeitsweise von JSF erläutert und einige der hier aufgezählten Eigenschaften konkretisiert. Dazu erfolgt zunächst eine isolierte Betrachtung der wichtigsten vom Anwendungsentwickler zu erstellenden Artefakte. Danach wird auf deren Verknüpfung eingegangen und am Schluss das Verhalten zur Laufzeit während der Anfrageverarbeitung analysiert.

7.2 Die Arbeitsweise von JSF anhand eines Beispiels

Das einfache Beispielformular aus Abb. 7.3a dient als Ausgangspunkt zur Funktionsbeschreibung von JSF. Anwendungsfall ist eine Benutzerregistrierung, bei der Name und Geburtsdatum angegeben werden sollen. Dabei ist der Name ein Pflichtfeld, das mindestens 2 und höchstens 40 Zeichen enthalten darf. Die Angabe des Geburtsdatums ist optional. Sofern ein Eintrag vorliegt, muss dieser dem Format TT.MM.JJJJ, z. B. 17.08.1974, entsprechen.

Abb. 7.3a Mit JSF erzeugtes HTML-Formular

Abbildung 7.3b zeigt die Komposition der JSF-Komponenten zur Erstellung des obigen Formulars in JSP-Notation. Implementierungen der JSF-Spezifikation müssen die Integration mit JSP leisten ([Sun06a], S. 9-1). Für jede JSF-Komponente existiert eine

entsprechende Tag-Extension, die in eine JSP-Seite eingebunden werden kann. Dabei werden die JSF-Komponenten durch zwei Tag-Libraries repräsentiert (Zeilen 1 und 2). Die erste enthält die von der Darstellung (HTML) unabhängigen Komponenten, z. B. Typkonverter und Validatoren, und besitzt die URI `http://java.sun.com/jsf/core`. Als Präfix wird per Konvention der Buchstabe `f` genutzt. Die zweite mit der URI `http://java.sun.com/jsf/html` (Präfix `h`) ist spezifisch für HTML und enthält visuelle Komponenten wie Eingabefelder oder Buttons.

```
01 <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
02 <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   […]
03 <f:view>
04
05 <h1>Registrierung</h1>
06
07 <h:form id="userForm">
08
09     <h:panelGrid columns="3">
10             <h:outputLabel for="name" id=”nameLabel”>
11                     <h:outputText value="Name:"/>
12             </h:outputLabel>
13
14             <h:inputText id="name" value="#{Registration.user.name}"
15                          required="true">
16                     <f:validateLength minimum="2" maximum="40"/>
17             </h:inputText>
18
19             <h:message style="color: red" id="nameError" for="name"/>
20
21             <h:outputLabel for="birthDate" id="birthDateLabel">
22                     <h:outputText value="Geburtsdatum:"/>
23             </h:outputLabel>
24
25             <h:inputText id="birthDate" value="#{Registration.user.birthDate}">
26                     <f:convertDateTime pattern="dd.MM.yyyy"/>
27             </h:inputText>
28
29             <h:message style="color: red" id="birthDateError" for="birthDate"/>
30
31     </h:panelGrid>
32
33     <h:commandButton id="register" action="#{Registration.register}"
34                     value="registrieren"/>
35
36 </h:form>
37
38 </f:view>
```

Abb. 7.3b Komposition von JSF-Komponenten zur Erzeugung des HTML-Formulars aus Abb. 7.3a

Ohne auf die Bedeutung der einzelnen JSF-Komponenten aus Abb. 7.3b einzugehen, lässt sich erkennen, dass sie zum Aufbau einer Seite stets eine Baumstruktur bilden. Zu jeder dargestellten Seite, im JSF-Jargon View genannt, gehört ein solcher Komponentenbaum, dessen Wurzel die Tag-Extension `<f:view/>` festsetzt.

Weitere Teilnehmer zur Umsetzung des Registrierungsbeispiels sind die Klassen `User` (Abb. 7.4) und `RegistrationController` (Abb. 7.5). `User` ist die Model-Klasse, d. h., sie wird die Formulardaten für Name und Geburtsdatum aus dem View aufnehmen bzw. zur Anzeige an diesen liefern. Die Klasse `RegistrationController` dient als Controller und stößt die Anwendungslogik zur Verarbeitung des Models im Backend an (z. B. Speicherung des `User`-Objekts in einer Datenbank).

```
package de.hardan.jsfexample;

import java.util.Date;

public class User {
  private String name;
  private Date birthday

  public String getName() {return name;}
  public void setName(String name) {this.name = name;}

  public Date getBirthday() {return birthday;}
  public void setBirthday(Date birthday) {this.birthday = birthday;}
}
```

Abb. 7.4 Die Model-Klasse zum Aufnehmen der Formulareingaben

```
package de.hardan.jsfexample;

public class RegistrationController {
  private User user = new User();

  public User getUser() {return user;}
  public void setUser(User user) {this.user = user;}

  public String register() {
    //store user in DB
    return "success";
  }
}
```

Abb. 7.5 Die Controller-Klasse zum Ausführen der Anwendungslogik

Der Controller, auch als Backing- oder Managed-Bean bezeichnet, muss bei einer JSF-Implementierung registriert werden, damit diese ihn auffinden und nutzen kann. Diese Information liegt in einer XML-Datei, die sämtliche Konfigurationen für JSF enthält. Den entsprechenden Ausschnitt zur Controller-Konfiguration für die Benutzerregistrierung zeigt Abb. 7.6. Dort gibt der Wert `session` für `managed-bean-scope` an, dass die JSF-Implementierung pro Benutzersitzung genau eine Instanz der Klasse `RegistrationController` erzeugen soll.

```
<managed-bean>
  <description>
    Controller für die benutzerregistrierung
  </description>
  <managed-bean-name>Registration</managed-bean-name>
  <managed-bean-class>de.hardan.jsfexample.RegistrationController</managed-bean-
class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Abb. 7.6 Ausschnitt der Konfigurationsdatei faces-config.xml zur Definition eines Controllers

Bisher wurden die wichtigsten Artefakte vorgestellt, die zur Erstellung einer JSF-Anwendung nötig sind: Die JSF-Komponenten erzeugen durch Komposition eine darstellbare Seite (View). Gewöhnliche[39] Java-Klassen dienen als Model, das der View darstellt und das Eingaben aus dem View aufnimmt. Ebenso übernehmen gewöhnliche

[39] Es muss keine besondere Schnittstelle implementiert oder von JSF-spezifischen Klassen abgeleitet werden.

Klassen die Rolle des Controllers, der auf Ereignisse (z. B. das Absenden eines Formulars) aus dem View reagiert und Dienste des Backends zur Verarbeitung der Model-Daten nutzt.

Nach dieser isolierten und statischen Betrachtung gilt es in den folgenden Abschnitten zu klären, wie Model, View und Controller miteinander verknüpft werden und wie die Verarbeitung von (HTTP-) Anfragen zur Laufzeit vonstatten geht. Zur Klärung des ersten Aspektes dient die folgende Beschreibung der Unified[40] Expression Language (EL) anhand eines Ausschnitts des vorgestellten Komponentenbaums. Anschließend werden die sechs Phasen des Anfragelebenszyklus zur Klärung des zweiten Aspekts beleuchtet.

Anhand der Komponente für die Namenseingabe aus Abb. 7.3a sollen weitere Eigenschaften von JSF, insbesondere die Verbindung des Views mit Model und Controller, aufgezeigt werden. Abbildung 7.7 zeigt den entsprechenden Komponententeilbaum (die Zeilennummern stammen aus Abb. 7.3a).

```
14      <h:inputText id="name" value="#{Registration.user.name}"
15                   required="true">
16              <f:validateLength minimum="2" maximum="40"/>
17      </h:inputText>
```

Abb. 7.7 JSF-Komponenten zur Erzeugung eines Eingabefeldes

Die Tag-Extension `<h:inputText/>` entspricht einem einzeiligen Eingabefeld; in der HTML-Ausgabe wird also ein Formularfeld `<input type="text" …>` generiert. Die Angabe einer `id` (Zeile 14) ermöglicht es anderen Komponenten oder der Anwendung, dieses Eingabefeld zu referenzieren. Der Wert des Attributs `required` (Zeile 15) gibt an, ob eine Eingabe zwingend ist oder nicht.

Als Kindkomponente besitzt das Eingebefeld einen der in JSF bereits enthaltenen Validatoren zur Längenprüfung von Strings (Zeile 16), der in JSP-Syntax durch die Tag-Extension `<f:validateLength/>` repräsentiert wird. Dadurch ist sichergestellt, dass eine JSF-Implementierung nur Eingaben akzeptiert und an die Anwendung weitergibt, die in den angegebenen Grenzen liegen.

Der Wert des `value`-Attributs aus Zeile 14 nutzt die EL, um den Inhalt des Eingabefeldes an das `name`-Attribut des `User`-Objekts zu binden. Solche Werteverknüpfungen werden `ValueExpression` ganannt und gelten in beide Richtungen sowohl vom View zum Model (die Formulareingabe für den Benutzernamen wird im gebundenen `name`-Attribut des `User`-Objekts gespeichert) als auch vom Model zum View (der Wert des `name`-Attributs wird im Formular angezeigt).

Neben dieser Werteverknüpfung kann eine `MethodExpression` eine Methode an Ereignisse binden. Ein Beispiel dafür ist in Abb. 7.3b, Zeile 33 zu sehen, wo das Drücken eines Buttons den Aufruf der Methode `register` auslöst.

Wie JSF EL-Ausdrücke auswertet und so die Bindungen herstellen kann, zeigt der nächste Abschnitt anhand der Werteverknüpfung aus Abb. 7.7, Zeile 14.

[40] Vor Version 1.2 der JSF-Spezifikation und Version 2.1 der JSP-Spezifikation besaßen beide ähnliche aber inkompatible Sprachen zur Formulierung von Ausdrücken. Für die genannten Versionen wurde eine einheitliche Sprache geschaffen.

7.2.1 Die Unified Expression Language

Die meisten Komponentenattribute können neben Literalen auch in EL formulierte Ausdrücke als Wert aufnehmen. Die Auswertung eines EL-Ausdrucks erfolgt über mehrere `ELResolver`, die in einer „Chain of Responsibility“ ([GoF95]) zusammengeschaltet sind. Für jedes der durch Punkte getrennten Segmente des Ausdrucks werden alle `ELResolver` der Kette durchlaufen. Jeder `ELResolver` ist für einen festen Suchraum verantwortlich und signalisiert über einen gemeinsamen Kontext (`ELContext`), wenn er ein Segment erfolgreich auflösen konnte, sodass nachfolgende Resolver nicht mehr aufgerufen werden müssen. Für Lesezugriffe wird die Methode `getValue()` der `ELResolver` aufgerufen, für Schreibzugriffe `setValue()`. Abbildung 7.8 stellt die Auswertung des Ausdrucks aus dem `value`-Attribut (Abb. 7.7, Zeile 14) genauer dar.

Auswertung (lesend) von #{**`Registration.user.name`**}:

- Auswertung von **`Registration`** durch Aufruf von `getValue()` aller `ELResolver` in der Kette
  - Der `ManagedBean ELResolver` findet das zu `Registration` passende `RegistrationController`-Objekt im Sitzungskontext des Nutzers (vgl. Controller-Konfiguration in Abb. 7.6) und signalisiert dies über den `ELContext`. Es werden keine weiteren Resolver für dieses Segment abgefragt.
  - Das gefundene Objekt wird zurückgeliefert und dient als Basis für die folgende Auswertung.
- Auswertung von **`user`** durch Aufruf von `getValue()` aller `ELResolver`
  - Der `BeanELResolver` entdeckt im `RegistrationController`-Objekt, das als Parameter im Methodenaufruf von `getValue()` übergeben wurde, ein Attribut `user`, das über entsprechende Zugriffsmethoden gemäß JavaBean-Konvention (`getUser()`, `setUser()`) erreichbar ist (vgl. Abb. 7.5). Dieses Attribut wird über `getUser()` aus `RegistrationController` gelesen und als neue Basis für die folgende Auswertung zurückgeliefert. Über den `ELContext` wird das erfolgreiche Auflösen des Segments signalisiert, sodass keine weiteren Resolver zum Einsatz kommen.
- Auswertung von **`name`** durch Aufruf von `getValue()` aller `ELResolver`
  - Wieder findet der `BeanELResolver` das `name`-Attribut diesmal im als Basis übergebenen `User`-Objekt und reagiert wie zuvor, indem es `User.getName()` aufruft (vgl. Abb. 7.4), das erfolgreiche Auflösen signalisiert und das Ergebnis zurückliefert.
- Da alle Segmente abgearbeitet sind, enthält der letzte Rückgabewert das Ergebnis der Auswertung.

Abb. 7.8 Auswertung eines EL-Ausdrucks am Beispiel des Benutzernamens

Welche ELResolver in JSF vorkommen und wo eigene in die Kette eingebunden werden können, beschreibt [Sun06a] in Kapitel 5.6.

7.2.2 Anfrageverarbeitung in JSF

JSF definiert sechs Phasen zwischen Ankunft einer Anfrage und Rücksenden der Antwort (Abb. 7.9). Als Einstiegspunkt zur Bearbeitung von HTTP-Anfragen[41] enthält JSF das Servlet `javax.faces.webapps.FacesServlet`.

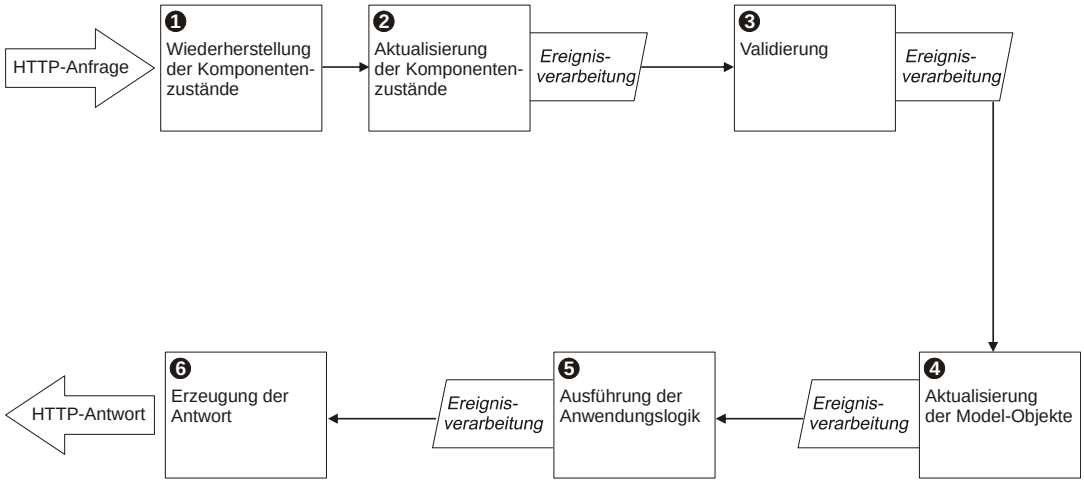


Abb. 7.9 Lebenszyklus einer HTTP-Anfrage in JSF

Nicht jede Anfrage führt zur Ausführung aller sechs Phasen. Schlägt z. B. bereits die Überprüfung der Längenbeschränkung eines Eingabetextes in Phase 3 fehl, dürfen die beiden folgenden Phasen nicht ausgeführt werden, da diese valide Werte erwarten. Die Reihenfolge der Phasen bleibt jedoch stets fest.

Den zentralen Mechanismus, um JSF und die eigene Anwendung zu integrieren, bilden Ereignisse, deren Behandlung nach Abschluss der Phasen 2-5 erfolgen kann. In JSF existieren zwei vorderfinierte Ereignistypen: `PhaseEvent` und `FacesEvent` (Abb. 7.10). JSF erzeugt einen PhaseEvent für jede der sechs Phasen und ermöglicht es der Anwendung, vor Ausführung und/oder nach Abschluss einer Phase Aktionen auszuführen. Ein `FacesEvent` geht stets von einer JSF-Komponente aus. Er besitzt zwei Spezialisierungen: den `ValueChangeEvent`, der angibt, dass der Wert z. B. eines Eingabefeldes geändert wurde, und den `ActionEvent`, der z. B. das Drücken eines Buttons signalisiert.

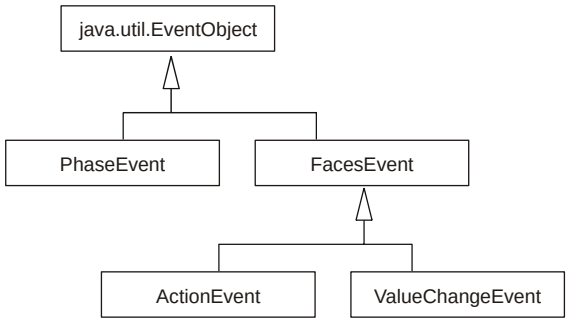


Abb. 7.10 Vordefinierte Ereignistypen in JSF (nach [Sun06a], S. 3-29)

[41] Grundsätzlich ist JSF unabhängig vom Transportprotokoll. Für die folgenden Beschreibungen wird jedoch stets HTTP angenommen, ebenso HTML für die Darstellung der Komponenten.

Details zu diesem Mechanismus werden in den folgenden Phasenbeschreibungen anhand der Benutzerregistrierung dargestellt, die bei Absenden des Formulars einen `ActionEvent` erzeugt.

Phase 1: Wiederherstellung der Komponentenzustände (Restore View)

In der ersten Phase werden für die anzuzeigende Seite der Komponentenbaum und sein Zustand wiederhergestellt, sofern beide bereits existieren. Dazu ermittelt JSF aus der Anfrage-URL eine `viewId`, die als Suchschlüssel zum Auffinden bereits gespeicherter Komponentenbäume dient. Für jeden Nutzer existieren natürlich eigene Suchräume und Komponentenzustände. Lässt sich kein Komponentenbaum finden, wird direkt mit Phase 6 fortgefahren, in der die Erzeugung eines neuen Baumes stattfindet.

Für das Formular aus Abb. 7.3a bedeutet dies, dass eine JSF-Implementierung bei erstmaligem Aufruf durch den Nutzer direkt zu Phase 6 springt, wo die JSP aus Abb. 7.3b sowohl den Komponentenbaum mit initialem (leerem) Zustand als auch den HTML-Code erzeugt. Der Komponentenbaum wird für die spätere Wiederherstellung gespeichert und der generierte HTML-Code an den Client zurücksendet.

Das Speichern des Komponentenbaums wird in Phase 6 genauer beschrieben. Es ermöglicht dieser ersten Phase in Folgeanfragen des Nutzers, den Komponentenbaum und seinen Zustand, also die im Formular eingetragenen Werte sowie Konverter und Validatoren, anhand der `viewId` zu rekonstruieren und zwischen Anfragen zu erhalten. Zum Zustand einer Komponente gehören auch alle ggf. für sie registrierte Listener, die auf von ihr erzeugte Ereignisse reagieren.

Der rekonstruierte Komponentenbaum sowie die Anfrage selbst[42] werden im `FacesContext` gespeichert. Der `FacesContext` wird für jede Anfrage erzeugt und besitzt dieselbe Lebensdauer. Auf diese Weise werden anfragespezifische Daten wie der Komponentenbaum an Folgephasen weitergereicht.

Nach Abschluss der ersten Phase enthält der `FacesContext`:

- Komponentenbaum (wiederhergestellt)
- Anfrage

Phase 2: Aktualisierung der Komponentenzustände (Apply Request Values)

Nach Ablauf der ersten Phase befinden sich alle für die aktuelle Anfrage relevanten JSF-Komponenten im `FacesContext`. Im nächsten Schritt muss deren Zustand erneuert werden, d. h. alle Attribute mit Werten aus der HTTP-Anfrage gefüllt werden. Dazu wird der Komponentenbaum traversiert und jede Komponente dazu aufgefordert, sich aus der Anfrage zu bedienen.

In dieser Phase wird lediglich der Zustand der JSF-Komponenten aktualisiert, nicht die Werte der gebundenen Model-Attribute. Komponenten wie Eingabefelder, die editierbare Werte enthalten, besitzen dafür zwei Repräsentationen. Eine enthält den Rohwert

[42] Die Anfrage wird nicht direkt, sondern über eine Fassade `ExternalContext` gespeichert, die JSF eine abstrakte Sicht auf die Umgebung erlaubt. In den meisten Fällen ist die Umgebung ein Servlet-Container und die Anfrage ein entsprechender `HttpServletRequest`.

aus der Anfrage (für HTTP ein String), die andere den konvertierten und validierten Wert. In dieser Phase wird lediglich der Rohwert geändert.

Für das Beispiel der Benutzerregistrierung würden in dieser Phase die Rohwerte der in Abb. 7.11 gezeigten Eingabekomponenten aktualisiert (Zeilennummern aus Abb. 7.3b).

```
14  <h:inputText id="name" value="#{Registration.user.name}"
15               required="true">
16     <f:validateLength minimum="2" maximum="40"/>
17  </h:inputText>

25  <h:inputText id="birthDate" value="#{Registration.user.birthDate}">
26     <f:convertDateTime pattern="dd.MM.yyyy"/>
27  </h:inputText>
```

Abb. 7.11 In Phase 2 des Anfragelebenszyklus aktualisierte JSF-Komponenten

Während der Traversierung des Komponentenbaums können Komponenten Ereignisse erzeugen. Diese werden zunächst lediglich in der Komponente selbst zur späteren Verarbeitung gespeichert. Abbildung 7.12 zeigt die einzige Komponente unseres Beispiels, die ein Ereignis, nämlich einen `ActionEvent`, bei Drücken des Buttons generiert.

```
33  <h:commandButton id="register" action="#{Registration.register}"
34               value="registrieren"/>
```

Abb. 7.12 Eine JSF-Komponente, die ein Ereignis erzeugt.

Jedes Ereignis erthält eine `phaseId`, die angibt, nach welcher Phase (2-5) es ausgelöst wird. In der Regel wird die `phaseId` auf Phase 5 gesetzt, so auch im Beispiel aus Abb. 7.12.

Am Ende dieser Phase enthält der FacesContext:

- Komponentenbaum mit
  - Aktualisierten Rohwerten
  - Erzeugten Ereignissen
- Anfrage

Ereignisverarbeitung

Nach der Beendigung der Phasen 2-5 erfolgt jeweils die Verarbeitung aller in den Komponenten gespeicherten Ereignisse, deren `phaseId` der abgeschlossenen Phase entspricht. Dazu wird der Komponentenbaum traversiert und jede Komponente aufgefordert, die zur `phaseId` passenden Ereignisse an die für sie registrierten Listener zur Verarbeitung zu leiten. Dies ist ein „Seiteneffekt“ der Phasenausführung ([Sun06a], S. 2-10).

Im Beispiel aus Abb. 7.12 registriert die JSF-Implementierung selbst einen `ActionListener` für die Komponente, der die im `action`-Attribut der Komponente angegebene `MethodExpression` auswertet und die so erhaltene Methode `RegistrationController.register()` aufruft.

Phase 3: Validierung (Process Validations)

In der dritten Phase werden die in den Eingabekomponenten gespeicherten Rohwerte aus der Anfrage typkonvertiert und validiert. Dazu wird erneut der Komponentenbaum

aus dem `FacesContext` geladen, traversiert und die Konverter und Validatoren ausgeführt. Jede Eingabekomponente, deren Wert erfolgreich konvertiert und validiert werden konnte, wird als `valid` markiert. Zusätzlich wird neben dem Rohwert der typisierte und geprüfte Wert in der Komponente gespeichert.

Tritt jedoch ein Fehler auf, wird eine entsprechende Fehlermeldung generiert und dem `FacesContext` unter Angabe der `id` der betroffenen Komponente übergeben. Ebenfalls wird dann über den `FacesContext` signalisiert, nach Beendigung dieser Phase die beiden folgenden auszulassen und mit Phase 6 fortzufahren.

Nach Abarbeitung der dritten Phase enthält der `FacesContext`:

- Komponentenbaum mit
  - Aktualisierten Rohwerten
  - Erzeugten Ereignissen
  - Konvertierten und validierten Werten
  - Als `valid` markierten Komponenten
- Anfrage
- Fehlermeldungen (geschlüsselt nach `id` der fehlerhaften Komponente)

Phase 4: Aktualisierung der Model-Objekte (Update Model Values)

In der vierten Phase wird jede `ValueExpression` einer als `valid` markierten Komponente, wie in Abb. 7.8 skizziert, als Schreibzugriff ausgewertet. Danach enthält z. B. das über den `RegistrationController` referenzierte `User`-Objekt die geprüften Werte aus den gebundenen Formularkomponenten.

Der `FacesContext` bleibt i. d. R. unverändert.

Phase 5: Ausführung der Anwendungslogik (Invoke Application)

Die einzige Aufgabe dieser Phase ist lediglich, die Verarbeitung der in den Komponenten verbliebenen Ereignisse wie beschrieben anzustoßen. Auch hier ändert sich im `FacesContext` nichts.

Phase 6: Erzeugung der Antwort (Render Response)

In der letzten Phase wird die zum View gehörende JSP ausgeführt. Die in ihr enthaltenen Tag-Extensions der visuellen JSF-Komponenten fordern diese auf, sich darzustellen, also HTML-Code zu erzeugen.

Exemplarisch seien hier die durch die Tag-Extension `<h:message/>` repräsentierte JSF-Komponente und der zugehörige HTML-Code in Abb. 7.13a und 7.13b gezeigt. Diese Komponente wird nur dargestellt, wenn im `FacesContext` eine entsprechende Fehlermeldung aus Phase 3 mit dem im `from`-Attribut gesetzten Schlüssel existiert. In Abb. 7.13c ist die Fehlermeldung im Browser dargestellt

```
29  <h:message style="color: red" id="birthDateError" for="birthDate"/>
```

Abb. 7.13a Tag-Extension einer JSF-Komponente für die Anzeige einer Fehlermeldung

```
<span id="userForm:birthDateError" style="color: red">invalid</span>
```

Abb. 7.13b HTML-Code der dargestellten Komponente aus Abb. 7.13a

Abb. 7.13c Darstellung der Fehlermeldung aus Abb. 7.13b im Browser

Als letzte Aufgabe muss der Zustand des Komponentenbaums gespeichert werden, damit er in Folgeanfragen in Phase 1 wiederhergestellt werden kann. Dafür definiert JSF zwei Mechanismen: die Speicherung auf dem Client und die Speicherung auf dem Server.

Findet die Speicherung auf dem Client statt, wird der komplette Zustand des Komponentenbaumes serialisiert und mit der HTTP-Antwort an den Client zurückgegeben. Dazu wird der Komponentenzustand in einem versteckten HTML-Formularfeld abgelegt. Bei einer Folgeanfrage sendet der Client diesen mit den restlichen Formularfeldern mit.

Die Speicherung auf dem Server nutzt den Sitzungsspeicher des Nutzers zum Ablegen des Zustandes geschlüsselt über die `viewId`.

7.3 Metamodelle

Wie bei der Generierung der Entity Beans wurden die nötigen Metamodelle mit EMF erstellt und entsprechende Modelleditoren generiert. Neu sind die Metamodelle für Anfragen und XML.

7.3.1 Anfrage-Metamodell

Die Modelle, die gemäß dem Metamodell in Abb. 7.14 Anfragen an das System beschreiben, dienen einerseits der Erstellung der Model- und Controller-Klassen, andererseits wird daraus die Konfigurationsdatei erzeugt, die JSF benötigt, um diese Klassen miteinander zu verdrahten und deren Lebenszyklus zu kontrollieren.

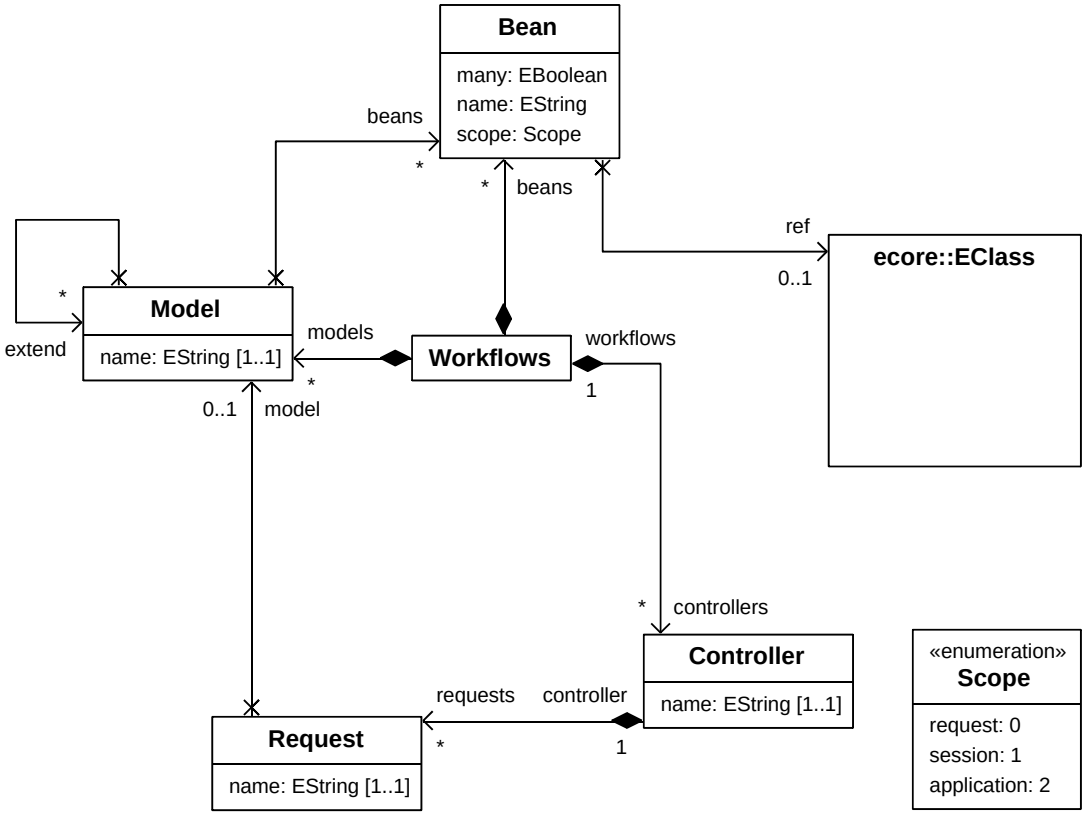


Abb. 7.14 Metamodell der Anfrageverarbeitung

Jede Anfrage wird durch ein `Request`-Objekt repräsentiert und von einem `Controller` entgegengenommen. Weiterhin ist jeder `Request` mit einem `Model` verbunden. Ein `Model` ist dabei von mehreren `Requests` nutzbar. Ein `Model` fasst Objekte aus dem Ecore-Modell zusammen, welche aber zunächst durch ein `Bean`-Objekt dekoriert werden, das vor allem zur Festlegung der Lebensdauer dient. Neben der Lebensdauer (`scope`) ist dort auch die Multiplizität (`many`), ob das Model ein oder mehrere Objekte des referenzierten Ecore-Elements enthält, festgelegt.

Das `Workflows`-Objekt dient lediglich als Wurzelelement im Modell und erleichtert die Navigation im Quellmodell, was schlankere und übersichtlichere ATL-Regeln ermöglicht.

Das Metamodell sieht keine Möglichkeit vor, die Verknüpfung zwischen Service-Objekt und Controller festzulegen. Dies erfolgt implizit, indem für jeden `Controller` gemäß Namenskonvention ein analoges `Service`-Interface generiert wird, das pro `Request` eine für dessen Verarbeitung vorgesehene Methode enthält. Die Implementierung dieser Schnittstelle obliegt dann dem Entwickler.

7.3.2 XML-Metamodell

Das XML-Metamodell (Abb. 7.15) spiegelt nicht die für XML charakteristische Baumstruktur wider, sondern ist allgemeiner, da ein Kindknoten mehrere Elternknoten besitzen kann. Dies ist eine Konzession an die Arbeitsweise von ATL, da eine strenge Baumstruktur entweder Redundanzen im Modell oder in den ATL-Transformationen bedeutet hätte. Das Metamodell ist also weitgehend geprägt von dessen Einsatzzweck im Transformationsprozess. Dies ist dadurch gerechtfertigt, dass die XML-Modelle lediglich Zwischenprodukte bilden.

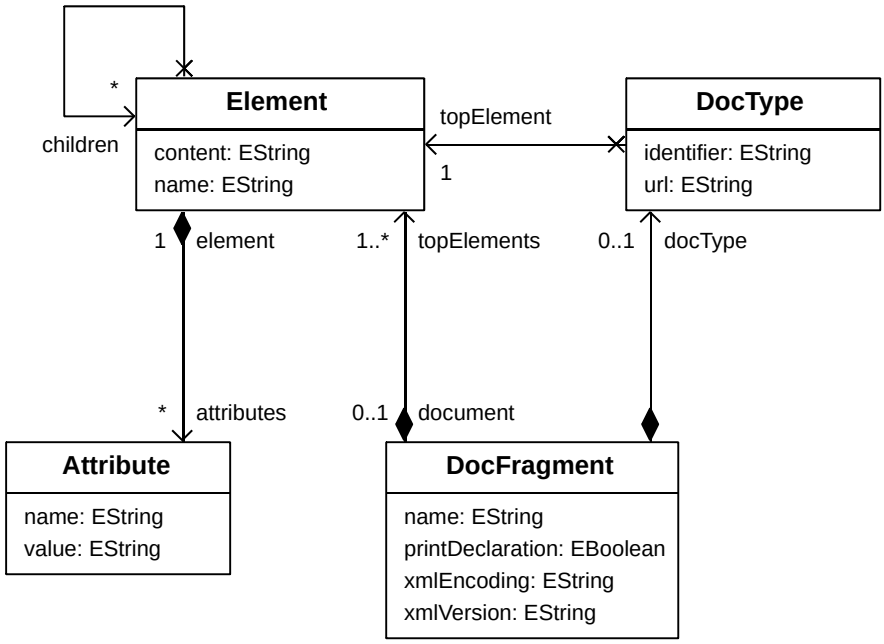


Abb. 7.15 XML-Metamodell

7.4 ATL-Transformationen

Anders als im Falle von Entity Beans (Kapitel 6), in dem alle Transformationsschritte in Java-Quellcode münden, benötigt JSF zusätzlich eine XML-Konfigurationsdatei (`faces-config.xml`). Die Transformationen zu deren Erzeugung werden nach den Transformationen zur Generierung der Model- und Controller-Klassen vorgestellt.

7.4.1 ATL-Transformationen zur Generierung der Model- und Controller-Klassen

Bevor die ATL-Transformationen zur Generierung der Klassen untersucht werden, soll die gewählte Paketstruktur ohne den sie erzeugenden ATL-Code erwähnt werden. Ausgehend von einem beliebigen Wurzelpaket, erhalten Model- und Controller-Klassen sowie die Service-Schnittstellen die jeweiligen Unterpakete `model`, `controller` und `service`. Zur Erzeugung dieser Paketstruktur dient das Wurzelelement des Anfrage-Modells (`Workflows`, vgl. Abb. 7.14) als Quellmuster einer ATL-Regel.

Controller-Generierung

Abbildung 7.16 zeigt den ATL-Code zur Generierung der Controller. Die generierte Controller-Klasse besitzt für jede Anfrage (`Request`, vgl. Abb. 7.14) ein Attribut, um deren Model aufzunehmen, und eine Methode, welche die Anfrage an die Methode des Service-Objekts delegiert, das ebenfalls über ein Attribut referenziert wird. Für alle Attribute werden Zugriffsmethoden generiert, damit JSF die entsprechenden Objekte gemäß Vorgaben der `faces-config.xml` verdrahten kann, also deren Attribute setzen kann. Die Regelteile zur Erzeugung der Zugriffsmethoden für die Attribute fehlen in Abb. 7.16 zur Erhöhung der Leserlichkeit.

Der Controller wird durch zwei Schritte erzeugt. Der erste (ATL-Regel `Controller2Class`) generiert für jedes `Controller`-Objekt des Anfrage-Quellmodells im Java-Zielmodell eine Controller-Klasse (Zeilen 5-7), erzeugt darin ein Attribut zur Aufnahme der Referenz auf das Service-Objekt (Zeilen 16-21) und generiert die Service-Schnittstelle (noch ohne Methoden), über die diese Referenz erfolgt (Zeilen 9-14). Die Namen der Controller-Klassen und der Service-Schnittstelle werden aus dem Namen des Controllers im Quellmodell gemäß den String-Verkettungen in Zeile 6 bzw. Zeile 10 gebildet. Für einen Controller mit Namen `Login` entstünde also die Java-Klasse `LoginController` und die Java-Schnittstelle `ILoginService`.

Im zweiten Schritt (ATL-Regel `Request2Field`) entstehen für jeden dem Controller zugewiesenen `Request` die Attribute zur Aufnahme der Model-Objekte (Zeilen 29-34), die Methoden der Service-Schnittstelle (Zeilen 50-67) sowie die zugehörigen Weiterleitungsmethoden im Controller (Zeilen 36-48).

```
01 rule Controller2Class {
02     from
03             ic : workflow!Controller
04     to
05             controller : java!JavaClass (
06                     name <- ic.name + 'Controller'
07             ),
08             -- SPI
09             spi : java!JavaClass (
10                     name <- 'I' + ic.name + 'Service',
11                     javaPackage <- thisModule.resolveTemp( ic.workflows,
12 'service' ),
13                     kind <- #INTERFACE
14             ),
15             -- reference to SPI in controller
16             spiField : java!Field (
17                     name <- ic.name.firstToLower() + 'Service',
18                     type <- spi,
19                     javaClass <- ic,
20                     javaVisibility <- #PRIVATE
21             ),
   […]
22 }
23
24 rule Request2Field {
25     from
26             request : workflow!Request
27     to
28             -- controller field to store model
29             controllerField : java!Field (
30                     name <- request.name + '_' + request.model.javaName(),
31                     type <- request.model,
32                     javaClass <- request.controller,
33                     javaVisibility <- #PRIVATE
34             ),
35             -- controller method calling SPI method
36             controllerMethod : java!Method (
37                     name <- request.name,
38                     javaClass <- request.controller,
39                     returnType <- thisModule.resolveTemp(
40 request.controller.workflows, 'string'),
41                     source <- controllerMethodSource
42             ),
43             controllerMethodSource : java!Block (
44                     source <- '{return '
45 + request.controller.name.firstToLower() + 'Service.' + request.name + '( new '
46 + 'org.hardan.da.view.impl.FacesCallerContextImpl(), ' + request.name + '_'
47 + request.model.javaName() + ' );}'
48             ),
49             -- SPI method
50             spiMethod : java!Method (
51                     name <- request.name,
52                     returnType <- thisModule.resolveTemp(
53 request.controller.workflows, 'string'),
54                     abstract_ <- true,
55                     javaClass <- thisModule.resolveTemp(request.controller, 'spi')
56             ),
57             spiMethodCallerParam : java!JavaParameter (
58                     name <- 'caller',
59                     type <- thisModule.resolveTemp(request.controller.workflows,
60 'callerContext'),
61                     method <- spiMethod
62             ),
63             spiMethodModelParam : java!JavaParameter (
64                     name <- 'model',
65                     type <- request.model,
66                     method <- spiMethod
67             )
   […]
68 }
```

Abb. 7.16 ATL-Regeln zur Erzeugung der Java-Klasse eines Controllers (Ausschnitt)

Model-Generierung

Bei der Generierung der Model-Klassen fordert ATL Tribut vom Java-Metamodell. Das Anfrage-Metamodell erlaubt es, `Bean`-Instanzen, bzw. die von ihnen dekorierten `EClass`-Instanzen aus dem Ecore-Modell, in mehreren `Model`-Instanzen wiederzuverwenden. Demnach weisen verschiedene Java-Model-Klassen dieselben Attribute und zugehörige Zugriffsmethoden auf, sofern die `Model`-Instanzen im Quellmodell dieselben `Bean`-Instanzen referenzieren. Attribute und Methoden gehören im Java-Metamodell jedoch stets zu genau einer Klasse und lassen sich nicht wiederverwenden. Leider bietet ATL bei der Transformation vom Quell- ins Zielmodell hierzu keine Lösung, weshalb die Metaklasse `JavaClass` im Java-Metamodell um das Konzept von gemeinsam genutzten Attributen und Methoden (`sharedFields` bzw. `sharedMethods`) erweitert werden musste.

In den ATL-Regeln in Abb. 7.17 ist dieser Lösungsansatz über gemeinsam genutzte Attribute und Methoden erkennbar. Die erste Regel (Zeilen 1-22) erzeugt für jede `Bean`-Instanz im Quellmodell Attribute und Zugriffsmethoden (Methoden für schreibenden Zugriff fehlen zugunsten der Übersichtlichkeit). Dabei wird kein Bezug zu einer Java-Klasse hergestellt. Dies erfolgt erst durch die zweite, die Model-Klasse erzeugende, ATL-Regel (Zeilen 24-36) über `sharedFields` (Zeile 30) und `sharedMethods` (Zeilen 31-34). So können beliebig viele Model-Klassen die nur einmal von der Regel `Bean2Field` erzeugten Attribute und Methoden einbinden.

```
01 rule Bean2Field {
02     from
03             bean : workflow!Bean
04     to
05             field : java!Field (
06                     name <- bean.name,
07                     type <- if bean.many = false then bean.realRef()
08 else workflow!Workflows.allInstances()->collect(
09 w | thisModule.resolveTemp( w, 'listDataModel' ))->first() endif,
10                     javaVisibility <- #PRIVATE
11             ),
12             getter : java!Method (
13                     name <- 'get' + bean.name.firstToUpper(),
14                     returnType <- if bean.many = false then bean.realRef()
15 else workflow!Workflows.allInstances()->collect(
16 w | thisModule.resolveTemp( w, 'listDataModel' ))->first() endif,
17                     source <- getterImpl
18             ),
19             getterImpl : java!Block (
20                     source <- bean.name.javaGetterImpl()
21             ),
   […]
22 }
23
24 rule Model2Class {
25     from
26             model : workflow!Model
27     to
28             clazz : java!JavaClass (
29                     name <- model.name + 'Model',
   […]
30                     sharedFields <- model.beans,
31                     sharedMethods <- model.beans->collect(
32 b | thisModule.resolveTemp( b, 'getter' ) )->union(
33 model.beans->collect( b | thisModule.resolveTemp( b, 'setter' ) )
34                     )
35             )
36 }
```

Abb. 7.17 ATL-Regeln zur Erzeugung der Java-Model-Klasse (Ausschnitt)

Die Java-Klassen für die mit den `Bean`-Instanzen verknüpften `EClass`-Instanzen des Ecore-Modells müssen nicht mehr generiert werden. Dies erfolgte bereits im Zuge der Entity-Bean-Erzeugung (vgl. Kapitel 6.4.2).

Die ATL-Transformation zur Generierung des Java-Quellcodes für Model und Controller ist bis auf die Unterstützung für gemeinsame Attribute und Methoden identisch und wurde entsprechend erweitert.

7.4.2 ATL-Transformationen zur Generierung der JSF-Konfiguration

Anfrage- und Ecore-Modell dienen nicht nur der Generierung benötigter Java-Klassen, sondern auch zur Erzeugung der JSF-Konfiguration, genauer des Teils, der für die Verdrahtung der Objekte und die Steuerung von deren Lebenszyklus verantwortlich ist.

Die JSF-Konfiguration besteht aus einer Menge von Managed-Beans, die mit Hilfe der EL miteinander verknüpft werden. Das Beispiel in Abb. 7.18 zeigt die generierte Deklaration für `LoginController`, `LoginModel` und `User`. Die Lebensdauer der Objekte bestimmt das Element `managed-bean-scope`. Model- und Controller Objekte werden für jede HTTP-Anfrage neu von JSF erzeugt, das `User`-Objekt existiert für die Dauer der Nutzersitzung. Mit Hilfe der EL werden über das Element `managed-property` die Objekte miteinander verdrahtet. Eine `managed-property` sagt JSF, dass ein Attribut mit Zugriffsmethode gemäß JavaBean-Konvention existiert, die das Setzen eines Wertes erlaubt. So besitzt z. B. die Klasse `LoginModel` ein Attribut `user`, das über die Methode `setUser` gesetzt werden kann. Den zu setzenden Wert ermittelt JSF durch Auswertung des EL-Ausdrucks im `value`-Element (zur EL-Auswertung vgl. Abb. 7.8). Der Wert, den JSF über `setUser()` im `LoginModel`-Objekt setzt, ist also das `User`-Objekt aus dem Sitzungskontext des Nutzers.

```
<managed-bean>
  <managed-bean-name>loginController</managed-bean-name>
  <managed-bean-class>de.hardan.da.controller.LoginController</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>loginService</property-name>
    <value>#{loginService}</value>
  </managed-property>
  <managed-property>
    <property-name>loginUser_LoginModel</property-name>
    <value>#{loginModel}</value>
  </managed-property>
[…]
</managed-bean>

<managed-bean>
  <managed-bean-name>loginModel</managed-bean-name>
  <managed-bean-class>de.hardan.da.model.LoginModel</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>user</property-name>
    <value>#{user}</value>
  </managed-property>
  <managed-property>
[…]
</managed-bean>

<managed-bean>
  <managed-bean-name>user</managed-bean-name>
  <managed-bean-class>de.hardan.da.beans.User</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Abb. 7.18 Ausschnit der generierten JSF-Konfiguration

Generierung der Controller-Konfiguration

Aufgabe der ATL-Transformationen aus Abb. 7.19 ist es, aus dem Anfragemodell die `managed-bean`-Einträge für die Controller mit den `managed-property`-Verknüpfungen zu den Service- und Model-Objekten zu erzeugen.

Auch hier erfolgt die Transformation in zwei Schritten. Die erste ATL-Regel (Zeilen 1-35) erzeugt das `managed-bean`-Element im XML-Zielmodell sowie die `managed-property` zur Verknüpfung mit dem Service-Objekt. Die `managed-property`-Elemente zur Aufnahme der Model-Objekte fügt die zweite Regel (Zeilen 37-53) hinzu.

```
01 rule Controller2Element {
02     from
03             ic : workflow!Controller
04     to
05             bean : xml!Element (
06                     name <- 'managed-bean',
07                     children <- Sequence{name, clazz, scope, service}->union(
08 ic.requests )
09             ),
10             name : xml!Element (
11                     name <- 'managed-bean-name',
12                     content <- ic.name.firstToLower() + 'Controller'
13             ),
14             clazz : xml!Element (
15                     name <- 'managed-bean-class',
16                     content <- 'controller.'.javaRootPkg() + ic.name +'Controller'
17             ),
18             scope : xml!Element (
19                     name <- 'managed-bean-scope',
20                     content <- 'request'
21             ),
22             -- property for service
23             service : xml!Element (
24                     name <- 'managed-property',
25                     children <- Sequence{propName, propValue}
26             ),
27             propName : xml!Element (
28                     name <- 'property-name',
29                     content <- ic.name.firstToLower() + 'Service'
30             ),
31             propValue : xml!Element (
32                     name <- 'value',
33                     content <- '#{' + ic.name.firstToLower() + 'Service'+ '}'
34             )
35 }
36
37 rule Request2Element {
38     from
39             ir : workflow!Request
40     to
41             prop : xml!Element (
42                     name <- 'managed-property',
43                     children <- Sequence{propName, propValue}
44             ),
45             propName : xml!Element (
46                     name <- 'property-name',
47                     content <- ir.name + '_' + ir.model.javaName()
48             ),
49             propValue : xml!Element (
50                     name <- 'value',
51                     content <- '#{' + ir.model.beanName() + '}'
52             )
53 }
```

Abb. 7.19 ATL-Regeln zur Erzeugung der Controller-Konfiguration für JSF

Da die Implementierung der eigentlichen Anfrageverarbeitung in den Service-Objekten dem Entwickler überlassen bleibt, müssen diese Implementierungen JSF manuell bekannt gemacht werden. Dies erfolgt in einer separaten XML-Konfigurationsdatei mit identischer Struktur, in der die Managed-Beans für die Service-Implementierungen angegeben werden. Dabei müssen sie der Namenskonvention gemäß Zeile 29 folgen, damit JSF den generierten EL-Ausdruck korrekt auflösen kann. Die Managed-Bean der zum `LoginController` gehörenden Service-Implementierung muss demnach den Namen `loginService` tragen (Abb. 7.20).

```
<managed-bean>
  <managed-bean-name>loginService</managed-bean-name>
  <managed-bean-class>de.hardan.da.service.impl.LoginServiceImpl
  </managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
[…]
</managed-bean>
```

Abb. 7.20 Ausschnit der manuellen JSF-Konfiguration

Generierung der Model-Konfiguration

Hauptaufgabe der ATL-Transformationen zur Generierung der Model-Konfiguration ist es, die Abhängigkeiten der Klassen aus dem Ecore-Modell zu einem `Model` des Anfragemodells mit dem `managed-property`-Mechanismus zu beschreiben (Abb. 7.21). Jede `Bean` im Anfragemodell führt zu einem `managed-bean`-Element, das die dekorierte `EClass` für die Nutzung durch JSF registriert (vgl. `user` in Abb. 7.18). Danach werden diese mittels `managed-property` in das zugehörige `Model` eingebunden (vgl. `loginModel` in Abb. 7.18).

Der Umstand, dass ein Bean von mehreren Model-Instanzen genutzt werden kann, ist auch hier der Grund für das ungewöhnliche XML-Metamodell, das keine Baumstruktur fordert. So erzeugt die erste Regel (Zeilen 1-35) für jede von einer `Bean` referenzierten `EClass` den entsprechenden `managed-bean`-Eintrag und auch gleich den `managed-property`-Eintrag (Zeilen 23-34) für das `Model`. Letzterer wandert erst durch die zweite Regel (Teilen 37-58) an seinen Platz, da dort das `managed-bean`-Element für das `Model` entsteht.

```
01 rule Bean2Element {
02     from
03             ib : workflow!Bean
04     to
05             bean : xml!Element (
06                     name <- 'managed-bean',
07                     children <- Sequence{ name, clazz, scope }
08             ),
09             name : xml!Element (
10                     name <- 'managed-bean-name',
11                     content <- ib.name.firstToLower()
12             ),
13             clazz : xml!Element (
14                     name <- 'managed-bean-class',
15                     content <- ib.realRef().toFQN()
16             ),
17             scope : xml!Element (
18                     name <- 'managed-bean-scope',
19                     content <- if not ib.scope.oclIsUndefined()
20 then ib.scope.toString() else 'request' endif
21             ),
22             -- property in model
23             prop : xml!Element (
24                     name <- 'managed-property',
25                     children <- Sequence{ propName, value }
26             ),
27             propName : xml!Element (
28                     name <- 'property-name',
29                     content <- ib.name.firstToLower()
30             ),
31             value : xml!Element (
32                     name <- 'value',
33                     content <- '#{' + ib.name.firstToLower() + '}'
34             )
35 }
36
37 rule Model2Element {
38     from
39             im : workflow!Model
40     to
41             bean : xml!Element (
42                     name <- 'managed-bean',
43                     children <- Sequence{ name, clazz, scope }->union(
44 im.beans->collect( b | thisModule.resolveTemp(b, 'prop') ) )
45             ),
46             name : xml!Element (
47                     name <- 'managed-bean-name',
48                     content <- im.beanName()
49             ),
50             clazz : xml!Element (
51                     name <- 'managed-bean-class',
52                     content <- 'de.hardan.da.model.' + im.javaName()
53             ),
54             scope : xml!Element (
55                     name <- 'managed-bean-scope',
56                     content <- 'request'
57             )
58 }
```

Abb. 7.21 ATL-Regeln zur Erzeugung der Model-Konfiguration für JSF

8 Die lauffähige Anwendung des Fallbeispiels

Die Betrachtung der generierten und nicht generierten Artefakte in der laufenden Anwendung steht im Mittelpunkt dieses Kapitels. Am Beispiel der Gebotsabgabe für einen Spieler am Transfermarkt (vgl. Kapitel 2.2.3) wird das Zusammenspiel der beteiligten Bausteine von der Annahme der HTTP-Anfrage bis zum Speichern in der Datenbank untersucht.

Ausgangspunkt ist ein angemeldeter Nutzer, der bereits einer Miniliga beigetreten ist und nun seinen Kader mit Spielern füllen möchte. Dazu lässt er sich alle Offerten am Transfermarkt anzeigen (Abb. 8.1) und wählt die erste aus, woraufhin das System ihn auffordert, den Gebotsbetrag einzugeben (Abb. 8.2).

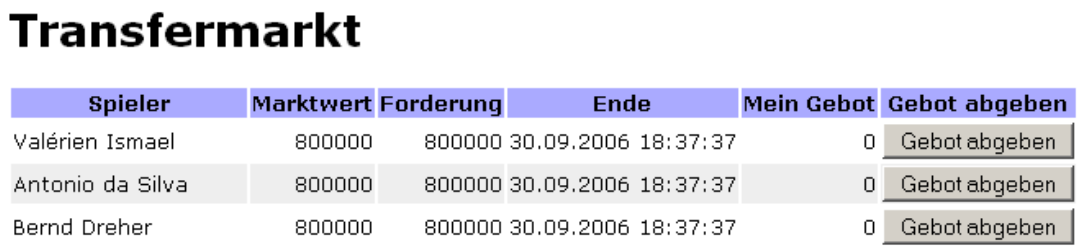


Abb. 8.1 Liste der Offerten auf dem Transfermarkt

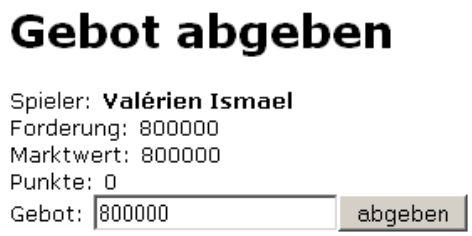


Abb. 8.2 Gebotsabgabe zu einer Offerte

8.1 An der Gebotsabgabe beteiligte Modellausschnitte

Abbildungen 8.3a-c zeigen die für den vorliegenden Anwendungsfall relevanten Modellausschnitte, wie sie die Editoren, die von EMF zuvor generiert wurden (vgl. Kapitel 4.3), darstellen. Die modellierten Entity Beans aus Abb. 8.3a werden mit Persistenzinformationen aus Abb. 8.3b dekoriert, indem dort jeder Klasse aus Abb. 8.3a eine Datenbanktabelle zugewiesen wird (Attribut `mappedClass`, vgl. Kapitel 6.3.1). Die Klasse `UserTeam` bezeichnet die Mannschaft des Nutzers, `Offer` eine Offerte und `Bid` ein Gebot.

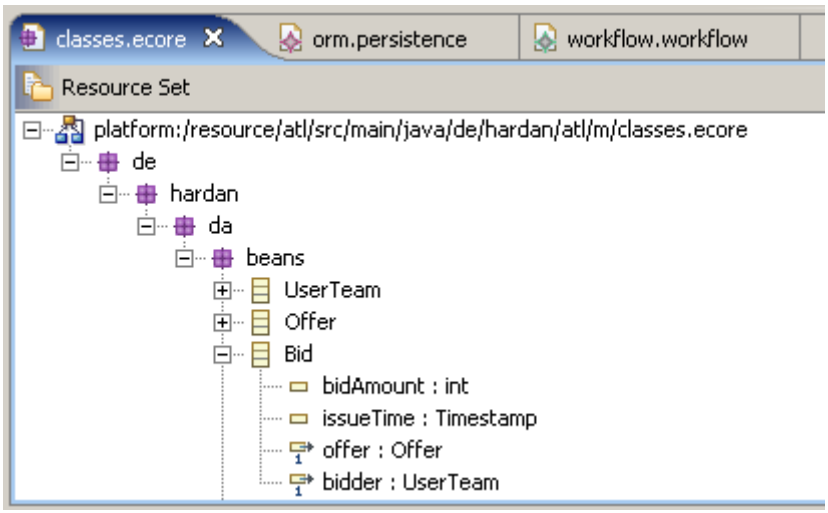


Abb. 8.3a Für die Gebotsabgabe relevanter Ausschnitt aus dem Klassenmodell

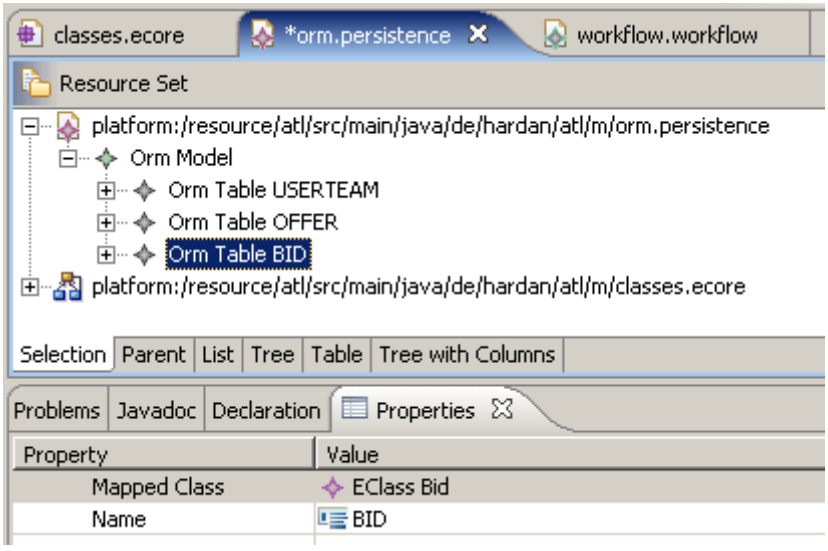


Abb. 8.3b Für die Gebotsabgabe relevanter Ausschnitt aus dem Persistenzmodell

Das Anfragemodell aus Abb. 8.3c greift ebenfalls auf das Klassenmodell zurück. Das von allen Anfragen (`listOffers`, `showOfferDetails`, `placeBid`) genutzte Model `Transfer` referenziert die Klassen `Offer`, `Bid` und `UserTeam` über die verschiedenen `Bean`-Elemente. Dabei dient die `Bean offers` mit gesetztem `many`-Attribut (vgl. Kapitel 7.3.1) zur Anzeige aller Offerten einer Miniliga (Abb. 8.1) und `offer` zum Festhalten der ausgewählten Offerte (Abb. 8.2).

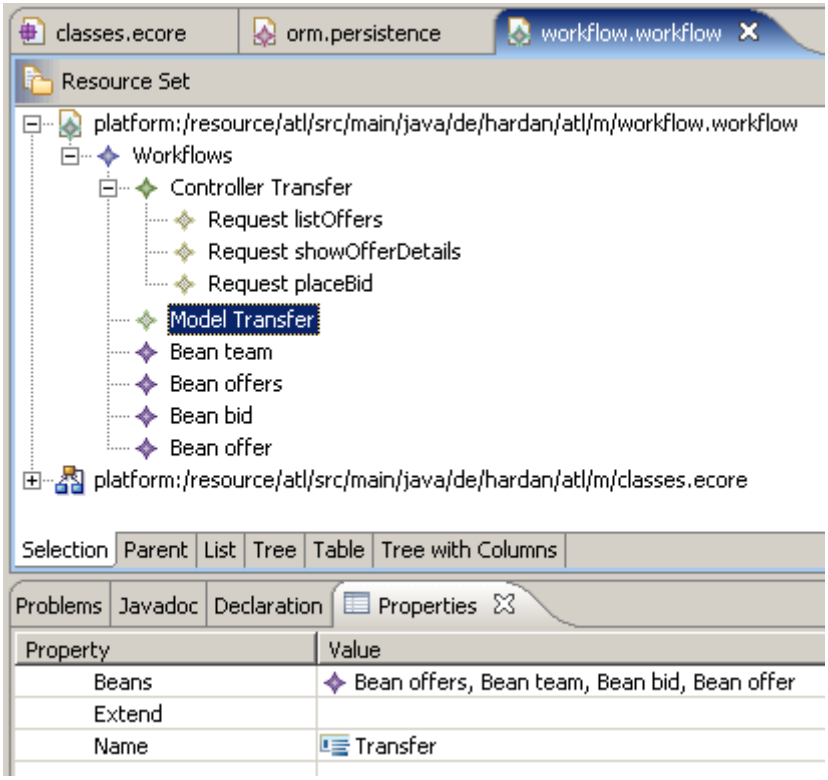


Abb. 8.3c Für die Gebotsabgabe relevanter Ausschnitt aus dem Anfragemodell

8.2 An der Gebotsabgabe beteiligte Laufzeitkomponenten

Ausgehend vom Formular für die Gebotsabgabe (Abb. 8.2) erfolgt nun die Betrachtung sämtlicher Arbeitsschritte der Anwendung bis zur Speicherung des Gebots in der Datenbank (vgl. Kapitel 5). Der JSP-Code aus Abb. 8.4 bringt das HTML-Formular aus Abb. 8.2 hervor. Die `ValueExpressions` (vgl. Kapitel 7.2) in den Zeilen 3-6 werden nur lesend verwendet und liefern Daten wie Spielername, Marktwert usw. Die `ValueExpression` aus Zeile 7 bindet das Attribut `bidAmount` des `Bid`-Objekts an das Eingabefeld und dient lesendem und schreibendem Zugriff. Schließlich sorgt die `MethodExpression` in Zeile 8 für die Verbindung zum Controller.

```
01 <h:form id="placeBidForm">
02     <h1>Gebot abgeben</h1>
03     Spieler: <b><h:outputText value="#{offer.player.name}"/></b><br/>
04     Forderung: <h:outputText value="#{offer.claim}"/><br/>
05     Marktwert: <h:outputText value="#{offer.player.marketValue}"/><br/>
06     Punkte: <h:outputText value="#{offer.player.points}"/><br/>
07     Gebot: <h:inputText id="bid" value="#{bid.bidAmount}"/>
08     <h:commandButton action="#{transferController.placeBid}" value="abgeben"/>
09     <br/><h:message id="bidError" for="bid" showDetail="true" />
10 </h:form>
```

Abb. 8.4 JSP-Code zum Formular aus Abb. 8.2

Wie im Beispiel aus Kapitel 7.2.1 beginnt auch die Auswertung dieser EL-Ausdrücke mit dem `ManagedBean ELResolver`. Dafür müssen jeweils Java-Klassen unter den Namen `offer`, `bid` und `transferController` bei JSF registriert sein, was Aufgabe der Konfigurationsdatei `faces-config.xml` ist. Abbildung 8.5 zeigt den entsprechenden Ausschnitt und darüber hinaus die Deklaration des `transferModels` (Zeilen 15-31), das der Controller an das Service-Objekt übergibt (vgl. Kapitel 5). Die Konfigurationsdatei ist das Transformationsergebnis vom Modell aus Abb. 8.3c.

Bevor dies geschieht, erreicht die vom Browser nach Drücken des „abgeben“-Knopfes (Abb. 8.2) gesendete HTTP-Anfrage zunächst den Web-Container des Servers. Damit JSF die Anfrage entgegennehmen kann, ist das `FacesServlet` beim Web-Container registriert (vgl. Kapitel 3.5). Es leitet den in Kapitel 7.2.2 beschriebenen Lebenszyklus ein, der in der Ereignisverarbeitung am Ende von Phase 5 für den Aufruf des `transferControllers` sorgt, ausgelöst durch den zum Knopfdruck gehörigen `ActionEvent`.

```
<managed-bean>
    <managed-bean-name>transferController</managed-bean-name>
    <managed-bean-class>de.hardan.da.controller.TransferController</managed-bean-
class>
    <managed-bean-scope>request</managed-bean-scope>
    <managed-property>
           <property-name>transferService</property-name>
           <value>#{transferService}</value>
    </managed-property>
    <managed-property>
           <property-name>placeBid_TransferModel</property-name>
           <value>#{transferModel}</value>
    </managed-property>
[…]
</managed-bean>

<managed-bean>
    <managed-bean-name>transferModel</managed-bean-name>
    <managed-bean-class>de.hardan.da.model.TransferModel</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
[…]
    <managed-property>
           <property-name>team</property-name>
           <value>#{team}</value>
    </managed-property>
    <managed-property>
           <property-name>bid</property-name>
           <value>#{bid}</value>
    </managed-property>
    <managed-property>
           <property-name>offer</property-name>
           <value>#{offer}</value>
    </managed-property>
</managed-bean>

[…]
<managed-bean>
    <managed-bean-name>team</managed-bean-name>
    <managed-bean-class>de.hardan.da.beans.UserTeam</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
<managed-bean>
    <managed-bean-name>bid</managed-bean-name>
    <managed-bean-class>de.hardan.da.beans.Bid</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
<managed-bean>
    <managed-bean-name>offer</managed-bean-name>
    <managed-bean-class>de.hardan.da.beans.Offer</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Abb. 8.5 Ausschnitt der JSF-Konfiguration zur Gebotsabgabe

Dem EL-Ausdruck aus Abb. 8.4, Zeile 8 folgend, ruft JSF die Methode `placeBid()` des generierten `TransferControllers` auf, die den Aufruf wie in Kapitel 5 skizziert an das zugeordnete Service-Objekt delegiert. (Abb. 8.6).

```
public class TransferController {
	private de.hardan.da.service.ITransferService transferService;
	private de.hardan.da.model.TransferModel placeBid_TransferModel;
[…]
	public java.lang.String placeBid() {
		return transferService.placeBid( new
org.hardan.da.view.impl.FacesCallerContextImpl(), placeBid_TransferModel );
	}
}
```

Abb. 8.6 Ausschnitt des TransferControllers

Die vom Entwickler erstellte Implementierung der `ITransferService`-Schnittstelle in Abb. 8.7 sorgt für die Speicherung des Gebots in der Datenbank über die JPA. Zunächst wird die ausgewählte Offerte über das Model-Objekt aus dem Sitzungsspeicher (vgl. Abb. 8.4, Zeile 46) geholt und über eine Datenabfrage ermittelt, ob der Nutzer für diese Offerte bereits ein Gebot abgegeben hat. Wenn nicht, werden die Attribute des Gebots gesetzt, anderenfalls lediglich der Gebotsbetrag angepasst. Zuletzt erfolgt die Speicherung in der Datenbank.

```
public class TransferServiceImpl implements ITransferService {

	public String placeBid( CallerContext caller, TransferModel model ) {
		Offer offer = model.getOffer();
		emf = Persistence.createEntityManagerFactory( "default", new
java.util.HashMap() );
		EntityManager em = emf.createEntityManager();

		Query query = em.createNamedQuery( "BidForOfferAndTeam" );
		query.setParameter( "bidder", model.getTeam() );
		query.setParameter( "offer", offer );
		Bid bid = new Bid();
		try {
			bid = (Bid) query.getSingleResult();
		} catch (NoResultException e) {
			//no bid placed before for this offer
			//keeping created Bid instance
			bid.setIssueTime( DateHelper.timestamp() );
			bid.setOffer( offer );
			bid.setBidder( model.getTeam() );
		}
		bid.setBidAmount( model.getBid().getBidAmount() );

		em.getTransaction().begin();
		em.persist( bid );
		em.getTransaction().commit();
		em.close();
		emf.close();

		return listOffers( caller, model );
	}

}
```

Abb. 8.7 Ausschnitt der Implementierung der ITransferService-Schnittstelle

Abgesehen von JSP-Code (Abb. 8.4) und Service-Implementierung (Abb. 8.7) sind sämtliche Artefakte, Entity Beans wie Offer oder Bid, Model- und Controller-Klassen sowie JSF-Konfiguration, aus Modellen (Abb. 8.3a-c) generiert.

9 Schluss

Diese Arbeit zeigt, dass modellgetriebene Softwareentwicklung im Sinne der einleitenden Beschreibung mit integrierten Entwicklungswerkzeugen auch für eine komplexe Anwendung und Zielplattform umsetzbar ist. Zwar verlief nicht die Erstellung aller Softwareteile modellgetrieben, jedoch konnte ein großer Bereich aus redundanzfreien, in problemadäquaten Sprachen formulierten Modellen generiert werden. Die größte Herausforderung stellten dabei die Modelltransformationen dar, da sie Modelle, Metamodelle und Artefakte der Zielplattform miteinander verbinden müssen.

9.1 Bewertung der eingesetzten Techniken und Werkzeuge

Eclipse hat sich als Entwicklungsplattform zur Integration der Programmierungs-, Modellierungs- und Transformationswerkzeuge bewährt. Die Verwaltung sämtlicher Artefakte war wegen des ausgereiften Workspace-Konzepts komfortabel.

Die Beschreibung der verschiedenen Metamodelle mit dem Ecore-Editor und die Generierung von und Arbeit mit entsprechenden Editoren waren ebenfalls problemlos möglich. Ecore besitzt als Meta-Metamodell genug Ausdruckskraft zur Spezifikation der abstrakten Syntax der hier eingesetzten Modellierungssprachen.

Nicht zuletzt glänzen sowohl Eclipse im Allgemeinen als auch EMF im Besonderen mit einer guten Dokumentation.

Lediglich ATL offenbart sowohl bei den Sprachkonzepten als auch bei den Werkzeugen und der Dokumentation Schwächen. Die Codegenerierung wirkt wie eine Notlösung. Entsprechende ATL-Dateien sind unleserlich und schlecht zu warten. Auch im eigentlichen Einsatzgebiet von ATL, den Modell-zu-Modell-Transformationen, zeigt sie Schwächen. So waren Änderungen an Metamodellen nötig, um gewisse Transformationen ohne das Einfügen von Redundanzen in den Modellen überhaupt zu ermöglichen (vgl. Kapitel 7.3.2 und 7.4.1). Die fehlerhafte Handhabung der Identität von Modellelementen erzwang ebenfalls Hilfslösungen (vgl. Kapitel 6.4.2). Die Einschränkung, dass Modellelemente einer Metaklasse von genau einer Regel[43] transformiert werden müssen, führte oft zu umständlichen Formulierungen.

Der als Eclipse-Plug-In angebotene ATL-Editor leistet durch farbliche Unterscheidung syntaktischer Elemente und Markierung von Eingabefehlern gute Dienste. Bei Laufzeitfehlern bieten die ATL-Werkzeuge kaum Unterstützung. Einen Debugger gibt es nicht und die kryptischen Ausgaben der ATL-Konsole während des Ablaufs einer Transformation sind bestenfalls von den Entwicklern zu entziffern und nirgends dokumentiert.

Leider lassen sich mehrere ATL-Transformationen nicht verketten. Jeder Schritt ist vom Entwickler einzeln in der richtigen Reihenfolge auszuführen. Für die Generierung sämtlicher Artefakte werden im Fallbeispiel sechs Transformationen benötigt.

Die Dokumentation zur Transformationssprache und zu mitgelieferten Bibliotheken ist knapp oder nicht vorhanden. Einige Funktionen konnten nur durch Studieren anderer ATL-Dateien entdeckt werden.

---

[43] Zwar können mehrere Regeln zu einer Metaklasse existieren, dann müssen die zugehörigen Modellelemente aber über Prädikate in disjunkte Teilmengen zerlegt werden.

9.2 Ausblick

Die bisher erfolglose Suche nach Standards im Bereich der Modelltransformationssprachen dokumentiert die vielen Probleme, die eine solche Sprache und entsprechende Werkzeuge lösen müssen.

Über die skizzierten Schwächen von ATL hinaus existieren weitere, tieferliegende Probleme beim Einsatz von Modelltransformationen. So bedarf es vor allem einer engeren Integration mit der Entwicklungsumgebung. Konzeptionell unterscheidet sich eine Modelltransformation nicht von der Übersetzung eines Stückes Quellcode in Objektcode durch einen Compiler. Lediglich die Beobachtungsebene ist eine andere. Betrachtet man die Leistungsfähigkeit von Entwicklungsumgebungen wie Eclipse mit JDT für Java, wird die Lücke zu den Modelltransformationen deutlich. Inkrementelle Übersetzung beispielsweise, mit JDT selbstverständlich, liegen für Modelltransformationen noch in weiter Ferne. Jede Modelländerung erfordert die Generierung aller vom gesamten Modell abhängigen Artefakte.

Für die Umsetzung von MDSD stehen weitgehend ausgereifte Konzepte bereit. Im Kern, den Modelltransformationen, liegt das größte Verbesserungspotenzial, das für einen breiten Einsatz erst erschlossen werden muss.

Literaturverzeichnis

[Atla06] ATLAS group (2006): *ATL: Atlas Transformation Language: ATL User Manual*. Version 0.7
URL: http://www.eclipse.org/gmt/atl/doc/ATL_User_Manual%5Bv0.7%5D.doc
Abruf: 28.09.2006

[BaGr05] Bacvanski, V.; Graff, P. (2005): *Mastering Eclipse Modeling Framework*. EclipseCon 2005, Burlingame
URL: http://www.eclipsecon.org/2005/presentations/EclipseCon2005_Tutorial28.pdf
Abruf: 28.09.2006

[CzHe03] Czarnecki, K.; Simon, H. (2003): *Classification of Model Transformation Approaches*. OOPSLA 2003 Workshop on Generative Techniques in the Context of MDA, Anaheim

[DFK+05] D'Anjou, J.; Fairbrother, S.; Kehn, D.; Kellerman, J.; McCarthy, P. (2005): *The Java Developer's Guide to Eclipse*. 2. Auflage, Boston, Addison-Wesley

[Ecli05a] Eclipse Foundation (2005): *The Eclipse Modeling Framework (EMF) Overview*. EMF Documents
URL: http://dev.eclipse.org/viewcvs/indextools.cgi/*checkout*/org.eclipse.emf/doc/org.eclipse.emf.doc/references/overview/EMF.html
Abruf: 28.09.2006

[Ecli05b] Eclipse Foundation (2005): *org.eclipse.emf.ecore (EMF Javadoc)*.
URL: http://download.eclipse.org/tools/emf/javadoc?org/eclipse/emf/ecore/package-summary.html#details
Abruf: 28.09.2006

[Falk05] Falkowski, K. (2005): *Modelltransformationsansätze im Kontext Modellgetriebener Softwareentwicklung*. Diplomarbeit, Institut für Softwaretechnik, Universität Koblenz-Landau Abteilung Koblenz
URL: http://www.uni-koblenz.de/~ist/documents/falkowski2005da.pdf
Abruf: 28.09.2006

[GoF95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1995): *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA

[JoHo04] Johnson, R.; Hoeller, J. (2004): *J2EE Development without EJB.* Wrox Press, Birmingham

[JoKu05] Jouault, F.; Kurtev, I. (2005): *Transforming Models with ATL*. Model Transformation in Practice Workshop, MoDELS 2005 Conference, Montego Bay, Jamaica
URL: http://sosym.dcs.kcl.ac.uk/events/mtip05/submissions/jouault_kurtev__transforming_models_with_atl.pdf
Abruf: 28.09.2006

[MeSt05] Merks, E.; Steinberg, D. (2005): *From Models to Code with the Eclipse Modeling Framework*. EclipseCon 2005, Burlingame
URL: http://www.eclipsecon.org/2005/presentations/EclipseCon2005_Tutorial11final.pdf
Abruf: 28.09.2006

[OMG02a] Object Management Group (2002): *Meta Object Facility (MOF) Specification*. Version 1.4, April 2002
URL: http://www.omg.org/docs/formal/02-04-03.pdf
Abruf: 28.09.2006

[OMG02b] Object Management Group (2002): *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*. April 2002
URL: http://www.omg.org/docs/ad/02-04-10.pdf
Abruf: 28.09.2006

[OMG03a] Object Management Group (2003): *MDA Guide Version 1.0.1*. Juni 2003
URL: http://www.omg.org/docs/omg/03-06-01.pdf
Abruf: 28.09.2006

[OMG03b] Object Management Group (2003): *UML 2.0 Infrastructure Specification*. Final Adopted Specification, September 2003
URL: http://www.omg.org/docs/ptc/03-09-15.pdf
Abruf: 28.09.2006

[OMG03c] Object Management Group (2003): *Meta Object Facility (MOF) 2.0 Core Specification*. Final Adopted Specification, Oktober 2003
URL: http://www.omg.org/docs/ptc/03-10-04.pdf
Abruf: 28.09.2006

[OMG04] Object Management Group (2004): *Common Object Request Broker Architecture: Core Specification*. Version 3.0.3, März 2004
URL: http://www.omg.org/cgi-bin/apps/doc?formal/04-03-12.pdf
Abruf: 28.09.2006

[OMG05a] Object Management Group (2005): *A Proposal for an MDA Foundation Model*. ORMSC White Paper, Juni 2005
URL: http://www.omg.org/docs/ormsc/05-04-01.pdf
Abruf: 28.09.2006

[OMG05b] Object Management Group (2005): *OCL 2.0 Specification*. Version 2.0, Juni 2005
URL: http://www.omg.org/docs/ptc/05-06-06.pdf
Abruf: 28.09.2006

[Reen79] Reenskaug, T. (1979): *Models – Views - Controllers*. Xerox PARC technical note.
URL: http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf
Abruf: 28.09.2006

[Stra98] Strahringer, S. (1998): *Ein sprachbasierter Metamodellbegriff und seine Verallgemeinerung durch das Konzept des Metaisierungsprinzips*. In: Pohl, K.; Schürr, A.; Vossen, G. (Hrsg.): CEUR Workshop Proceedings zur Modellierung '98, GI-Workshop in Münster, 11.-13. März 1998
URL: http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-9/Strahringer.ps
Abruf: 28.09.2006

[Sun03a] Sun Microsystems (2003): *Enterprise JavaBeans Specification, Version 2.1*

[Sun03b] Sun Microsystems (2003): *Java Servlet Specification, Version 2.4*

[Sun03c] Sun Microsystems (2003): *JavaServer Pages Specification, Version 2.0*

[Sun06a] Sun Microsystems (2006): *JavaServer Faces Specification*. Version 1.2, Proposed Final Draft 2.

[Sun06b] Sun Microsystems (2006): *JSR 220: Enterprise JavaBeans, Version 3.0: EJB 3.0 Simplified API*. Final Release

[Sun06c] Sun Microsystems (2006): *JSR 220: Enterprise JavaBeans, Version 3.0: Java Persistence API*. Final Release

[Sun06d] Sun Microsystems (2006): *JSR 220: Enterprise JavaBeans, Version 3.0: EJB Core Contracts and Requirements*. Final Release

[Wint00] Winter, A. (2000): *Referenz-Metaschema für visuelle Modellierungssprachen*. Deutscher Universitätsverlag, Wiesbaden. Zugl. Dissertation, Institut für Informatik, Universität Koblenz-Landau
URL: http://www.uni-koblenz.de/~winter/papers/winter2000.pdf
Abruf: 28.09.2006